Boosting Practical Control-Flow Integrity with Complete Field Sensitivity and Origin Awareness

Hao Xiang
State Key Lab of ISN
School of Cyber Engineering
Xidian University
Xi'an, China

Zehui Cheng
State Key Lab of ISN
School of Cyber Engineering
Xidian University
Xi'an, China

Jinku Li*
State Key Lab of ISN
School of Cyber Engineering
Xidian University
Xi'an, China

Jianfeng Ma
State Key Lab of ISN
School of Cyber Engineering
Xidian University
Xi'an, China

Kangjie Lu
University of Minnesota-Twins Cities
Minneapolis, USA

ABSTRACT

Control-flow integrity (CFI) is a strong and efficient defense mechanism against memory-corruption attacks. The practical versions of CFI, which have been integrated into compilers, employ static analysis to collect all possibly valid target functions of indirect calls. They are however less effective because the static analysis is imprecise. While more precise CFI techniques have been proposed, such as dynamic CFI, they are not yet practical due to issues on performance, compatibility, and deployability. We believe that to be practical, CFI based on static analysis is still the promising direction. However, these years have not seen much progress on the effectiveness of such practical CFI.

This paper aims to boost the effectiveness of *practical* CFI by dramatically optimizing the target-function sets (aka equivalence class or EC) of indirect calls. We first identify two fundamental limitations that lead to the imprecision of static indirect-call analysis: incomplete field sensitivity due to variable field indexes and the unawareness of the origins of point-to targets. We then propose two novel analysis techniques, *complete* field sensitivity and origin awareness, which handle variable field indexes and distinguish target origins. The techniques dramatically reduce the size of target functions. To enforce the origin awareness, we further employ Intel Memory Protection Keys to safely store the origin information. We implement our techniques as a system called ECCUT. The evaluation results show that compared to the mainline LLVM CFI, ECCUT achieves a substantial reduction of 94.8% and 90.3% in the average and the largest EC sizes. While compared to the state-of-the-art origin-aware CFI (i.e., OS-CFI), ECCUT reduces the average and the largest EC sizes by 90.2% and 89.3% respectively. Additionally, ECCUT introduces an acceptable performance overhead (7.2% on average) observed across a comprehensive range of C/C++ benchmark tests in SPEC CPU2006, SPEC CPU2017, and six real-world applications.

*Corresponding author (email: jkli@xidian.edu.cn).

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670308

CCS CONCEPTS

• **Security and privacy** → **Systems security**; **Software and application security**.

KEYWORDS

Control-flow integrity, Static analysis, Complete field sensitivity, Origin awareness

ACM Reference Format:
Hao Xiang, Zehui Cheng, Jinku Li, Jianfeng Ma, and Kangjie Lu. 2024. Boosting Practical Control-Flow Integrity with Complete Field Sensitivity and Origin Awareness. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670308

1 INTRODUCTION

The direct memory-access capability of C and C++ programs provides excellent performance but also allows memory-corruption attacks. For example, an attacker can tamper with function pointer or return address data in memory through buffer overflow vulnerabilities, thereby reversing the control flow of a program to an illegitimate location. Such memory corruption has been considered the most critical and common attack since 1980s [6].

To address this issue, various defense mechanisms have been proposed, including NX bit [22], stack canary [11], memory randomization [54], control-flow integrity (CFI) [2], memory safety [3], etc. Among these mechanisms, CFI is a particularly promising one because it provides a strong defense against memory-corruption attacks, and more importantly, it is practical—claimed to be less than 1% of runtime overhead [4], integrated into compilers [55], and adopted by major software vendors such as Microsoft and Google [41, 55].

The idea of CFI is to ensure that the control flow, which determines the order in which the instructions of a program are executed, adheres to predefined rules and constraints, known as the control-flow graph (CFG). Since the introduction of CFI, a stream of CFI solutions have been designed [4–6, 12, 15–21, 23, 25, 28–30, 35, 39, 40, 42–45, 47, 55, 57, 60–63]. In the early stage, a number of CFI systems mainly use coarse-grained CFGs to enforce protection [2, 55, 62, 63]. However, subsequent research shows that coarse-grained CFIs can be bypassed by well-designed attacks [7, 10, 17, 23]. In response, later researchers adopt flow-sensitive and field-sensitive approaches to implement fine-grained CFI protection [21, 35, 40, 47], which improves the security of programs. Unfortunately, the existence of multiple jump targets for an indirect call is a common case in generic programs, which can be exploited by advanced attackers to implement control flow "bending" between equivalence classes (ECs) [6, 36]. Note that an EC is a set of targets for an indirect control transfer (ICT) [1] that are indistinguishable from each other. In other words, CFI is not able to identify control-flow deviations inside an EC.

Overall, recent CFI techniques can be classified into two categories. The first category typically generates CFGs through static analysis and verifies the legitimacy of the target when an ICT occurs. The static analysis employs either point-to analysis or type-based analysis to conservatively find all possible targets of indirect calls. As this category of CFI has been integrated into compilers and adopted by major software vendors, we refer to this category as *practical CFI*. The second category focuses on further reducing the size of target functions by relying on dynamic analysis when more information is available. We refer to this category as *dynamic CFI*. In general, the dynamic CFI offers improved security, but its practicality may be constrained by factors like performance, compatibility, and deployability. For instance, to achieve high-precision CFG, μCFI [25] records an extensive amount of contextual information with Intel process tracer (PT), causing PT to lose packets. This leads to μCFI impractical for the programs with substantial codebases. While PathAmror [57] utilizes Intel processor's Last Branch Record (LBR) to record program-specific execution paths but is limited by the fact that LBR can only record the last sixteen branches, which limits its ability to generate dynamic CFGs and allows it to focus protection only on critical system calls.

We believe that to be practical, CFI based on static analysis is still the promising direction. Existing compilers and software vendors all adopt this kind of CFI [41, 55]. That said, a major concern with practical CFI is its effectiveness resulting from the imprecise static analysis which leads to large EC, i.e., an over-approximation of control-flow transfers which allows more indirect call targets than there should be [4]. Critically, these years have not seen much progress on the effectiveness of such practical CFI.

In this paper, we propose a new approach, called ECCut, which boosts the effectiveness of practical CFI by greatly reducing the average and the largest EC sizes with novel analysis techniques. In particular, we first conduct an empirical analysis to identify the major causes of the imprecision of static analysis. We found that existing static analysis claims to be field-sensitive; in reality, their field sensitivity is far from complete due to the common variable indexes in struct accesses. We observe that even the state-of-the-art pointer-to analysis tools (e.g., SVF [53]) and type-based analysis [40] are field-insensitive in many cases—when there is a variable index in struct access (which is very common), they downgrade to field-insensitive analysis. This issue amplifies exponentially with a higher number of variable indexes, leading to imprecise analysis results. In addition, existing static analysis (both point-to analysis and type-based analysis) is unaware of the origins (i.e., sources) of function pointers and conservatively combines all possible targets regardless of origins. Even worse, the origin tracking for ICTs is essentially a taint analysis process. Although function addresses are usually not as widespread as the regular data (e.g., a network packet) [51], it is still a path explosion problem.

To address the problems, we propose *complete* field sensitivity and origin awareness. Note that the concept of "completeness" here means when a function pointer variable is located in an array or a (nested) sub-field of a struct, ECCut can analyze it to the ultimate location to get its precise value, regardless of whether its index is a constant or a variable. Thus, our *complete* field sensitivity supports both constant and variable indexes. We develop an optimization mechanism to properly integrate runtime acquisition of variable indexes; the runtime acquisition is minimized and placed in a location with the highest EC reduction, which refers to the specific index location selected by us to achieve the optimal reduction in EC size. On the other hand, our novel origin-awareness approach employs static analysis to trace program path information backwardly from the ICT and strives to identify paths containing function pointer assignments. This approach helps circumvent the performance overhead incurred by parsing a large volume of path information at runtime and contributes to a reduction in average EC size.

To validate our approach, we develop a prototype of ECCut with LLVM [33] compiler and SVF [53]. In addition, as a *complete* field-sensitive and origin-aware CFI, we leverage the Intel Memory Protection Keys (MPK) [27, 46] to record the variable indexes in struct accesses and protect the origin information. Our evaluation with standard benchmarks, real-world applications, and real exploits shows a significant reduction in both the average and the largest EC sizes, which can effectively defend against control-flow hijacking attacks. In comparison to the mainline LLVM CFI [55], ECCut achieves a remarkable 94.8% reduction in the average EC (from 32.4 to 1.7 on average) and a 90.3% reduction in the largest EC (from 154 to 15 on average). While compared to the state-of-the-art origin-aware CFI system [30], ECCut reduces the average EC by 90.2% (from 17.3 to 1.7 on average) and the largest EC by 89.3% (from 140.2 to 15 on average). In addition, our approach incurs an acceptable performance overhead (7.2% on average) observed across SPEC CPU2006, SPEC CPU2017, and six real-world applications. To engage the community, we will release the source code of ECCut at https://github.com/XDU-SysSec/ECCut.

In summary, our paper makes the following contributions:

- We identify two fundamental limitations with existing practical CFI and propose two novel techniques to address them: complete field sensitivity and origin awareness for significantly reducing the EC size of indirect-call targets.
- We implement a prototype of ECCut, which constructs highly accurate CFGs through static path-based origin analysis and complete field analysis. Additionally, we leverage Intel MPK technology to safeguard the runtime origin information.

[1]In this paper, we use ICT to indicate forward-edge control-flow transfers.

- We thoroughly evaluate the security and performance of ECCut using standard benchmarks, real-world applications, and real exploits. The results show that ECCut significantly reduces the average and the largest EC sizes with an acceptable performance overhead.

2 BACKGROUD AND MOTIVATION

2.1 Practical CFI vs. Dynamic CFI

We define practical CFI as the ones that use static analysis to resolve indirect-call targets, which have been integrated into compilers [55]. Such CFI does not require expensive dynamic analysis or heavyweight code instrumentation; therefore, they tend to be highly efficient (e.g., as low as 1% of runtime overhead according to LLVM CFI) and easily deployable. Note that such CFI might record some (light) facts at runtime to assist in later verification [30].

We define dynamic CFI as the ones that rely on dynamic analysis to precisely determine the correct target of indirect calls. The dynamic CFI has the following limitations that impede their practical utility. First, dynamic CFIs [18, 25, 45] rely on a significant number of contexts to generate dynamic CFGs, introducing additional performance overhead to the system. Second, certain CFIs [18, 25, 57] that employ dynamic methods may require specific hardware features like LBR and PT support, which restrict their applicability to systems without these features. Third, due to constraints in system design, dynamic CFIs are often limited in their ability of protecting specific objects or components. For example, PittyPat [18] utilizes PT to record execution path and constraint data information that is so voluminous as to lose packets, which prevents it from being applied to large programs. The protections of PittyPat, PathAmror, and μCFI only cover selected syscalls.

2.2 Field Insensitivity of Practical CFI

We observe that the primary reason for large EC sizes lies in the presence of a large number of function pointer-typed fields within nested structs. Consequently, a natural approach to address this issue is to employ a field-sensitive policy. Nevertheless, the state-of-the-art analysis tools (e.g., SVF [53] and MLTA [40]) with field sensitivity do not adequately decompose the largest EC. The fundamental problem is that *their analysis falls back to field insensitivity whenever a variable index is encountered, which is common.* We demonstrate this issue with the example as shown in Figure 1.

```
1  typedef int (*EVALFUNC)(int sq, int c);
2  static EVALFUNC evalRoutines[7]={
3  ErrorIt,
4  Pawn,
5  Knight,
6  King,
7  Rook,
8  Queen,
9  Bishop
10 };
11 int std_eval(int alpha, int beta){
12   for(j=1, a=1; (a<=piece_count); j++){
13     score +=
14     (*(evalRoutines[piecet(i)]))
15     (i,pieceside(i));  //ICT
16   }
17 }
```

Figure 1: The indirect call and targets in `458.sjeng`.

Specifically, `458.sjeng` is a benchmark program written in *C* language from SPEC CPU2006, which is designed for playing chess and various chess variants. It features only one ICT located at lines 13-15. The function pointer within this ICT is composed of array `evalRoutines` and variable `piecet(i)` as an offset. Array `evalRoutines` hosts a total of seven targets from line 3 to line 9. However, since `piecet(i)` is a variable, certain tools like SVF degrade to field insensitivity when analyzing this ICT, resulting in an EC size of 7.

An even more concerning scenario arises when the function pointer is located in a nested struct with multiple layers. In such cases, the analysis results grow exponentially with an increase in the number of variable offsets. To tackle this issue, we introduce a *complete* field-sensitive policy. This approach records the variable offset values (depicted as `piecet(i)` in Figure 1) at runtime and combines it with the CFG generated through static analysis, effectively segmenting the largest EC of the program. Our initial study shows that it is able to break down the largest EC in `445.gobmk` benchmark from 1637 to 14 after introducing a *complete* field-sensitive policy to the analysis, which significantly reduces the potential attack surface (see details in Section 5.1). Among the two typical benchmarks used for field-insensitive analysis, `400.perlbench` exhibits the largest EC size of 349, while `445.gobmk` has the largest EC size of 1637. Our study indicates that the function pointers located in nested structs of `445.gobmk` are all affected by variable indexes; and 22% of function pointers within structs in `400.perlbench` are affected by variable indexes, and more than half of these pointers have 349 targets.

2.3 Origin Unawareness of Practical CFI

In the following, we illustrate the limitations of origin-unaware CFIs using a real program (i.e., `400.perlbench` benchmark program from SPEC CPU2006) as depicted in Figure 2. Within this figure, there exists an ICT situated at line 2, and the function pointer of this ICT is `compare`, which serves as an argument to function `S_qsortsvu`. This ICT comprises multiple origins, and we only present three representative ones in the figure: at lines 5, 8, and 24, labeled as `origin1`, `origin2`, and `origin3`, respectively.

During program execution, the function pointer `compare` acquires different values from distinct origins contingent upon the specific circumstances. For instance, it can receive the target `cmpindir_desc` or `cmpindir` from `origin1` depending on the value of `flags`. Alternatively, `compare` can obtain the target `cmp_desc` from `origin2`, which also includes just one direct call in line 9. In the case of `origin3`, `compare` can take targets between `sortcv_xsub`, `sortcv_stacked`, and `sortcv`, which is determined by the values of `is_xsub` and `hasargs`.

The presence of these three origins puts origin unawareness CFIs into an invalid state. For example, CFI-LB [29] is very weak against this function pointer, which assumes that the function pointer can take any value. OS-CFI [30], although somewhat origin-aware, can only recognize `origin2`, and it is not aware of the existence of `origin1` and `origin3`. Consequently, it will lead to the reduction

```
STATIC void S_qsortsvu(..., SVCOMPARE_t compare) {
  s = compare(...);  //ICT
}
STATIC void S_qsortsv(..., SVCOMPARE_t cmp, U32 flags) {
  S_qsortsvu(..., flags ? cmpindir_desc : cmpindir);  //origin1
  if (...) {
    cmp = cmp_desc;  //origin2
    S_qsortsvu(..., cmp);
  } else {
    S_qsortsvu(..., cmp);
  }
}
void Perl_sortsv(..., SVCOMPARE_t cmp) {
  void (*sortsvp)(..., SVCOMPARE_t cmp, U32 flags) = S_mergesortsv;
  if (...) {
    sortsvp = S_qsortsv;
  else
    sortsvp = S_mergesortsv;
  sortsvp(..., cmp, 0);
}
OP* Prel_pp_sort(){
  void (*sortsvp)(..., SVCOMPARE_t cmp) = Perl_sortsv;
  sortsvp(..., is_xsub ? sortcv_xsub : hasargs ? sortcv_stacked : sortcv); //origin3
}
```

Figure 2: The indirect call and targets in `400.perlbench`.

of EC size (from 6 to 2 on average) if being aware of `origin1` and `origin3`.

3 SYSTEM DESIGN

3.1 Overview

Threat Model and Assumptions. In this work, we assume non-writable code (NWC) and non-executable data (NXD), as the original CFI [2] does, thus attackers cannot modify code memory at runtime, or execute data as if it were code. Meanwhile, we assume that attackers have full access to the memory space and can tamper with function pointer data in arbitrary writable areas. Also, our system is designed to be open and transparent to attackers. Further, as we leverage MPK to safeguard runtime context information, we assume that MPK protection cannot be bypassed, e.g., by leveraging unsafe *WPKRU* or *XRSTOR* instructions or OS abstractions demonstrated by previous researches [9, 24, 48, 56, 59]. Thus, we assume that the MPK protection is trustworthy and its security limitations are out of scope. Moreover, side-channel attacks are also out of scope in this work.

To achieve the goal of defending against control-flow hijacking attacks within the threat model, the entire system is divided into two distinct parts: static analysis and runtime verification. The static analysis phase primarily employs complete field sensitivity and origin awareness to create CFGs. Specifically, we first utilize the practical origin-aware policy to analyze the IR and identify the origins. Then, we employ SVF [53] to analyze the function pointers from the identified origins and generate CFGs. Finally, we enhance the CFGs with complete field sensitivity. The runtime verification phase is responsible for using runtime origin information to validate whether the jump targets of ICTs are within the CFG.

Table 1: Target sets corresponding to all possible cases of `offset1` and `offset2`.

(offset1, offset2)	Target Set
(0, 0)	set1
(0, 1)	set2
(0, 2)	set3
(1, 0)	set4
(1, 1)	set5
(1, 2)	set6

3.2 Static analysis

3.2.1 Practical Origin-Aware Policy. Our practical origin-aware policy aims to split the EC of an indirect call based on the origin of the function pointer, which will reduce the EC size. Our key observation here is from the intuition that the actual target of an indirect call is based on the sources of the function pointer—what values are assigned to it. A natural solution would be employing a path-sensitive analysis, which is however impractical due to path explosion.

To address this problem, we introduce a new origin-aware policy. In particular, for the C-type indirect call, our practical origin-aware policy analyzes the def-use chain of function pointers to identify origins, which consist of assignments to function pointers or structs containing function pointers. To the greatest extent, the policy ensures that these assignments are original assignments to function pointers. An "original assignment" means that the value of the function pointer in the assignment instruction is an explicitly defined function or an initialized global variable, allowing for the direct identification of the jump target for the ICT passing through this origin.

When handling the C++ virtual calls, we can directly determine that the origins are in the constructor of the class in which the virtual call pointer is located. This is because the virtual functions in a class are stored in a virtual table. When a class object is allocated, the virtual table is assigned to the member variables of the object in the constructor. It is worth pointing out that some virtual calls do not follow the assignment method described above, in which case we can handle them as we do with indirect calls.

3.2.2 CFG Construction. Undoubtedly, the CFG plays a critical role in the security of the CFI system. This is because any errors (e.g., a false negative) within the CFG can potentially result in issues within the CFI system, thus disrupting the execution of the entire program. Consequently, the creation of accurate and highly precise CFGs is very important. Ideally, if all the origins identified by the practical origin-aware policy consist of original assignments, a flawless CFG can be directly generated. However, factor considerations lead to the situation that not all the origins are original assignments (see detail in Section 4.3). This necessitates the use of a pointer analysis tool to construct the CFG. As a result, we select SVF [53], a precise static points analysis tool that claims to be context-, flow-, and field-sensitive, to generate the CFG.

Although SVF only needs to generate CFGs for source points that are not original assignments, there are still two problems.

First, SVF may encounter difficulties in analyzing certain benchmarks(`400.perlbench`, `403.gcc`, `445.gobmk`, `447.dealII`, `450.soplex`, `453.porvray`, `471.omnetpp`, `483.xalancbmk`) [30]. Specifically, SVF may return wrong results in the points-to sets (e.g., functions with wrong signatures) and return empty results because of language features it does not support (e.g., C++'s pointers to member functions). *Second*, although the practical origin-aware policy is looking for all origins as much as possible, there are still some ICTs whose origins cannot be successfully obtained. In such cases, we utilize type-based matching as a last resort to ensure that the CFG has no false negatives, although this approach may result in a larger EC size. Fortunately, the percentage of such ICTs is small (see details in Section 5.1).

Next, we give the composition of CFG tuples. Due to the introduction of type-based matching, in total, we have three forms of CFG tuples:

- **(ICT_id, Path_id, Target):** This tuple is for the practical origin-aware policy. **ICT_id** denotes the marker of the indirect call. **Path_id** denotes the marker of the origin information. And **Target** denotes the value of the function pointer.
- **(ICT_id, Path_id, Target, Offset):** This tuple is for *complete* field-sensitive policy as described in Section 3.2.3. **ICT_id**, **Path_id**, and **Target** denote the same as the practical origin-aware policy. And **Offset** denotes the variable offset.
- **(ICT_id, Target):** This tuple is for the CFG of the ICT that uses type-based matching.

3.2.3 Complete Field-Sensitive Policy. As the state-of-the-art point-to analysis tool, SVF [53] claims to be field sensitive. However, its field sensitivity will degrade to field insensitivity when it encounters function pointers whose values are determined by the variable offsets of the struct. This leads to a great largest EC (e.g., 1637 in `445.gobmk`) in the CFGs generated by SVF, so we need to enhance the generated CFGs by introducing a *complete* field-sensitive policy.

We assume the presence of a function pointer within a struct, characterized by two variable offsets: `offset1` which ranges from 0 to 1, and `offset2` which ranges from 0 to 2. Table 1 shows the set of targets corresponding to all their possible values. This information can also be derived through static analysis of the global variable housing the function pointer. An ideal scenario would involve recording the values of `offset1` and `offset2` separately at runtime, then matching these values to their respective target sets and verifying if the function target belongs to the set before an indirect call. However, this solution comes with a considerable performance cost, particularly when the number of offsets increases: the overhead of storing the offsets at runtime grows linearly, and the overhead of looking up the corresponding target sets escalates exponentially.

To achieve a balance between performance overhead and security, our *complete* field-sensitive policy selects an offset that minimizes the largest EC during the static analysis phase. We only record the value of this offset at runtime. It is worth pointing out that the removed contexts are not redundant and it reduces the security without that. However, we believe it is an optimal choice for combined consideration of security and performance. For `offset1`, its largest EC is equal to max($\{set1,\ set2,\ set3\}$, $\{set4,\ set5,\ set6\}$). For `offset2`, its largest EC is equal to max($\{set1,\ set4\}$, $\{set2,\ set5\}$, $\{set3,\ set6\}$). If `offset1` has the smaller largest EC, we change the CFG of this ICT from (**ICT_id**, **Path_id**, $\{set1,\ set2,\ set3,\ set4,\ set5,\ set6\}$) to (**ICT_id**, **Path_id**, $\{set1,\ set2,\ set3\}$, 0) and (**ICT_id**, **Path_id**, $\{set4,\ set5,\ set6\}$, 1). Otherwise, if `offset2` has the smaller largest EC, we change the CFG of this ICT from (**ICT_id**, **Path_id**, $\{set1,\ set2,\ set3,\ set4,\ set5,\ set6\}$) to (**ICT_id**, **Path_id**, $\{set1,\ set4\}$, 0) and (**ICT_id**, **Path_id**, $\{set2,\ set5\}$, 1), **Path_id**, $\{set3,\ set6\}$, 2). When the program executes, we can leverage the recorded value of `offset1` or `offset2` to determine which CFG tuples should be compared to the jump targets of this ICT.

When there is only one variable offset, we directly select it as the context information for runtime recording; when there are multiple offsets, we can use the above method to select an appropriate offset as the context information for runtime recording. By employing the field-sensitive policy, we significantly break down the largest EC. For instance, we have decomposed the EC of 1637 in `445.gobmk` into 14 (see details in Section 5.1).

3.3 Runtime Verification

Runtime validation is a critical step to ensure that the jump targets of ICTs have not been tampered with. It leverages the collected runtime information in conjunction with the CFG to ascertain the legitimacy of the targets. ECCut employs a hash table named runtime table to store runtime context information. Entries in this table are indexed by the hash of the function pointer address, which is computed by the variant xxhash algorithm.

The validation process consists of two steps. *First*, the system verifies whether the function pointer address and the target match within the runtime table. This step ensures that the function pointer has not been tampered with from the origin to the indirect callsite. *Second*, the `Path_id` and `Offset` (if exists) values are retrieved from the runtime table. For practical origin-aware policy, this includes origin information, while *complete* field-sensitive policy involves the offset and origin information. These pieces of data are combined to create a tuple, which is then one-to-one correspondence within the CFGs to confirm its existence. This process effectively identifies tampering of function pointer values that may have occurred before origins. However, note that this approach cannot detect tampering of jump targets within an EC [36].

3.4 Metadata Storage

Throughout the ECCut system design, two types of data need to be protected: CFGs generated by static analysis and runtime contextual information stored in the runtime table. For CFG data, we can just keep it in read-only memory as the previous CFI systems have done to ensure security. But this does not work for the runtime context information, because we have to update those data in real-time while the program executes. Protecting data that is readable and writable at runtime is always a challenge. To achieve that, we use the Intel MPK [27, 46] feature to protect runtime contextual information in the runtime table.

Specifically, MPK [27] can protect memory by setting access rights to memory pages. The runtime information of the program is stored in the runtime table, storing the tuple **(Ptr_addr, Target, Path_id, Offset)**. When the program executes, we request a large enough piece of memory and set it as access-disabled; when we need to record context information, we set it as write-disabled, and check the corresponding hash indexes to avoid the hash collision. While it is possible to access the corresponding entry for **Ptr_addr** based on its hash value, the challenge lies in distinguishing whether this hash corresponds to **Ptr_addr** or is the result of a hash collision caused by some other address, so we keep the function pointer in the runtime table. If the hash value comes from the other address, we save it to the next tuple where **Ptr_addr** is empty or to the same location as its function pointer address. When we insert the runtime table, we set the memory as writable, update the tuple information, and reset the memory as access-disabled. When performing the first step of verification, we set the memory to write-disabled, read the information, and reset the memory to access-disabled at the end of the first step of verification.

Note that in the above process, frequent modification for the memory page access permission has been involved, which aims to minimize the exposure of context information within the runtime tables. This precaution is taken because the attacker can potentially access any unprotected memory. Such access allows the attacker to retrieve the CFGs stored in read-only memory; if they acquire the `path_id` from the runtime table, they can manipulate the control flow within the EC without being detected. Although this requires frequent memory page state changes, the performance consumption is not significant, because `RDPKRU` and `WRPKRU` instructions that read and write PKRU (protection key rights for user pages) are not privileged and thus can be executed in user space without context switching.

4 IMPLEMENTATION

4.1 Static Analysis for Origin Awareness

Our static analysis process is a depth-first traversal along the def-use chain of function pointers at indirect calls, to find all assignments of function pointers and structs where function pointers are located. These assignments are origins. We illustrate our practical origin-aware policy with a simple example. Figure 3a is a sample of a C source code file named *example.c*. Lines 1-2 show two callee functions, i.e., `calleei` and `calleej`. The indirect call in line 10 uses the function pointer `fp`, which can take its value through two origins. The first origin is line 8 with the value `calleei`. The second origin is line 17 with the value `calleej`.

The situation becomes more complicated when we analyze the function pointer `fp` in IR with the practical origin-aware policy. This is because the def-use chain of function pointers resembles a tree structure, with the function pointer at the end leaf node and the `alloca` instruction being the root node of the tree, as determined by the static single assignment nature of IR. We need to first backtrack from the function pointer to the `alloca` instruction, and then recursively traverse all the uses of the `alloca` instruction and find all assignment nodes related to the function pointer.

We next demonstrate this process with Figure 3b. The variable `%7` is the function pointer of ICT in line 12. The root node of `%7` is the variable `%2`. There are three uses of `%2`: the `store` instruction in line 3, the `store` instruction in line 8, and the `load` instruction in line 11. Line 11 is the incoming site and does not need to be analyzed. Line 8 is the origin of `%2` within the function `caller`. Once we have acquired an origin, we determine whether it dominates the ICT to determine the subsequent analysis process. For our analysis, the fact that the origin dominates the ICT means that all paths to the ICT must pass through the origin. In this case, the traversal for the other uses of `alloca` can be finished. But this does not mean the end of the analysis if the origin is not the original assignment. We apply a practical origin-aware policy to the value of the origin until the original assignment is found. It is obvious that line 8 is an original assignment and does not dominate line 12. The remaining use of `%2` is in line 2, which assigns the argument `%0` to `%2` and leads us to look for `call` instructions to `caller` in other functions. Notice that there are only two `call` instructions for `caller`: lines 16 and 17, which assign `%0` with the values `null` and `calleej`, respectively. The origin of line 16 will be overwritten by the origin of line 8 at runtime as the two origins are on the same path.

The above process is mainly the practical origin-aware analysis within one function. Then we discuss the cross-functional process. In our study, we identified cyclic calls across functions as the primary challenge encountered during the analysis process. Next, we show the analysis process of the cross-function process with Figure 4. In the function `callee1`, there is an ICT with the function pointer `%2`, which value comes from the argument `%0`. So we find that function `callee2` calls function `callee1` with argument `%x2` (step 1). We call this type of cross-functional call-style. And the value of `%2` comes from the call instruction `%x1`. So we find the called function `callee3` in instruction `%x1` and get its return instructon `ret %y1` (step 2). We call this type of cross-function ret-style.

One difficulty in cross-functional analysis, which is shown in Figure 4, is the inter-call between two functions. In the above analysis, we go through a call-style and a ret-style cross-functions. In the function `callee3`, we start from the `ret` instruction to a ret-style cross-function for `callee2` (step 3). When entering `callee2` again, we start from the `ret` instruction for the repeated ret-style cross-function for `callee3` (step 2). Then we have a chain of infinite loops: $1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 2 \longrightarrow 3 \longrightarrow \cdots$, which is impossible to analyze completely.

To solve this problem, we give an assertion based on the flow of intra-functional backtracking: any analysis starting from the same position in the same function is equivalent. Thus, we only need to determine whether an analysis of that function was previously performed at the same position before a new cross-function to solve any loop. And even in the worst case, we only need to perform a finite number of analyses for all functions in the IR. The case of direct calls is discussed previously, while for an indirect call, we type-match the indirect call and backtrack through all matched functions.

4.2 Path Explosion for Origin Awareness

Solving the looping calls problem can alleviate but not solve the path-explosion problem. As we increase the number of layers of backtracking, the path-explosion problem still occurs.

We investigated the path explosion problem on SPEC CPU2006 benchmark programs, Httpd, Lighttpd, Nginx, and Redis during our practical origin-aware analysis. The results are shown in Table 2. In the table, the *ICTs* column indicates the number of ICTs that can be origin-aware analyzed (note that some ICTs may lack an origin).

```c
void calleei(int i) {...};
void calleej(int j) {...};

void caller(void (*fp)(int))
{
  if(fp==NULL)
  {
    fp=calleei;
  }
  fp(0);
  return 0;
}

void main()
{
  caller(NULL);
  caller(calleej);
  return 0;
}
```

(a) example.c

```llvm
define dso_local void @caller(void (i32)*) #0 {
  %2 = alloca void (i32)*, align 8
  store void (i32)* %0, void (i32)** %2, align 8
  %3 = load void (i32)*, void (i32)** %2, align 8
  %4 = icmp eq void (i32)* %3, null
  br i1 %4, label %5, label %6
; <label>:5:                        ; preds = %1
  store void (i32)* @calleei, void (i32)** %2, align 8
  br label %6
; <label>:6:                        ; preds = %5, %1
  %7 = load void (i32)*, void (i32)** %2, align 8
  call void %7(i32 0)
  ret void
}
define dso_local void @main() #0 {
  call void @caller(void (i32)* null)
  call void @caller(void (i32)* @calleej)
  ret void
}
```

(b) the LLVM IR of example.c

Figure 3: An example for practical origin-aware policy.

Table 2: Path information during practical origin-aware analysis.

Benchmark	ICTs	1st-layer	Original	Percent	Avg EC	2nd-layer	Path growth
400.perlbench	136	545	185	33.9%	2.8	34601	96.1X
401.bzip2	20	29	9	31.0%	1	2810	140.5X
403.gcc	425	1136	715	62.9%	2.9	1010	2.4X
433.milc	4	8	8	100.0%	1	0	-
444.namd	12	30	30	100.0%	1	0	-
445.gobmk	44	652	604	92.6%	1.7	1412	29.4X
447.dealII	165	344	335	97.4%	1	9	1.0X
450.soplex	512	1255	1234	98.3%	1	21	1.0X
453.porvray	165	11496	440	3.8%	2.8	153688	13.9X
456.hmmer	9	46	8	17.4%	1	2048	53.9X
458.sjeng	1	1	1	100.0%	1	0	-
464.h264ref	367	2243	1875	83.6%	1.1	366281	995.3X
471.omnetpp	673	15112	1802	11.9%	1.2	12401	0.9X
473.astar	1	1	1	100.0%	1	0	-
482.sphinx3	2	10	10	100.0%	1	0	-
483.xalancbmk	7480	394363	392849	99.6%	1.0	6351	4.2X
Httpd	194	589	34	5.8%	2.2	8134	14.7X
Lighttpd	109	400	391	97.8%	1.0	33	3.7X
Nginx	313	3077	2159	70.2%	1.7	3705	4.0X
Redis	554	6830	2991	43.8%	1.9	4842	1.3X
Average	559	21908.4	20289.1	92.6%	1.5	29867.3	90.8X

The *1st-layer* column indicates the assignment number of the function pointer (or the struct containing the function pointer) closest to the ICTs. The *Original* column indicates the number of original assignments, with their percentage of the *1st-layer* column's number shown in the *Percent* column. Note that an original assignment of a function pointer means that the value of the function pointer in the assignment instruction is an explicitly defined function or an initialized global variable. The *Avg EC* column indicates the average EC size of these ICTs when only the 1st-layer analysis is performed. The *2nd-layer* column shows the number of paths if we continue the analysis after eliminating all the original assignments, and the *Path growth* column indicates the expansion multiplier of the path number. In the result, we see that the average EC decreases to less than 3 with just the 1st-layer analysis, which is undoubtedly a positive outcome. One reasonable explanation is that the origins analyzed by the 1st-layer already correspond to the original assignments. Further analysis indicates that when employing just the 1st-layer analysis, approximately 92.6% of the assignments are the original assignments. In contrast, the average number of paths inflates by more than 90 times when ECCut performs the 2nd-layer analysis. The worst case occurs in `464.h264ref`, which experiences

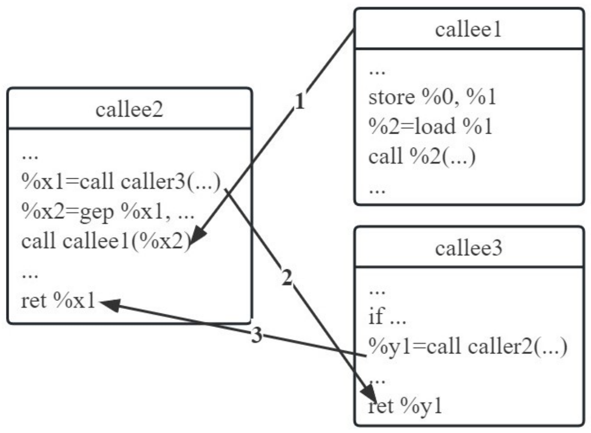

Figure 4: Conceptual diagram of a cross-functional loop.

```
1 %6 = call i32 @make_gather(void (i32, i32, i32, i32, i32*, i32,
↪  i32*, i32*, i32*, i32*)* @third_neighbor, i32* %1, i32 1, i32
↪  0, i32 1)

2 %816 = select i1 %814, i32 (%struct.sv*, %struct.sv*)*
↪  @sortcv_stacked, i32 (%struct.sv*, %struct.sv*)* @sortcv
3 br label %817
4 ; <label>:817:          ; preds = %811, %810
5 %818 = phi i32 (%struct.sv*, %struct.sv*)* [ @sortcv_xsub, %810
↪  ], [ %816, %811 ]

6 sortsvp(aTHX_ start, max, is_xsub ? sortcv_xsub : hasargs ?
↪  sortcv_stacked : sortcv);
```

Figure 5: Path without function pointer address.

an inflated number of paths by 995.3 times. As a result, we decide to analyze only the 1st-layer to avoid the path explosion problem.

4.3 Origin Awareness Context

The origin awareness context consists of the tuple **(Ptr_addr, Target, Path_id)**. Specifically, **Ptr_addr** represents the address of the function pointer, and **Target** is the value of the function pointer in this tuple. These two values are closely related, and we can extract them from the IR and retrieve them at runtime. **Path_id** serves as the marker for the origin, which is for distinguishing from other origins. Since instructions with the same by string may exist in different functions within the IR, we derive the hash value of all strings within this origin and the name of the function as it is in the **Path_id**.

The above description is for the ideal case of instrumentation, but in practice, we face many challenges because of the wide variety of paths. Specifically, not all paths happen to be assignments to function pointers. There are a lot of assignments to structs or even nested structs where function pointers are located in the paths we backtrack. To solve this problem, we need to get the real function pointer address and function pointer value. This is possible because the LLVM IR supports modifications to the source code and the compiler provides several functions to allow us to modify the IR. We can record all the instructions on the path during backtracking, and rebuild the function pointer corresponding to the ICT by these instructions.

Figure 5 shows the special cases where we cannot find the corresponding function pointer address with IR. This happens mainly when the program passes the function name as a real parameter as shown in line 1. This is a real-world example from `433.milc` benchmark program. The function name `third_neighbor` is passed as a function parameter, which prevents us from creating a variable that can access the stack during static analysis. In this case, we have to replace the **Ptr_addr** with the **Path_id**. If this path is a common path for multiple ICTs, then we need to perform instrumentation several times before the `call` instruction, which increases the performance overhead.

The other two cases without function pointer addresses are caused by the `select` (line 2) and `phi` (line 5) instructions in IR, but these instructions are still for fundamental parameter passing during function calls. Line 6 shows a real-world function call in `400.perlbench` benchmark program, which is the prototype of the instructions in lines 2-5. The `phi` and `select` instructions are the nested conditional expressions in IR. We handle these two cases in the same way as the previous ones; the only difference is that the instrumenting place is after the instruction. The reason we do not continue to analysis is that it is impossible to get the function pointer address here; and if we continue to use **Path_id** instead of **Ptr_addr**, it will bring a big impact on performance since this path corresponds to more than one ICT.

4.4 *Complete* Field-Sensitive Context

The field context is **(Ptr_addr, Target, Path_id, Offset)**. The **Ptr_addr**, **Target**, and **Path_id** are processed in the same way with the origin-aware context. The field context has **Path_id** because the combination of *complete* field sensitivity and practical origin awareness has a better EC reduction (see Section 5.1).

Staking **Offset** becomes a challenge because the block where the offset is located is not the same as the block where the assignment is located. Since the offset is a variable, we have to deal with complications. If the offset and the origin are in the same function, we can reproduce the value of the offset after the origin; if not, we have to make a trade-off between the origin awareness and field sensitivity, since cross-function assignment of variables is almost impossible at the IR level. Finally, we drop the origin and use the `GEP` instruction where the offset is located as the new origin, which may increase EC size if multiple paths are traced back after the `GEP` instruction. However, this does not affect the existence of `Path_id` because an ICT can have both the origin context and the field context in different paths. So we put the origin and the field context validation in one verification function and the runtime table only needs to hold one form of tuple.

5 EVALUATION

5.1 EC Reduction

Our evaluation dataset includes C/C++ benchmarks of the SPEC CPU2006 and SPEC CPU2017 suites, and six real-world applications, namely `Httpd` (the Apache HTTP server, v.2.4.58), `Lighttpd` (a lightweight web server, v.1.4.60), `Nginx` (a web server, usable also as a reverse proxy, load balancer, mail proxy, and HTTP cache, v.1.20.2),

Table 3: The overall statistics of ECCut in all evaluation programs.

Benchmark	ICTs	Origin awareness and Field sensitivity							Type-base			Overall	
		ICTs	Only origin		Only field		Field and origin		ICTs	Avg	Lg	Avg	Lg
			Avg	Lg	Avg	Lg	Avg	Lg					
400.perlbench	137	128	10.6	349	7.0	42	2.8	42	9	0.8	7	2.8	42
401.bzip2	20	20	1	1	1	1	1	1	0	0	0	1	1
403.gcc	442	404	5.8	218	3.5	40	2.9	40	38	7.8	43	3.4	43
433.milc	4	4	1	1	2	2	1	1	0	0	0	1	1
444.namd	12	12	1	1	2.5	3	1	1	0	0	0	1	1
445.gobmk	44	35	250.2	590	7.3	14	1.7	14	9	0.9	8	1.9	14
447.dealII	167	160	1	2	1	2	1	2	7	2.6	15	1.1	15
450.soplex	513	507	1	1	1	1	1	1	6	3.2	5	1.0	5
453.porvray	173	151	2.8	5	3.0	30	2.8	5	22	4.3	15	3.2	15
456.hmmer	9	9	1	1	2.8	10	1	1	0	0	0	1	1
458.sjeng	1	1	7	7	1	1	1	1	0	0	0	1	1
464.h264ref	367	367	1.1	2	1.1	12	1.1	2	0	0	0	1.1	2
471.omnetpp	709	615	1.5	168	1.2	43	1.2	43	94	0.3	9	1.2	43
473.astar	1	1	1	1	1	1	1	1	0	0	0	1	1
482.sphinx3	2	2	1	1	5	5	1	1	0	0	0	1	1
483.xalancbmk	8135	7113	1.0	29	1.0	29	1.0	29	1022	0.9	29	1.1	29
500.perlbench	239	206	3.8	47	1.9	11	1.3	11	33	7.3	60	2.1	60
502.gcc	3150	2734	20.6	506	2.6	28	2.6	28	416	30.4	49	6.3	49
505.mcf	14	14	1	1	1.6	2	1	1	0	0	0	1	1
508.namd	12	12	1	1	4.3	6	1	1	0	0	0	1	1
510.parest	1126	1048	1.3	6	1.9	47	1.3	6	78	4.3	27	1.5	27
520.omnetpp	6416	5742	1.2	1	1.3	274	1	1	674	2.9	13	1.3	13
523.xalancbmk	9276	8235	1.3	31	6.2	87	1.3	10	1041	1.8	20	1.3	20
525.x264	154	113	1.6	2	2.1	16	1.6	2	41	1.7	25	1.6	25
526.blender	10310	9625	2.7	16	3.8	1324	2.7	16	685	24.6	48	4.2	48
538.imagick	80	71	1.2	3	5.8	16	1.2	3	9	6.1	7	1.8	7
544.nab	3	3	1	1	1	1	1	1	0	0	0	1	1
557.xz	44	36	1.4	4	5.3	12	1.3	4	8	2.6	6	1.6	6
Httpd	208	158	3.7	78	4.0	66	2.2	6	50	9.4	22	3.9	22
Lighttpd	133	72	1.0	2	2.3	5	1.0	2	61	1.1	5	1.0	5
Nginx	356	293	3.0	73	5.8	33	1.7	6	63	2.4	19	1.8	19
Redis	607	525	2.1	91	2.1	19	1.9	9	82	12.2	46	3.8	46
Edbrowse	32	31	1.8	2	1.8	2	1.8	2	1	1	1	1.7	2
Firefox	1056	912	1.4	7	1.4	7	1.4	7	144	7.7	21	2.3	21

`Redis` (a memory-based data storage system, v.7.0.0), `Edbrowse` (a combination editor, browser, and mail client, v.3.8.9), and `Firefox` (an open source web browser, v.126.0a1). All the experiments were conducted on a server with a Xeon Gold 6130 processor and 256 GB of memory, running a 64-bit Ubuntu 22.04 LTS Server system.

Note that the CFG is the basis of CFI system security, and an EC represents mutually indistinguishable targets of an ICT. So the average EC size can reflect the security of the whole CFI system to some extent, and the largest EC size represents the attack surface size of attackers.

Table 3 shows the overall statistics of ECCut when applied to all the evaluation programs. Note that for SPEC CPU2006 benchmarks, we excluded benchmarks `429.mcf`, `462.libquantum`, and `470.lbm` as they do not have an ICT in their main programs. Similarly, for SPEC CPU2017 benchmarks, we only evaluated the 12 benchmarks that have ICTs in their main programs. The second and third columns from the left of the table (with the same name: *ICTs*) show the total number of ICTs in each program and the number of ICTs that employ *complete* field sensitivity and origin awareness policy. The columns labeled *Avg* and *Lg* show the average and the largest EC sizes respectively. Further, we calculated the average and the largest ECs using only *complete* field sensitivity (*Only field*), only practical origin-awareness (*Only origin*), and both of them (*Field and origin*), respectively. The results indicate that *complete* field sensitivity significantly reduces the size of the largest EC, and the average EC in `445.gobmk` benchmark. And the origin awareness significantly reduces the size of the largest EC in `526.blender` (from 1324 to 16). With the combined effect of *complete* field sensitivity and origin awareness, we obtain the better average and largest ECs. For comparison, on average for all of its benchmarks,

Table 4: The effectiveness of ECCut compared to LLVM CFI and OS-CFI.

Benchmark	ECCut		LLVM CFI				OS-CFI			
	Avg	Lg	Avg		Lg		Avg		Lg	
			Size	Reduce by	Size	Reduce by	Size	Reduce by	Size	Reduce by
400.perlbench	2.8	42	15.3	81.5%	349	87.9%	11.4	75.2%	349	88.0%
401.bzip2	1	1	1.0	0.0%	1	0.0%	1	0.0%	1	0.0%
403.gcc	3.4	43	35.3	90.4%	218	80.3%	3.4	0.0%	218	80.3%
433.milc	1	1	2.0	50.0%	2	50.0%	1	0.0%	1	0.0%
444.namd	1	1	40.0	97.5%	40	97.5%	1	0.0%	1	0.0%
445.gobmk	1.9	14	524.9	99.6%	1637	99.1%	246.3	99.2%	1637	99.1%
447.dealII	1.1	15	1.2	8.5%	37	59.5%	6.7	83.7%	37	59.5%
450.soplex	1.0	5	5.4	81.0%	24	79.2%	1.2	14.2%	11	54.5%
453.porvray	3.2	15	4.2	23.6%	79	81.0%	7.5	57.5%	79	81.0%
456.hmmer	1	1	2.8	64.0%	10	90.0%	1	0.0%	1	0.0%
458.sjeng	1	1	7.0	85.7%	7	85.7%	7	85.7%	7	85.7%
464.h264ref	1.1	2	2.0	46.0%	12	83.3%	1.1	0.0%	2	0.0%
471.omnetpp	1.2	43	2.7	55.6%	244	82.4%	9.2	86.7%	44	2.3%
473.astar	1	1	1.0	0.0%	1	0.0%	1	0.0%	1	0.0%
482.sphinx3	1	1	5.0	80.0%	5	80.0%	1	0.0%	1	0.0%
483.xalancbmk	1.1	29	8.7	87.1%	201	85.6%	3.5	68.0%	29	0.0%
Httpd	3.9	22	14.1	72.3%	83	73.5%	-	-	-	-
Lighttpd	1.0	5	2.8	64.3%	14	64.3%	-	-	-	-
Nginx	1.8	19	7.5	76.0%	78	75.6%	6.6	72.7%	102	81.4%
Redis	3.8	46	15.0	74.7%	259	82.2%	-	-	-	-
Edbrowse	1.7	2	9.2	81.5%	24	91.7%	1.9	12.4%	3	33.3%
Firefox	2.3	21	4.5	48.9%	63	66.7%	-	-	-	-
Average	1.7	15	32.4	94.8%	154	90.3%	17.3	90.2%	140.2	89.3%

the average and largest EC sizes for SPEC CPU2006 are 1.5 and 13.4, while the average and largest EC sizes for SPEC CPU2017 are 2.1 and 21.5, respectively.

For those ICTs that have no origin or whose origins cannot be analyzed by SVF, we use type-based matching to get their CFG, which necessarily biases the average and largest EC. For C programs, most indirect calls have origins and can be analyzed by SVF; for C++ programs, some have more indirect calls using type-based matching (the number of such cases in `483.xalancbmk` is even more than 1/8). This is because SVF cannot analyze C++ virtual calls well, while ECCut avoids this problem by utilizing complete field sensitivity and origin awareness to analyze many original assignments in the C++ programs. It is worth pointing out that the partial type-based matching has an average EC value of less than 1, which means the partial ICTs do not have any targets, as mentioned in other work [31].

To further demonstrate the effectiveness of our system, we compared ECCut with the mainline LLVM CFI (with cfi-icall and cfi-mfcall schemes enabled) [55] and OS-CFI [30], which is the state-of-the-art origin-aware (or context-sensitive) CFI. Table 4 shows the results and it indicates that ECCut can significantly reduce the average and largest sizes of EC. Note that we use the overall average EC and largest EC for comparison. As a result, ECCut can reduce the largest EC size of `445.gobmk` from 1637 to 14, which is a 99.1% reduction; and its average EC size is reduced from 524.9 to 1.9, which is a 99.6% reduction. Overall, compared to LLVM CFI, ECCut reduces the average and the largest EC sizes by 94.8% (from 32.4 to 1.7 on average) and 90.3% (from 154 to 15 on average). While compared to OS-CFI, ECCut reduces the average and the largest EC sizes by 90.2% (from 17.3 to 1.7 on average) and 89.3% (from 140.2 to 15 on average) respectively.

5.1.1 Case Studies. **The EC in 458.sjeng:** As shown in Figure 1, this benchmark only has one indirect call (line 14), and the function pointer is calculated from a static function array with its offset. OS-CFI [30] fails to provide the context for the indirect call due to SVF's [53] failure to field insensitivity. However, for ECCut, it leverages *complete* field sensitivity by adding the offset (`piecet(i)`) as origin at runtime to reduce the largest EC from 7 to 1.

The EC in 400.perlbench: This benchmark has two large function pointer arrays, `PL_check` and `PL_ppaddr`, which contain 348 and 349 function pointers respectively. ECCut employs *complete* field-sensitive policy to reduce the EC of the indirect call instructions corresponding to these two arrays to 1.

The EC in 445.gobmk: There are many pointers of indirect call instructions in this benchmark that can be associated with 14 nested arrays of structs. These arrays contain a total of 1637 function targets that SVF cannot analyze. When ECCut takes a path-based origin analysis of IR, it can analyze the assignment of fourteen global variables; the largest one has 590 targets, which is the largest EC for only origin-aware analysis. However, due to the variable offsets, we have to stake the node after the `GEP` instruction,

```
1  int eight_byte_size_ready (unsigned char const *read_from_)
2  {
3     const uint64_t msg_size = get_uint64 (_tmpbuf);
4     return size_ready (msg_size, read_from_);
5  }
6  int size_ready (uint64_t msg_size, unsigned char const
↪   *read_pos)
7  {
8     ...
9     if (unlikely (!_zero_copy
10    || ((unsigned char *) read_pos + msg_size
11     > (allocator.data () + allocator.size ())))) {...}
12    ...
13  }
14  struct content_t{
15    void *data;
16    size_t size;
17    msg_free_fn *ffn; //function pointer
18    void *hint;
19    zmq::atomic_counter_t refcnt;
20  };
21 int close () {
22    ...
23    u.lmsg.content->ffn (u.lmsg.content->data,
24                         u.lmsg.content->hint);
25  }
```

Figure 6: Sketch of the vulnerable code in libzmq v.4.2.x.

thus losing path information, which makes the largest EC for the field-only analysis as 14.

5.2 Real World Security

We experimented ECCut with three real-world vulnerabilities (CVE-2019-6250, CVE-2020-24349, and CVE-2021-43527) and a constructed COOP attack. Among them, CVE-2019-6250 and the COOP attack can be prevented by ECCut and OS-CFI [30], CVE-2020-24349 can be prevented by ECCut, PathArmor [57], and OS-CFI, while CVE-2021-43527 can only be prevented by ECCut. To assess the effectiveness of ECCut in mitigating these vulnerabilities, we use an existing proof-of-concept (PoC) exploit to manipulate a function pointer and perform control-flow hijacking. Initially, we verify the successful exploitation of these vulnerabilities in their unprotected state to establish a baseline. Subsequently, we repeat the tests with the vulnerabilities being protected by ECCut. By comparing the results before and after applying ECCut protection, we can assess the effectiveness of the proposed approach in mitigating the vulnerabilities and preserving control-flow integrity. Note that this experiment does not prove that ECCut prevents the exploitation of the vulnerability, as such vulnerability can be exploited using other techniques or the set of legal targets. It only shows that using this specific exploit, the vulnerability cannot be exploited.

CVE-2019-6250: This is an integer overflow vulnerability in *libzmq*. As shown in Figure 6, in function `eight_byte_size_ready`, the attacker can provide an uint64_t of his choosing (line 3). In function `size_ready`, a comparison is performed to check if this peer-supplied `msg_size` is within the bounds of the currently allocated block of memory (lines 9-11). When the `msg_size` bytes do not fit in the currently allocated block, this comparison will compute as *false*, causing a very large `msg_size` to overflow the pointer `read_pos`. As it turns out, the space that the attacker is writing to is immediately followed by a struct `content_t` block

```
1  static njs_int_t njs_json_parse_iterator_call(...) {
2    ...
3    if (njs_fast_path(njs_is_fast_array(&state->value) && ...)) {
4      if (njs_is_undefined(&parse->retval)) {
5        njs_set_invalid(value);
6      } else {
7        *value = parse->retval;
8      }
9      break;
10   }
11 }
12 njs_int_t njs_value_property(...) {
13     ...
14     prop = pq.lhq.value;
15     case NJS_PROPERTY_HANDLER:
16       prop = &pq.scratch;
17       ret = prop->value.data.u.prop_handler(...)
18     ...
19 }
```

Figure 7: Sketch of the vulnerable code in njs v.0.4.3.

(lines 14-20). And in the struct `content_t`, `ffn` is a function pointer field (line 17), which is called with two parameters, i.e., `data` (line 15) and `hint` (line 16). This means the attacker can call an arbitrary function/address with two arbitrary parameters.

The function pointer `fnn` is called in function `close` (lines 23-24) to release the message data when the message object is destroyed. ECCut finds that its value is initialized by the user when creating a message object, which is a typical function pointer that can be protected using the origin-awareness policy. Thus, the attack will be detected and prevented by ECCut.

CVE-2020-24349: This vulnerability is a use-after-free (UAF) vulnerability in *njs* through v.0.4.3 (used in Nginx). Figure 7 shows the vulnerable pointer `prop value.data.u.prop_handler` (line 17) that can be overwritten by an attacker to achieve arbitrary code execution. Initially, the function wrongly assumes that the `value` (line 7) pointer is still valid when `njs_is_fast_array (&state->value)` (line 3) is true and the pointer can be used in the `njs_fast_path`. This is not the case when the array object is resized.

The indirect call present in line 17 is safeguarded against control-flow hijacking through the implementation of ECCut. When ECCut is enabled, we discover two distinct origins for `prop` (lines 14 and 16). During the runtime, the environment provides us with valuable contextual information that allows us to identify the only target. Consequently, attempts by an attacker to hijack the control flow are detected.

COOP Attack: We leverage the example code in Figure 8 to illustrate how ECCut can protect against COOP attacks [50]. There is a virtual call (line 39) and a vulnerable function `getID` (lines 20-31). The `getID` function contains a heap-based overflow vulnerability (line 28), which allows the attacker to compromise the vPtr pointer of the returned object, for example, to overwrite the vPtr of `Student` to the vtable of `Teacher`.

ECCut can get two origins of this virtual call, i.e., `origin1` and `origin2`, which locate in lines 24 and 26 respectively. Accordingly, their CFG tuples are `(line 39, origin1, Teacher::score)` and `(line 39, origin2, Student::score)`. When the program executes, it only passes through one origin at a time. The legal target of the program that has passed `origin1` can only be `Teacher::score`,

```
class Person {
public:
  virtual void score() const = 0;
  virtual ~Person() = default;
};

class Student: public Person {
public:
  std::string ID;
  Student(const std::string& id) : studentID(id) {}
  void score() const override {/*get student score*/}
};
class Teacher: public Person {
public:
  std::string ID;
  Teacher(const std::string& id) : ID(id) {}
  void score() const override {/*get all students score*/}
};

Person* getID(const string ID) {
  char *name = (char*)malloc(10);
  Person *P;
  if (isTeacher(ID)) {
     P = new Teacher(ID); //origin1
  } else {
     P = new Student(ID); //origin2
  }
  gets(name);
  ...
  return P;
}

bool isTeacher(const string ID) {...}

int main() {
  ...
  Person* person1 = isTeacher(ID);
  if (isTeacher(P->ID)) {
    person1->score(); //only allow teacher get score
  }
  return 0;
}
```

Figure 8: A program vulnerable to COOP attack.

```
1  static VFYContext *vfy_CreateContext
2   (const SECKEYPublicKey *key, const SECItem *sig,...){
3    ...
4    cx = (VFYContext *)PORT_ZAlloc(sizeof(VFYContext));
5    ...
6    PORT_Memcpy(cx->u.buffer, sig->data, sigLen);
7  }
8  struct VFYContext {
9    ...
10   const SECHashObject *hashobj;
11   ...
12 }
13 const SECHashObject
14 *HASH_GetHashObjectByOidTag(SECOidTag hashOid){
15   HASH_HashType ht = HASH_GetHashTypeByOidTag(hashOid);
16   return (ht == HASH_AlgNULL) ? NULL : &SECHashObjects[ht];
17 }
```

Figure 9: Sketch of the vulnerable code in NSS.

and the legal target of the program that has passed `origin2` can only be `Student::score`. Even if the vptr of the origin is tampered with, ECCut can detect the attack before the virtual call executes.

CVE-2021-43527: This is an NSS cache overflow vulnerability. Figure 9 shows the vulnerability exploitation process. An attacker can utilize the copy function `PORT_Memcpy` in line 6 to manipulate the variable `sig` to overwrite `cx`. The variable `cx` is of struct `VFYContext` type (line 4) and it contains a pointer `hashobj` with type of struct `SECHashObject` array. The `SECHashObject` struct contains a large number of function pointers. Hence the attacker can exploit this vulnerability to execute function calls.

Line 16 is a possible assignment for `hashobj`. Its specific value consists of the array `SECHashObjects` and the variable index `ht`. Thus other CFIs such as LLVM CFI [55], PathArmor [57], and OS-CFI [30] will degenerate into field insensitivity here, while ECCut utilizes the complete field-sensitive policy, which can protect the control flow from tampering very well.

5.3 Performance Evaluation

To demonstrate the performance overhead introduced by our system, we evaluated the performance of ECCut on all the C/C++ benchmarks in SPEC CPU2006 and SPEC CPU2017, as well as the six real-world applications.

The results are shown in Figure 10. On average, ECCut introduces an acceptable performance overhead of 7.2%. The program with the lowest performance consumption is the `444.namd` benchmark, with an overhead of only 0.3%. It has 12 ICTs with average and largest EC sizes of 1. The program that exhibits the highest performance impact is the `526.blender` benchmark, with an overhead of 16.4%. This result is not surprising as `526.blender` has over 10,000 indirect calls, and to protect the program at runtime, we inserted much checking code. For comparison, the average overheads introduced by ECCut on SPEC CPU2006 and SPEC CPU2017 are 6.1% and 7.7% respectively.

Further, we compare the overhead of ECCut with LLVM CFI [55], PathArmor [57], and OS-CFI [30]. When compared with LLVM CFI, the datasets include all SPEC CPU2006 benchmarks, Httpd, Lighttpd, Nginx, Redis, and Edbrowse (We exclude Firefox as it introduces so many false positives for LLVM CFI). Note that for LLVM CFI, we enable all 7 schemes on twelve benchmarks, but only enable some schemes on other benchmarks and applications to avoid false positives. The overheads of LLVM CFI and ECCut are 4.7% and 6.7% respectively. When compared with PathArmor, as PathArmor does not support C++ exceptions, we select all the C programs in both SPEC CPU2006 benchmarks and our real-world applications for comparison. Thus, the datasets include 9 benchmark programs and five real-world applications, i.e., Httpd, Lighttpd, Nginx, Redis, and Edbrowse (we exclude Firefox as it is a C++ program). In the results, the average overheads introduced by PathArmor and ECCut are 6.1% and 7.0% respectively. Although PathArmor only has one SPEC CPU2006 benchmark with an overhead bigger than 10%, it has a large performance consumption on applications, e.g., it introduces 27.3% performance consumption on Lighttpd. When compared with OS-CFI, due to its severe compatibility issues [36], we use the performance data from the OS-CFI paper. Accordingly, the datasets include all SPEC CPU2006 benchmarks and Nginx. In the results, the overheads of ECCut and OS-CFI are 6.8% and 7.1% respectively.

6 DISCUSSION

First, to achieve complete field sensitivity and origin awareness for practical CFI protection, ECCut requires performing static

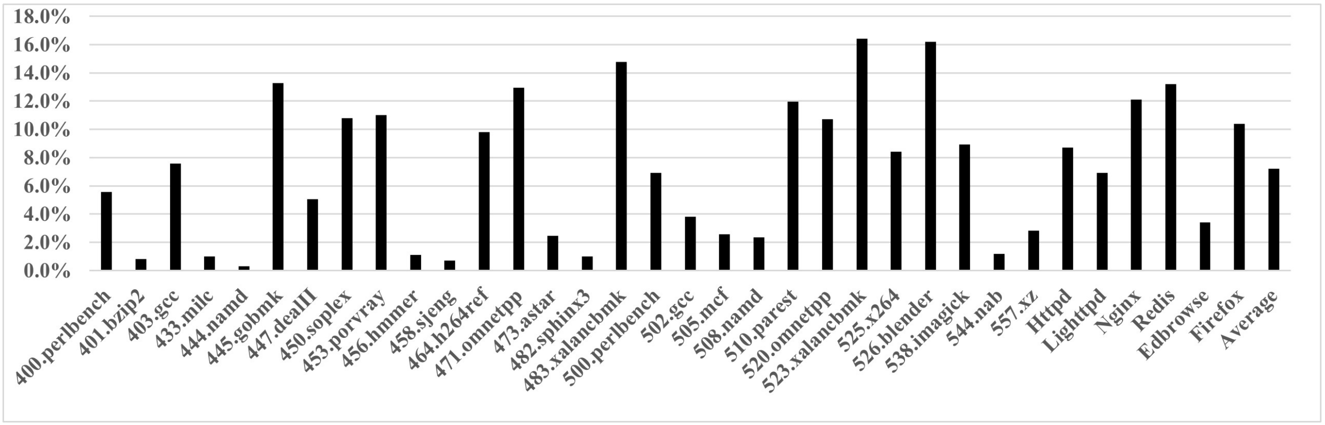

Figure 10: The performance overhead results of ECCut. Average is the average of the performance overhead percentage.

analysis on the IR code compiled from the source code of programs to construct the CFG. This indicates that our approach needs access to the source code. As a result, ECCut does not support pre-compiled executables or third-party drivers. To provide CFI protection for binaries, a number of solutions have been proposed by researchers [32, 37, 38, 58, 63]. However, without source code, it is hard to enforce fine-grained and practical CFI protection for programs.

Second, our system primarily defends against forward-edge ICTs and does not encompass protection for return addresses. Note that various methods have been designed for safeguarding return addresses, such as shadow stack [13], ALSR [52], Intel CET [26], and so on. It is not difficult to integrate such methods into our system if additional protection is required. We leave it as one of our future work.

Third, in this paper we aim to provide CFI protection instead of preventing all kinds of attacks on programs. Particularly, non-control data attacks [8] and data-only attacks [14] are out of our scope. However, we propose a practical CFI with complete field sensitivity and origin awareness for programs, which raises the bar against certain attacks.

Fourth, our prototype of ECCut is built on top of SVF [53], which indicates the requirement for such a state-of-the-art point-to analysis tool. Fortunately, such a tool is available and free for deployment. Further, ECCut uses MPK to store runtime information for later verification, which limits its deployment on other platforms without MPK. Fortunately, ECCut only uses MPK to protect a single memory region at runtime. As alternative solutions in case MPK is unavailable, we can achieve the protection using other in-process isolation techniques [24, 34, 48, 49] to support various platforms.

7 RELATED WORK

A range of CFI systems have been proposed ever since the first CFI work was introduced [2]. Specifically, these systems can be categorized into two main groups based on the way CFGs are generated: dynamic CFI [15, 18, 25, 45, 57, 62] and practical CFI [2, 29, 30, 35, 40]. Dynamic CFI systems typically incur notable performance overhead as they generate dynamic CFGs during program execution and offer the advantage of potentially higher security. On the other hand, practical CFI systems rely primarily on static analysis to construct CFGs. While practical CFI is often more compatible with existing programs, it may not have a fine-grained CFG. Next, we discuss several representative and close-related systems and make a comparison to our approach.

As a dynamic CFI, PathArmor [57] leverages the recent execution history as the context to ensure that the path before a sensitive function call has not been diverted, while ECCut records function pointer assignments and later uses this data to verify whether the jump target is legal before an ICT. To be more specific, there are three differences between the two systems. *First*, PathArmor utilizes Intel LBR to record branch information taken by the process for later verification, which requires changes to the OS kernel as LBR is privileged and only accessible by the kernel. This impedes the deployment of PathArmor. In contrast, ECCut uses MPK to store runtime information for verification, which can be directly accessed in the user space. *Second*, as the transition into and out of the kernel is expensive, PathArmor only protects selected system calls in programs. Thus, the big concern for PathArmor is to protect the remaining part of the system from attacks. In contrast, ECCut provides protection for all wide-spread ICTs, which greatly enhances the security of the whole program. *Third*, as PathArmor does not support C++ exceptions, its current prototypes can only work for C programs. In contrast, ECCut protects all indirect and virtual calls in C and C++ programs.

As another dynamic CFI, μCFl [25] enhances security by enforcing the unique code target (UCT) property, which ensures that an ICT has only one valid target at one time of execution. However, ensuring UCT requires the analysis of a large amount of runtime information, recorded by the PT. Unfortunately, the sheer volume of this information leads to packet drops in PT, posing a significant constraint on the deployment of μCFI in large programs. In contrast, ECCut utilizes the practical origin awareness policy to selectively identify and safeguard crucial origin information, resulting in a reduction in runtime consumption.

As a practical CFI, MLTA [40] involves matching multi-layer types of function pointers and functions to optimize the EC of ICTs. By carefully examining the types and relationships between function pointers and functions, MLTA can effectively reduce the

EC, leading to improved security for the program. However, MLTA does face a limitation when it encounters variable field indexes. In such situations, it tends to degrade into a field-insensitive CFI, meaning it cannot maintain the same level of fine-grained control over control flow as it does in other cases. In contrast, ECCut takes a different approach to address this limitation. It utilizes a complete field-sensitive policy, which can efficiently handle cases involving variable field indexes. By embracing complete field sensitivity, ECCut aims to maintain a high level of precision in control flow protection, even when it faces variable offsets in complex program structures. Further, MLTA is origin-unaware, which becomes over-approximation when analyzing function pointers that are not in structs. In contrast, ECCut can analyze all function pointers, whether they are inside structs or not.

CFI-LB [29] utilizes function call stack information to partition the EC of an ICT. However, since CFI-LB is origin unaware, the call stack information it records may not be fully effective for dividing the EC. It fails when the chain of function pointer passes in a program is too long. In contrast, thanks to the implementation of a practical origin-aware policy, ECCut can recognize specific origins and thus effectively reduce the size of the average EC.

As an origin-aware CFI, OS-CFI [30] also leverages the origin of ICTs to reduce the average and largest EC. However, unlike ECCut, which employs practical origin awareness, the origin of OS-CFI is restricted to explicit assignments to function pointers, resulting in a mere 48.5% coverage of the origin policy. This significantly impacts the security of the system. OS-CFI relies on SVF [53] to generate its CFG, which encounters the issue of field-sensitive degradation into field insensitivity. On the other hand, ECCut utilizes a complete field-sensitive policy to significantly reduce the size of the largest EC (see details in Section 5.1). Additionally, OS-CFI uses MPX's bound table to store metadata, which affects the normal usage of MPX [1]. In contrast, ECCut utilizes the memory access control feature of MPK [27] to protect runtime data without interfering with other MPK functions.

8 CONCLUSION

Practical CFI that employs static analysis to compute the EC of indirect calls have been integrated into compilers. We identify two fundamental problems with existing static analysis for CFI, i.e., incomplete field sensitivity and origin unawareness. To address the problems, we propose a complete field-sensitive and origin-aware CFI system. The new techniques significantly improve the security of CFI by reducing the largest and average EC sizes. By optimizing the instrumenting code and the verification process, our system incurs an acceptable overhead.

ACKNOWLEDGEMENT

We would like to thank all the anonymous reviewers sincerely for their valuable comments. Those comments helped us improve our paper. This work was partially supported by the Foundation for Innovative Research Groups of the National Natural Science Foundation of China (Grant No. 62121001), by the '111 Center' (B16037), by the National Natural Science Foundation of China (Key Program Grant No. 62232013), and by the NSF awards (Grant No. CNS2045478, CNS-2106771, CNS-2154989, and CNS-2247434). Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] [n. d.]. Design of Intel MPX. https://intel-mpx.github.io/design/.
[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA) *(CCS '05)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165
[3] Starr Andersen and Vincent Abella. 2004. Data execution prevention. *Changes to functionality in microsoft windows xp service pack* 2 (2004).
[4] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* 50, 1, Article 16 (apr 2017), 33 pages. https://doi.org/10.1145/3054924
[5] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. 2018. Cfixx: Object type integrity for c++ virtual dispatch. In *Symposium on Network and Distributed System Security (NDSS)*.
[6] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T.R. Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. *24Th USENIX Security Symposium (USENIX Security 15)* (01 2015), 161–176.
[7] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 385–399.
[8] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-control-data attacks are realistic threats.. In *USENIX security symposium*, Vol. 5. 146.
[9] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1409–1426. https://www.usenix.org/conference/usenixsecurity20/presentation/connor
[10] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 952–963. https://doi.org/10.1145/2810103.2813671
[11] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.. In *USENIX security symposium*, Vol. 98. San Antonio, TX, 63–78.
[12] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *2014 IEEE Symposium on Security and Privacy*. 292–307. https://doi.org/10.1109/SP.2014.26
[13] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (Singapore, Republic of Singapore) *(ASIA CCS '15)*. Association for Computing Machinery, New York, NY, USA, 555–566. https://doi.org/10.1145/2714576.2714635
[14] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables.. In *NDSS*.
[15] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. 2014. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC.2014.6881460
[16] Lucas Davi and Ahmad-Reza Sadeghi. 2015. Building Control-Flow Integrity Defenses. In *Building Secure Defenses Against Code-Reuse Attacks*. Springer International Publishing, Cham, 27–54. https://doi.org/10.1007/978-3-319-25546-0_3
[17] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 401–416.
[18] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient Protection of Path-Sensitive Control Security. In *USENIX Security Symposium*. 131–148.
[19] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 901–913. https://doi.org/10.1145/2810103.2813646
[20] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and*

Operating Systems (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 585–598. https://doi.org/10.1145/3037697.3037716

[21] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-Grained Control-Flow Integrity for Kernel Software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. 179–194. https://doi.org/10.1109/EuroSP.2016.24

[22] Eric Grevstad. 2004. CPU-based security: The NX bit. *Earthweb: Hardware* (2004).

[23] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *2014 IEEE Symposium on Security and Privacy*. 575–589. https://doi.org/10.1109/SP.2014.43

[24] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor:{Intra-Process} isolation for {High-Throughput} data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 489–504.

[25] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Chung, William Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1470–1486. https://doi.org/10.1145/3243734.3243797

[26] Intel. 2018. Control-flow Enforcement. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[27] Intel. 2018. Intel® 64 and IA-32 Architectures Software Developer's Manual.

[28] Mohannad Ismail, Jinwoo Yom, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. 2021. VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 1612–1626. https://doi.org/10.1145/3460120.3485376

[29] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. 2019. Adaptive Call-Site Sensitive Control Flow Integrity. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 95–110. https://doi.org/10.1109/EuroSP.2019.00017

[30] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-Sensitive Control Flow Integrity. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 195–211.

[31] Sun Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level. In *Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2021.24386

[32] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level.. In *NDSS*.

[33] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. 75–86. https://doi.org/10.1109/CGO.2004.1281665

[34] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. 2018. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1441–1454.

[35] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. 2018. Fine-CFI: Fine-Grained Control-Flow Integrity for Operating System Kernels. *IEEE Transactions on Information Forensics and Security* 13, 6 (June 2018), 1535–1550. https://doi.org/10.1109/TIFS.2018.2797932

[36] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. 2020. Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1821–1835. https://doi.org/10.1145/3372297.3417867

[37] Yan Lin and Debin Gao. 2021. When Function Signature Recovery Meets Compiler Optimization. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 36–52. https://doi.org/10.1109/SP40001.2021.00006

[38] Ziyi Lin, Jinku Li, Bowen Li, Haoyu Ma, Debin Gao, and Jianfeng Ma. 2023. Type-Squeezer: When Static Recovery of Function Signatures for Binary Executables Meets Dynamic Analysis. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*.

[39] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 529–540. https://doi.org/10.1109/HPCA.2017.18

[40] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1867–1881. https://doi.org/10.1145/3319535.3354244

[41] M.D.Network. [n. d.]. Control flow guard, 2015, [online]. https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx.

[42] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity. In *NDSS*, Vol. 26. 27–30. https://doi.org/10.14722/ndss.2015.23271

[43] Ben Niu and Gang Tan. 2014. Modular Control-Flow Integrity. *ACM SIGPLAN Notices* 49 (06 2014). https://doi.org/10.1145/2594291.2594295

[44] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-In-Time compilation using modular Control-Flow Integrity. *Proceedings of the ACM Conference on Computer and Communications Security* (11 2014), 1317–1328. https://doi.org/10.1145/2660267.2660281

[45] Ben Niu and Gang Tan. 2015. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 914–926. https://doi.org/10.1145/2810103.2813644

[46] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 241–254.

[47] Aravind Prakash, Xunchao Hu, and Heng Yin. 2015. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries.. In *NDSS*.

[48] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for {PKU-based} Memory Isolation Systems. In *31st USENIX Security Symposium (USENIX Security 22)*. 936–952.

[49] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys–Efficient {In-Process} Isolation for {RISC-V} and x86. In *29th USENIX Security Symposium (USENIX Security 20)*. 1677–1694.

[50] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy*. 745–762. https://doi.org/10.1109/SP.2015.51

[51] E. J. Schwartz, T. Avgerinos, and D. Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 317–331. https://doi.org/10.1109/SP.2010.26

[52] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security* (Washington DC, USA) *(CCS '04)*. Association for Computing Machinery, New York, NY, USA, 298–307. https://doi.org/10.1145/1030083.1030124

[53] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235

[54] PaX Team. 2003. PaX address space layout randomization (ASLR). *http://pax. grsecurity. net/docs/aslr. txt* (2003).

[55] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 941–955.

[56] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. {ERIM}: Secure, Efficient In-process Isolation with Protection Keys ({{{{{{MPK}}}}}}). In *28th USENIX Security Symposium (USENIX Security 19)*. 1221–1238.

[57] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. 927–940. https://doi.org/10.1145/2810103.2813673

[58] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–953.

[59] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You shall not (by) pass! practical, secure, and fast PKU-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 266–282.

[60] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *2010 IEEE Symposium on Security and Privacy*. 380–395. https://doi.org/10.1109/SP.2010.30

[61] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. 1–12. https://doi.org/10.1109/DSN.2012.6263958

[62] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *2013 IEEE Symposium on Security and Privacy*. 559–573. https://doi.org/10.1109/SP.2013.44

[63] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 337–352.