Enforcing Unique Code Target Property for Control-Flow Integrity

Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung,
William R. Harris[†*], Taesoo Kim and Wenke Lee

Georgia Institute of Technology [†]*Galois Inc.*

ABSTRACT

The goal of control-flow integrity (CFI) is to stop control-hijacking attacks by ensuring that each indirect control-flow transfer (ICT) jumps to its legitimate target. However, existing implementations of CFI have fallen short of this goal because their approaches are inaccurate and as a result, the set of allowable targets for an ICT instruction is too large, making illegal jumps possible.

In this paper, we propose the Unique Code Target (UCT) property for CFI. Namely, for each invocation of an ICT instruction, there should be one and only one valid target. We develop a prototype called μCFI to enforce this new property. During compilation, μCFI identifies the sensitive instructions that influence ICT and instruments the program to record necessary execution context. At runtime, μCFI monitors the program execution in a different process, and performs points-to analysis by interpreting sensitive instructions using the recorded execution context in a memory safe manner. It checks runtime ICT targets against the analysis results to detect CFI violations. We apply μCFI to SPEC benchmarks and 2 servers (nginx and vsftpd) to evaluate its efficacy of enforcing UCT and its overhead. We also test μCFI against control-hijacking attacks, including 5 real-world exploits, 1 proof of concept COOP attack, and 2 synthesized attacks that bypass existing defenses. The results show that μCFI strictly enforces the UCT property for protected programs, successfully detects all attacks, and introduces less than 10% performance overhead.

CCS CONCEPTS

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

KEYWORDS

Control-flow integrity; Unique code target; Performance; Intel PT

ACM Reference Format:

Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3243734.3243797

∗ This article contains work performed in part while Harris was supported by the Georgia Institute of Technology.

CCS '18, October 15–19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5693-0/18/10…$15.00
https://doi.org/10.1145/3243734.3243797

1 INTRODUCTION

Control-flow integrity (CFI) [1] is a principled solution to detect control-hijacking attacks, in which attackers corrupt control data, like a function pointer, to divert the control flow. It compares the runtime target of each indirect control-flow transfer (ICT) instruction (i.e., indirect `call`/`jmp` or `ret`) against a set of allowed targets, and reports any discrepancy as control-hijacking attacks.

The strength of a CFI system hinges on its model of secure behavior, expressed via its set of allowed targets for ICT instructions. An overly strict model breaks system functionality due to false alarms, while a permissive model can be evaded by attackers, like in [10, 25, 55]. These attacks highlight an inherent mismatch between current CFI models that rely on static analysis and the ideal model: static analysis identifies benign targets for each ICT instruction from *all possible runs*, while the ideal model defines the valid targets for each ICT instruction *for only the currently observed execution*. Recent approaches use runtime information to reduce the number of allowed targets [21, 48, 61]. However, these methods still permit hundreds of targets for some ICT instructions. Consider a code pointer retrieved from an array via a variable index. Without knowing the index value, CFI solutions have to treat all array elements as allowed targets.

In this paper, we propose a necessary feature of a precise CFI — the *Unique Code Target (UCT)* property. This property requires that at each step of a protected execution, a program may only transition to one unique valid target. For an execution without any attack, the allowed target for each invocation of an ICT instruction is the same as the one used in the execution to avoid false alarms. When control data is corrupted to hijack the execution path, the model should detect the deviation and conclude a control-hijacking attack. Similar to existing CFI work, we focus on control-data attacks and consider non-control data attacks [14, 32] out of scope.

The key to achieving the UCT property is collecting the necessary runtime information and using it to augment the points-to analysis on control data. As such information helps constrain the set of allowed targets, we call it *constraining data*. However, it is not trivial to design a CFI system that satisfies the UCT property. Specifically, we have to address the following three challenges, 1) how to accurately identify the constraining data, 2) how to collect this data efficiently, and 3) how to perform the points-to analysis efficiently and accurately.

```c
1  typedef void (*FP)(char *);
2  void A(char *); void B(char *); void C(char *);
3  void D(char *); void E(char *);
4  // uid can be 0, 1, 2
5  void handleReq(int uid, char * input) {
6    FP arr[3] = {&A, &B, &C};
7    FP fpt = &D;
8    FP fun = NULL;
9    char buf[20];
10   if (uid < 0 || uid > 2) return;
11   if (uid == 0) {
12     fun = arr[0];
13   } else { // uid can be 1, 2
14     fun = arr[uid];
15   }
16   strcpy(buf, input); // stack buffer overflow
17   (*fun)(buf);        // fun is corrupted
18 }
```

Figure 1: Code snippet vulnerable to control-flow hijacking attacks. Attackers can exploit the stack buffer overflow at line 16 to corrupt the function pointer `fun`.

We propose a system, μCFI, to address the aforementioned challenges and enforce the UCT property. μCFI performs static data-flow analysis to accurately identify constraining data from the program source code. The analysis starts from code pointers, and recursively identifies variables that are involved in calculating known constraining data. We also develop a novel *arbitrary data collection* technique to record all constraining data at runtime efficiently. Specifically, we encode the constraining data as indirect control-flow transfers, and rely on a hardware feature, Intel Processor Trace (PT) for efficient recording. μCFI runs a monitor in parallel with the program execution to parse recorded constraining data and uses it to argument points-to analysis. To support efficient analysis, we construct partial execution paths to avoid wasting effort on security-unrelated operations. For each invocation of each ICT instruction, the monitor compares the real target against the points-to analysis result, and reports inconsistencies as attacks.

We implement our design as a compiler and an execution monitor. The monitor performs CFI checks in a different process after each ICT instruction. To ensure security, it interacts with the kernel to block the program execution at any security-sensitive system call until all prior CFI checks succeed. This is similar to existing CFI enforcement approaches [15, 21, 23, 61] and aims to prevent attackers from inflicting damage on the system. Our prototype focuses on forward-edge CFI (i.e., protecting `call` and `jmp`), and leaves backward-edge CFI (i.e., protecting `ret`) to existing solutions [17, 33, 54]. We integrate a shadow stack [17] into μCFI to demonstrate its compatibility with backward-edge CFI solutions.

To measure the effectiveness and efficiency of our solution, we use μCFI to protect several benchmarks and real-world programs, including 14 SPEC CPU 2006 benchmarks, nginx web server, and vsftpd FTP server, from 5 real-world exploits, 1 proof of concept COOP attack, and 2 synthesized attacks that bypass existing defenses. μCFI successfully enforces the UCT property at each invocation of each ICT instruction for all tested programs. Attacks are successfully detected and blocked by μCFI, as they trigger CFI violations at runtime. μCFI introduces around 10% overhead to the protected programs. Heavy techniques like memory safety [35, 42–44] or data-flow integrity [12, 58] provide stronger security than the UCT property, but lead to unacceptable performance overhead (116% for SoftBound+CETS, and 104% for data-flow integrity). Thus, our method to enforce the UCT property is a more practical solution.

Table 1: Allowed target sets for various CFI solutions. The functions listed are allowed targets for `fun` at line 17 of Figure 1, when `uid=1`. Our solution μCFI allows the unique target.

line	no CFI	type CFI	static CFI	πCFI	PITTY-PAT	μCFI
0	*	A B C D E	A B C	-	-	-
6				A,B,C	arr[0]:A arr[1]:B arr[2]:C	arr[0]:A arr[1]:B arr[2]:C
7				A,B,C,D	fpt:D	fpt:D
14				A,B,C,D	fun:B,C	fun:B
17				A,B,C,D	fun:B,C	fun:B
#	∞	5	3	4	2	**1**

We make the following contributions in this paper:

- **Unique code target property**. We propose the UCT property as the ultimate goal of control-flow integrity. A CFI system that enforces the UCT property has exactly one allowed target for each invocation of each indirect control-flow transfer.
- **Enforcement of UCT property**. We design and implement an end-to-end system to enforce the UCT property. To achieve this goal, we develop novel solutions to record arbitrary execution information to support complete dynamic program analysis. At the same time, we develop several techniques to enable efficient UCT enforcement.
- **Empirical evaluation**. We evaluate our system on common benchmarks, real-world servers, and attacks. The results show that μCFI successfully enforces the UCT property on all tested programs with around 10% overhead.

The rest of the paper is organized as follows. §2 illustrates the problem we address. We describe our design in §3 and present implementation details in §4. §5 describes an empirical evaluation of our approach and §6 discusses implications of our system. We cover the related work in §7, and conclude in §8. Appendix A formally states and proves the correctness of our approach.

2 PROBLEM

In this section, we demonstrate the weakness of existing CFI implementations with a motivating example and present our idea for enforcing the UCT property.

2.1 Motivating example

Figure 1 shows a vulnerable code snippet that allows attackers to hijack the control-flow. Function `handleReq` contains a stack-based buffer overflow vulnerability at line 16, where the user input (pointed to by `input`) is copied into a fixed-size buffer `buf` without proper boundary checking. Attackers can craft inputs to corrupt local variables on the stack, like the function pointer `fun`. When `fun` is used at line 17 for the indirect function call, attackers can hijack the execution to perform a malicious action.

Control-flow integrity aims to prevent such attacks. The idea is to find the expected target(s) for each indirect control-flow transfer and compare it with the real target at runtime to detect inconsistencies. In this example, CFI will try to validate the value of fun at line 17. Ideally, the check only permits one target for each run, which is function A if uid is 0, function B if uid is 1, or function C if uid is 2. If fun is corrupted to any other value, CFI will detect that the ICT target is inconsistent and terminate the execution to prevent any possible damage.

2.2 Incomplete protection by existing CFIs

Here we demonstrate the weakness of existing CFI solutions in preventing attacks against this code. Table 1 shows the allowed target sets enforced by different CFI solutions at line 17 of Figure 1 when uid is 1. If the vulnerable code is not protected ("no CFI"), attackers can divert the control flow to any executable location (* in the table). The type-based CFI solutions allow all functions whose types match with the callsite [46, 59, 62], and thus permit 5 targets (A, B, C, D, E) for the function pointer fun. Static CFI solutions have to permit all possible targets for all possible benign inputs. Assuming there is an oracle that can enumerate all possible execution paths[1], static CFI will enforce 3 targets: A, B and C. As such oracle is still unavailable, real-world static CFI over-approximates the set of allowed targets. Since it does not consider runtime information, this set is the same across all invocations of the code.

We also consider two dynamic CFI solutions, πCFI and PittyPat, and conclude that neither successfully enforces the ideal CFI policy for this vulnerable code. πCFI starts with an empty set and adds functions at runtime as the function addresses are referenced. The code at line 6 uses the addresses of functions A, B, and C, so πCFI adds them to the allowed targets set. Similarly, it adds function D to the set at line 7 for variable fpt. Therefore, πCFI allows 4 targets at line 17. PittyPat provides the best security guarantee among the existing solutions in Table 1 by utilizing the dynamic execution path to perform points-to analysis. For example, at line 6, PittyPat updates the points-to relationship for each variable, e.g., arr[1] points to B. PittyPat works well when it can infer the points-to relationship from the execution path, but has to make approximations when it cannot. For example, at line 14 fun is either assigned the value of arr[1] or arr[2] depending on the value of uid. Since PittyPat cannot obtain this value from the execution path, it has to allow both targets at line 17. Attackers can choose between calling functions B or C.

2.3 Enforcing UCT with full context

We propose to use the full execution context to perform online points-to analysis on control data to enforce the UCT property. Unlike previous solutions, we collect both the control-flow and the necessary non-control data needed to produce a unique target for each ICT. We refer to such non-control data as *constraining data*, which we define by its property as follows:

Constraining data plays an important role in the calculation of the indirect control-flow transfer target. However, it is neither a control data that directly represents a code address, nor a pointer that will be dereferenced during the code pointer retrieval. The value of a constraining data cannot be inferred from even the accurate execution path until the affected indirect control-flow transfer happens. Once its value is known, the analysis can accurately deduce the unique ICT target for any execution path. Any data satisfying such properties is an instance of constraining data. In the motivating example in Figure 1, the function argument uid is constraining data since it is used to determine the function pointer fun during the array access at line 14, without which any analysis has to overapproximate the access result. There are three challenges to collect constraining data and perform full-context-based points-to analysis in real-world programs:

- **Constraining data identification**. We need to accurately identify constraining data from a tremendous number of program variables. Collecting superfluous data burdens both the collection and analysis.
- **Arbitrary data collection**. No method can efficiently pass arbitrary data from the execution to the analyzer. For example, hardware features like Intel PT only capture change of flow information [21]. Naive solutions with shared files or memory have adverse effects on the cache, leading to significant performance overhead [34].
- **Efficient analysis**. Dynamic analysis with execution context is time-consuming [21, 23], and thus may slow down the protected execution.

In this paper, we propose novel solutions to address these challenges and achieve efficient UCT enforcement. Before presenting our design, we define our threat model and provide a brief introduction of Intel PT.

2.4 Threat Model & Background

Our threat model is the same as related works [1, 59, 68, 69], in which the adversary has full control over the victim's process memory within the constraints of hardware page protection. Therefore, he can perform memory reads or writes at any time during the victim execution. His goal is to exploit memory errors (e.g., buffer overflow) to hijack control. We focus on user space attacks, making kernel exploits out of scope. For simplicity, we do not consider dynamically generated code (e.g., JIT code emission).

Intel Processor Trace. Intel PT is a hardware feature in modern Intel CPUs, which efficiently collects *change of flow information*. PT only collects events that cannot be derived statically. Specifically, *TNT* packets record the branches taken by conditional jumps, *TIP* packets log the targets of indirect control-flow transfers, *FUP* packets log control-flow transfers caused by signals and interrupts, and *PGE* and *PGD* packets indicate the addresses where PT enables and disables tracing, respectively. With a PT trace, we can completely reconstruct the program's runtime execution path. PT records traces directly to physical memory, bypassing the standard processor caches to minimize performance side-effects. Since the trace is collected by hardware and is only configurable from Ring 0, the attacker cannot use it as a channel to directly attack the monitor or evade its data collection.

3 SYSTEM DESIGN

We design μCFI as the first UCT enforcement system. It consists of two components, the static compiler and the dynamic monitor, as

[1]Currently, no efficient implementation of such an oracle exists so it is approximated using fuzzing or symbolic execution.

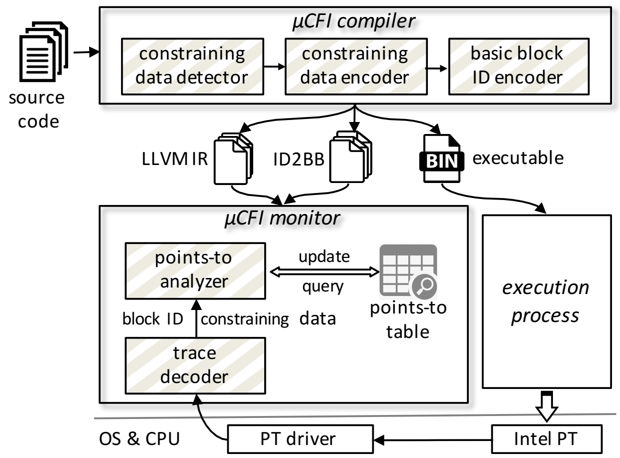

Figure 2: Overview of μCFI. The μCFI compiler takes the program source code as input and identifies constraining data. When the binary is executed, the μCFI monitor performs points-to analysis in an isolated parallel process.

shown in Figure 2. Given the program source code, the compiler performs static analysis to identify all constraining data (§3.1). It instruments the program to encode such data as indirect control-flows for efficient record (§3.2). At the same time, it assigns each basic block a unique ID and records them in the same way (§3.3). μCFI compiler generates three outputs: the instrumented binary, the LLVM IR for points-to analysis, and the mapping from ID to LLVM IR basic blocks. μCFI monitor works in parallel with the protected program and oversees the program's indirect control-flow transfers. It parses PT trace from the kernel driver to decode basic block ID (§3.3) and constraining data. With basic block ID, the monitor identifies executed basic blocks, and performs points-to analysis for every instruction. With the help of constraining data, the analysis generates the unique target for each ICT instruction (§3.4). After each indirect control-flow transfer, the monitor compares the real target used by the program (recorded in PT trace) with the allowed target from the points-to analysis (§3.5). If they do not match, the monitor informs the kernel to terminate the execution to prevent damage to the system.

3.1 Constraining data identification

As we define in §2.3, constraining data is involved in calculation of code pointers, but their values cannot be directly inferred from the execution path. Based on this property, we define a static analysis procedure in algorithm 1 to find all constraining data in two phases: first, we collect instructions related to ICT target calculation; second, we check operands of these instructions to find non-constant values – such values are constraining data.

In the first phase, we collect all instructions that directly or indirectly involve function pointer calculation. Direct involvement means the instruction reads or writes a function pointer. Indirect involvement means that the instruction prepares the data for direct involvement, like retrieving the pointer of the function pointer. We use a recursive approach to identify all such instructions. From line 1 to line 6, our algorithm checks all data types used in the program to locate *sensitive types*. A sensitive type is either a function pointer

Algorithm 1: Constraining data identification.

```
   Input: G - program to be protected
   Output: constraining data set
   TS ← ∅                                    // sensitive type set
1  repeat
2     for typ ∈ Types(G):
3        if typ is function-pointer type: TS ← TS ∪ {typ}
4        elif typ is composite type:
5           for sTyp ∈ allTypes(typ):
6              if sTyp ∈ TS: TS ← TS ∪ {typ}
   until no new sensitive type is found
   IS ← ∅                                    // sensitive instruction set
7  repeat
8     for instr ∈ Instructions(G):
9        if instr has type ∈ TS: IS ← IS ∪ {instr}
10       elif isLoadInst(instr) or isStoreInst(instr):
11          if value ∈ IS: IS ← IS ∪ {pointer}
12       elif isCallInst(instr):
13          if form-arg ∈ IS: IS ← IS ∪ {act-arg}
14          if act-arg ∈ IS: IS ← IS ∪ {form-arg}
15       ... ...
   until no new sensitive instruction is found
   CS ← ∅                                    // constraining data set
16 for instr ∈ IS:
17    for oprnd ∈ Operands(instr):
18       if oprnd ∉ IS and ¬isConstant(oprnd):
19          CS ← CS ∪ {oprnd}
```

type (line 3), or a composite type containing some members whose type is known to be sensitive (line 4-6). We repeat the search until no new sensitive type can be found. Then from line 7 to line 15, the algorithm checks all instructions to identify the *sensitive instructions* that either produce a value with a sensitive type (line 9), or involve the calculation of an already-identified sensitive instruction. For example, lines 10 and 11 check whether the value read from or written to the memory has been labeled as sensitive. If so, it will add the pointer to the sensitive instruction set. We redact the code to process other type instructions at line 15 for brevity. In the second phase, the algorithm checks the operands of each sensitive instruction (line 16-19). Any operand that is neither in the sensitive instruction set nor a constant value (line 18) is treated as constraining data and is added to the appropriate set (line 19). The algorithm returns the set of identified constraining data.

Table 2 shows the result of constraining data identification on the code in Figure 1. Our analysis finds two sensitive types (i.e., function pointer type `void (char*)*` and function pointer array type `[3 x void (char*)*]`), six sensitive instructions and one constraining data `uid`. As `uid` is neither a sensitive value, nor a constant in the sensitive instruction: `fun = arr[uid]`, it is constraining data.

3.2 Arbitrary data collection

We design a novel method to efficiently pass any information from the execution to the monitor. Our method uses software instrumentation to encode any data into control data, and then utilizes Intel PT to generate the encoded trace efficiently. As we discuss in §2.2,

Table 2: Identifying constraining data from code in Figure 1.

sensitive type	`void (char*)*` `[3 x void (char*)*]`

sensitive instruction	`FP arr[3] = {&A, &B, &C};`
	`FP fpt = &D;`
	`FP fun = NULL;`
	`fun = arr[0];`
	`fun = arr[uid];`
	`(*fun)(buf);`

constraining data	`uid`

typical PT tracing without our instrumentation cannot achieve the UCT property due to the lack of non-control information.

μCFI implements two functions, `write_data` in the protected program to encode arbitrary data, and `read_data` in the monitor to restore data for analysis, as shown in Figure 3. To log an arbitrary value `av`, μCFI instruments the program to call `write_data` with `av` as the argument. `write_data` divides `av` into several chunks, each containing `N` bits (lines 13 and 15). `write_data` adds a constant base pointer `BASE_ADDR` to each chunk to get a new code pointer (line 13) and uses the new pointer to launch an indirect function call (line 14). PT will record the new code pointer value into the trace. The base code pointer points to a special executable area (function `allRet`) filled with 2^N one-byte return instructions (`0xc3` for Intel CPU). Therefore, the indirect call immediately returns and `write_data` will process the next chunk (line 12). The μCFI monitor recovers the encoded value by calling function `read_data`. `read_data` reads PT packets from the trace, and restores the chunk value by subtracting the base code pointer value `BASE_ADDR` from the PT packet (line 21). By accumulating chunk values, `read_data` gets the encoded data (line 22). Then the monitor can perform online points-to analysis with the decoded data. μCFI imposes a small footprint in the data cache by sharing only a minimal set of constraining data (see §5 for performance and code overhead evaluation).

Security consequence. Readers may worry about that adding `write_data` to the protected program introduces another indirect function call and thus enlarges the attack surface. We clarify that such instrumentation does not change the program security, as attackers cannot utilize this indirect function call to build any exploit. In our implementation of `write_data`, the mask operation on `av` at line 13 guarantees that the offset from `BASE_ADDR` is within the boundary of the `allRet` function. The `new_ptr` variable is stored in a register, which is out of the attacker's control. Even if attackers corrupt the value of `av`, the execution will merely call a different `ret` instruction and return to the same location as a benign call.

Figure 4a shows the instrumented code from Figure 1. Since `uid` is constraining data in the instruction at line 14, the compiler inserts `write_data(uid)` at line 13 to record it. Consider an example that passes `0xABBBCCCDDDEEEFFF` from the execution to the analysis. Suppose that the special executable region `allRet` starts from address `0x1000`, and that μCFI uses 12-bit value as a chunk. `write_data` will trigger 6 indirect function calls, each encoding 12 bits (the last one encodes 4 bits). Then PT trace will contain the following packets: {`0x1FFF`, `0x1EEE`, `0x1DDD`, `0x1CCC`, `0x1BBB`, `0x100A`}.

```
1  #define CHUNK_SIZE    12
2  #define CHUNK_MASK    ((1 << CHUNK_SIZE) - 1)
3  #define CHUNK_COUNT   (((64 - 1) / CHUNK_SIZE) + 1)
4  #define BASE_ADDR     ((unsigned long)allRet)
5  typedef void (*FP)();
6  typedef unsigned long u64;
7  void    allRet();             // filled with 2^N returns
8  u64     getPTPacket();        // return next PT packet
9
10 void write_data(u64 av) {     // write data to PT trace
11   int count = 0;
12   while (count++ < CHUNK_COUNT) {
13     FP new_ptr = (FP)(BASE_ADDR + (av & CHUNK_MASK));
14     (*new_ptr)();
15     av >>= CHUNK_SIZE;
16 }}
17
18 u64 read_data() {             // recover data from PT trace
19   int count = 0;  u64 av = 0;
20   while (count < CHUNK_COUNT) {
21     u64 chunk = getPTPacket() - BASE_ADDR;
22     av += (chunk << (CHUNK_SIZE * count++));
23   }
24   return av;
25 }
```

Figure 3: Functions for arbitrary data collection. `write_data` encodes the input `X` into code data and dumps it into PT trace, while `read_data` restores the encoded value from the PT trace.

3.3 Efficient control-flow construction

μCFI monitor constructs the dynamic control-flow at LLVM IR-level from the PT trace so that the analyzer can perform points-to analysis for every executed IR instruction. However, constructing IR-level paths incurs the following two challenges. The first one is the time-consuming parsing of PT traces. Previous work [21, 23] demonstrates that reconstructing the complete execution path from the highly-compressed PT trace is computation-intensive. Griffin [23] has to use six extra kernel threads to achieve acceptable performance. The second challenge is the inconsistency between the binary-level path and the IR-level path. Due to complicated compiler optimizations (e.g., instruction scheduling and loop-invariant code motion), binary-level control-flow significantly differs from LLVM IR-level flow. Disabling all optimizations helps mitigate the inconsistency, but cannot completely solve this problem, and more importantly, hurts performance.

Our accurate and efficient IR-level control-flow reconstruction is inspired by three observations. First, regardless of optimizations, the compilation process always retains the program's high-level functionality, including the order of side-effecting operations (e.g., memory access and function call). As long as one IR-level control-flow has the same order of side-effecting operations as the binary-level control-flow, the analysis on it will be functionally equivalent to the analysis on an ideal IR-level flow (which exactly matches the binary-level flow). Second, instructions inside the same basic block get executed in a fixed order – from the first to the last. Therefore, we just need the control-flow on IR basic block level. Third, our points-to analysis does not require a complete control-flow. It merely requires the execution order of sensitive instructions (defined in §3.1). Such instructions usually account for a small portion of the whole program. Therefore, a partial control-flow covering all sensitive instructions should suffice.

```c
void handleReq(int uid,char *input) {
  write_data(ID1);          // BBID ID1
  FP arr[3] = {&A,&B,&C}; // s-instr
  FP fpt = &D;              // s-instr
  FP fun = NULL;            // s-instr
  char buf[20];
  if (uid < 0 || uid > 2) return;
  if (uid == 0) {
    write_data(ID2);        // BBID ID2
    fun = arr[0];           // s-instr
  } else  {
    write_data(ID3);        // BBID ID3
    write_data(uid);        // c-data
    fun = arr[uid];         // s-instr
  }
  write_data(ID4);          // BBID ID4
  strcpy(buf, input);
  (*fun)(buf);              // s-instr
}
```

=== TRACE ===>>

```c
1  // PTS: global points-to table, initialized with NULL
2  while (true) {
3    int BBID = read_data();
4    switch(BBID) {
5      case ID1: PTS[arr[0]] = A; PTS[arr[1]] = B;
6                PTS[arr[2]] = C; PTS[fpt] = D;
7                break;
8
9      case ID2: PTS[fun] = PTS[arr[0]];
10               break;
11
12     case ID3: int uid  = read_data();
13               PTS[fun] = PTS[arr[uid]];
14               break;
15
16     case ID4: int real_target = getPTPacket();
17               if (real_target != PTS[fun]) abort();
18               else continue;
19 }}
```

(a) Instrumented program in Figure 1 **(b) Monitor internal**

Figure 4: Instrumentation and execution monitoring of the program in Figure 1. μCFI compiler adds extra instructions (shaded) to record constraining data (c-data) and basic block IDs into the PT trace. μCFI monitor extracts BBIDs and constraining data from the trace, performs points-to analysis for sensitive instructions (s-instr), and validates ICT targets.

Our method for constructing IR-level control-flow is as follows. μCFI compiler identifies LLVM IR basic blocks that have at least one sensitive instruction, and assigns each a unique ID (referred to as a *BBID*). During program instrumentation, the compiler inserts `write_data` calls at block entries, taking BBID as the argument. Once the basic block gets executed, its BBID will be recorded into the PT trace. μCFI monitor extracts BBID from the PT trace, maps it to the corresponding LLVM IR-level basic block, and performs the points-to analysis for each contained sensitive instruction. As function `write_data` has side effect, the compiled binary has the same order of original side-effecting operations as the LLVM IR between any two consecutive BBID packets. μCFI compiler outputs a map (ID2BB in Figure 2) for μCFI monitor to translate a BBID in the PT trace into the corresponding IR basic block. With this novel method, our PT parser can simply ignore all TNT packets and focuses on the TIP packets that encode BBIDs (detailed in §5.3).

Figure 4a shows the instrumentation of the code in Figure 1, represented in `C` language for clarity. The newly added statements are highlighted, including 4 calls to `write_data` to dump BBIDs and another call to record the constraining data `uid`. The sensitive instructions related to ICT target calculation are labeled with "s-instr", which the monitor will use for analysis.

3.4 Online points-to analysis

We perform the points-to analysis with the full execution context to enforce the UCT property. μCFI monitor first extracts a BBID from the trace through the `read_data` function. Based on the mapping from BBID to LLVM IR-level basic block, the monitor locates the current basic block and performs the points-to analysis for each contained sensitive instruction. Our analysis maintains a global points-to table with one entry for each program variable. To process one sensitive instruction, the analysis simply queries the points-to table to retrieve the target of the source operand and uses it to update the entry of the destination operand according to the semantics of the instruction. Figure 4b illustrates the monitoring code for the example in Figure 1 in `C` format. Before the analysis loop, the code initializes the global points-to table `PTS` with `NULL` targets. Inside the loop, it retrieves a BBID (line 3) and performs the analysis (line 5-18). For example, if the BBID is `ID2`, the code will find the target of `arr[0]` from the table `PTS` and use it to update the entry for `fun`. When the BBID is `ID3`, the code retrieves the concrete value of the constraining data `uid` and then updates the entry for `fun` accordingly. Based on the instrumentation in Figure 4a, `uid` is logged immediately after `ID3` so the code at line 12 of Figure 4b will get the correct value.

The μCFI monitor performs its analysis in a memory-safe manner so that the results are correct in spite of any memory corruptions in the execution process due to spatial memory errors (e.g., buffer overflow) or temporal memory errors (e.g., use-after-free). Unlike the execution process, our analysis represents each object as a `Node` object and each pointer as a (`Node`,`offset`) pair. The first element of the pair indicates the object pointed by the pointer, and the second element is the distance from the object start to the pointed location. We use this representation to create shadow objects in the monitor and maintain their points-to information. The (`Node`,`offset`) representation allows us to implement a relaxed model of memory safety under the "infinite spacing" definition [31]. This mode has two properties: 1) there is no access to undefined memory; 2) memory regions are allocated infinitely far apart. The first property rules out memory errors like use-after-free and uninitialized memory access, as each pointer (`A`,`x`) has an attribute to indicate its status (e.g., `allocated`, `freed`). The second property rules out buffer overflows. Since data objects under our analysis are allocated from an infinite object space, writing to a `Node A` with an out-of-bound offset `x` will have no effect to other nodes, even though in the real execution the address corresponding to the pointer (`A`,`x`) could be the same address as some other pointer (`B`,`y`).

3.5 ICT target validation

When the analysis reaches an ICT instruction, the μCFI monitor queries the points-to table to get the unique allowed target calculated by the points-to analysis. It then retrieves the TIP packet from PT to find the real target used in the execution. If they mismatch, the monitor concludes that memory corruption occurred and terminates the protected program. In the strongest security policy, the monitor would block the execution after each ICT instruction to validate the target. However, frequent suspensions introduce an unacceptably high performance overhead. Therefore, μCFI performs CFI checks in parallel with the execution and only suspends the execution at critical system calls. It waits for the validation logic to finish checking all indirect control-flow transfers, and then resumes the execution if no CFI violations are detected. We consider the following system calls to be security-sensitive, similar to many other security systems [15, 21, 50, 61]: `mmap`, `mremap`, `remap_file_pages`, `mprotect`, `execve`, `execveat`, `sendmsg`, `sendmmsg`, `sendto`, and `write`.

μCFI focuses on determining the unique target for each invocation of each ICT instruction. Attackers may corrupt the constraining data before it is recorded by PT, as we do not enforce data integrity. In this case, our analysis may derive the wrong ICT target and thus miss an attack. However, malicious corruption of constraining data falls into the category of non-control data attacks [14, 32] and is thus out of the scope of this work.

4 IMPLEMENTATION

We implement a prototype of μCFI on x86_64 system with 6010 source lines of code for the program compiler and the execution monitor. We choose x86_64 system as it is widely used and long-term supported. However, our idea of enforcing the UCT property is general and applicable to similar systems, like x86.

Our compiler is built on top of LLVM 3.6, with a LLVM pass for IR-level instrumentation and a set of updates to the X86 backend for assembly-level instrumentation. The LLVM pass performs the constraining data identification and encoding, and the BBID encoding, as we discuss in Section §3. The updated X86 backend helps achieve trace size reduction and shadow stack protection, which we will discuss in this section. We implement the monitor as one root user process, which makes it suitable for protecting non-root processes. However, this is only a limitation of the current implementation and not the overall design, which can have a kernel monitor or additional protection mechanisms (e.g., SELinux). It uses two threads, one for PT trace parsing and another for points-to analysis and CFI validation. We use a modified version of the PT driver from Griffin [24] for trace management, in which we write the trace into per-thread pseudo-files and set appropriate permissions for our user-space μCFI monitor to read it. Next we present several implementation details of the μCFI system, including efforts for trace reduction, integration with shadow stack, and a practical type analyzer for the points-to analysis.

Trace reduction. PT allows users to specify the traced code range of a particular program, and only generates packets when the program executes inside the traced range. To utilize this feature to minimize the trace size, we perform program instrumentation to redirect all necessary packets into one dedicated code range. Specifically, we implement a function `iCall` to realize indirect function calls, and a function `oneRet` to achieve function returns. μCFI compiler replaces each indirect function call in the program with a direct call to `iCall`, with the original function pointer as the first argument. `iCall` contains one indirect jump instruction that goes to the address specified in the argument. μCFI replaces each `ret` instruction with a direct jump to the `oneRet` function, which contains one `ret` instruction to perform function return. During the execution, we configure the trace range to cover only `oneRet` and `iCall`, which is 48 bytes (8 for instruction and 40 for padding). In this way, we avoid all TNT packets that usually dominate PT traces. We show that our trace reduction significantly reduces the size and helps mitigate the performance overhead of parsing in Section §5.3.

Integration with shadow stack. To demonstrate the compatibility of μCFI with existing backward-edge CFI solutions, we implement a parallel shadow stack in μCFI compiler [17]. Parallel shadow stack saves return addresses in a different stack, but with a fixed (optionally randomized) offset from the original location. Upon function return, it compares the two versions of the return address to detect attacks, or overwrites the one on the real stack with the shadowed copy to disable attacks. Our implementation of parallel shadow stack contains a patch to LLVM X86 backend, and an ELF constructor function. The former inserts two assembly instructions into each function, one at the function entry for saving the return address to the shadow stack, and another before the `ret` instruction for bringing the shadow copy back. ELF constructor functions are invoked by the binary loader before giving control to the program code, which we use to set up the shadow stack and create guard pages between two stacks. We evaluate it with μCFI in Section §5.4.

Lazy type analysis. Type flattening is the technique of representing a composite type as basic types [21, 30, 38]. Our points-to analysis requires type flattening to represent an object as a set of (`Node`,`offset`) pairs, each representing a basic-type element. The common way to flatten a type is to recursively replace its element types with their definitions until all elements have basic types. However, this method requires accurate type information during the object allocation, which may not be available in highly optimized LLVM IR. We propose *lazy flattening* to expand an object when it is accessed at runtime. During the object allocation, we represent it as an empty set. When it is accessed through a pointer (`Node`,`x`), we know that at offset `x` the object has an element with a particular type, and will update the object representation accordingly. Therefore, lazy flattening tolerates the type missing problem. However, it may slow down the analysis due to the dynamic type analysis. μCFI uses a hybrid solution: we flatten an object as much as possible based on the type information during its allocation, and use lazy type flattening to address the type missing problem.

5 EVALUATION

We perform empirical evaluations to answer the following questions regarding μCFI's security and performance:

- **Q1.** can μCFI enforce the unique code target property?
- **Q2.** can μCFI prevent real-world advanced attacks?
- **Q3.** what is the cost of using μCFI for protection?
- **Q4.** can μCFI work well with backward CFI solutions?

Benchmarks. We use μCFI to protect 14 SPEC CPU2006 benchmarks and 2 real-world applications, the nginx web server and the vsftpd file server, and measure the allowed target number among all executed ICT instructions (**Q1**). We also measure the overhead of μCFI on these benchmarks and applications, including execution time, memory usage, and code size (**Q3**). We collect 5 publicly available control-hijacking attacks against 4 vulnerable applications, 1

Table 3: Evaluation result of μCFI on SPEC CPU2006, nginx and vsftpd. We measure the number of allowed targets for all ICT instructions in `Allowed Target #`. We report the overhead introduced by μCFI regarding `time`, `memory`, and `code size`; `instru` only covers code instrumentation; `+monitor` also considers the μCFI monitoring; `+stack` integrates the parallel shadow stack. Other columns show the number of PT packets for BBID, return and constraining data. – means no function pointer. Gray rows indicate C++ benchmarks. We calculate an extra average, excluding benchmarks without any ICT instruction.

	kilo-sLOC	Allowed Target #		Time Overhead (%)				Mem	vCode	PT Packet #		
		μCFI	w/o c-data	instru	+monitor	+stack	PittyPat	(%)	(%)	BBID	Ret	c-data
perlbench	128.2	1	**1~1.8e19**	13.79	49.67	47.63	47.3	4.95	32.13	130M	18M	17M
bzip2	5.7	1	1	0.70	1.06	1.82	17.7	0.25	5.12	439K	211K	0
mcf	1.6	–	–	0.22	-0.82	0.73	4.3	0.04	0.29	0	0	0
milc	9.6	1	1	0.56	0.52	1.25	1.8	0.56	0.14	498K	9	0
namd	3.9	1	1	0.07	0.24	-0.01	28.8	0.13	0.60	25K	12K	14K
gobmk	157.7	1	**1~1.8e19**	4.96	8.55	18.63	4.0	0.00	0.54	504M	19M	34M
soplex	28.3	1	1	0.11	3.95	3.37	27.5	0.46	13.77	10M	5M	501K
hmmer	20.7	1	1	1.25	1.29	0.78	20.2	1.70	0.24	9	2	0
sjeng	10.5	1	**7**	4.07	10.56	18.36	6.7	0.04	0.18	39M	8M	30M
libquantum	2.6	–	–	0.33	0.00	-1.64	14.1	1.73	0.09	0	0	0
h264ref	36.1	1	**1~1200**	6.53	24.32	35.14	11.8	0.76	2.68	21M	553K	460K
lbm	0.9	–	–	0.05	0.00	-0.02	0.7	0.01	0.28	0	0	0
astar	4.3	1	1	4.00	10.09	13.97	22.5	0.57	0.32	1G	7M	0
sphinx	13.1	1	1	1.09	0.86	0.03	16.0	0.19	0.17	8K	1K	0
Average		all above		2.65	**7.88**	**9.95**	17.95	0.81	4.04			
		w/o mcf, libq & lbm		3.38	**10.10**	**12.74**	18.57	0.87	5.08			
GeoMean				0.69	1.36	1.58	10.48	0.24	0.50			
Variance				0.15	1.92	2.32	1.72	0.02	0.79			
nginx (/req)	103.4	1	**1~6.2e6**	0.46	4.05	4.05	11. 9	5.61	20.25	11K	2K	7K
vsftpd (/req)	16.5	1	**1~13**	1.13	0.75	0.83	n/a	4.77	17.29	10K	10K	603

proof of concept COOP attack, and synthesize 2 advanced attacks that bypass existing CFI implementations. Then we check whether μCFI can prevent such attacks (**Q2**). We integrate one representative shadow stack, the parallel shadow stack [17], to check the compatibility of μCFI with backward-edge CFI solutions (**Q4**). We evaluate the correctness and overhead of the combined protection.

Due to the data loss problem of Intel PT, we cannot perform end-to-end evaluation for some SPEC benchmarks. We check this issue in §6.3 and discuss the missed benchmarks in §6.4.

Setup. We perform our evaluation on a 64-bit Ubuntu 16.04 system, equipped with an 8-core Intel i7-7740X CPU (4.30GHz frequency) and 32 GB RAM. We compile each program in two steps. First, we use wllvm [52] to generate the baseline binary and the LLVM IR representation of the whole program. Second, we use μCFI compiler to instrument the IR and generate the protected executable. Both compilations take default optimization levels and options, like, `O2` for SPEC and `O1` for nginx and vsftpd. We use the provided train data sets to evaluate SPEC benchmarks. For nginx and vsftpd, we set up the server on our evaluation environment, and request files with different size from another machine in the same local network. We request each file for 1000 times to avoid accidental deviations. To measure the overhead, we launch the protected execution together with the monitor, and count the time till all processes exit, including the protected execution, the monitor and their child processes.

Result summary. Table 3 and Table 4 summarize our evaluation results. μCFI successfully enforces the UCT property for tested programs as it only allows one valid target for all indirect control-flow transfers (**Q1**). μCFI introduces 7.88% runtime overhead for evaluated SPEC benchmarks on average, 4.05% runtime overhead for nginx and less than 1% overhead for vsftpd (**Q2**). This means that μCFI can efficiently protect these programs with a strong security guarantee. All attacks, including the real-world attacks, the COOP proof of concept attack and the synthesized attacks, are blocked by μCFI at runtime (**Q3**). Programs compiled with μCFI and the shadow stack work well. The combined protection introduces extra 2.07% overhead to SPEC benchmarks, and negligible extra overhead to nginx and vsftpd (**Q4**).

5.1 Enforcing UCT property

μCFI successfully enforces the unique code target property for evaluated SPEC CPU2006 benchmarks, nginx and vsftpd, as shown in the `uCFI` column (under `Allowed target #`) of Table 3, in which all ICT instructions have one and only one allowed target. SPEC benchmarks mcf, libquantum and lbm do not have ICT instructions in their LLVM IR, so we skip their numbers in the column.

5.1.1 Necessity of constraining data. To understand the advantage of μCFI, we emulate the analysis without constraining data (like in PittyPat [21]) to estimate the number of allowed targets for ICT instructions. Specifically, we associate each sensitive data with a `counter` variable to represent the number of its possible sources. This value is initialized as 1, and gets propagated among sensitive instructions. If one instruction uses constraining data to derive the destination from the source, we multiply the source `counter` by the maximum value of the constraining data and assign the result to the destination `counter`. The multiplication represents the inevitable overestimation the analysis has to make to conservatively permit all possible targets. We infer the maximum value of the constraining data from static analysis if possible (e.g., static array size); otherwise

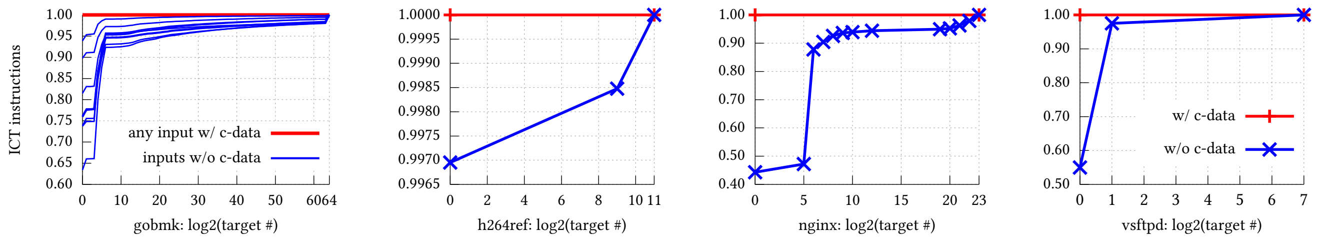

Figure 5: Cumulative distribution of allowed target number in h264ref, nginx, vsftpd and gobmk. X-axis shows the binary logarithm of allowed target numbers. Executions with constraining data always have unique targets, and therefore produce the horizontal line on the top.

```
1  struct { void (*mapping)(...); } SyntaxElement;
2  struct { SyntaxElement MB_SyntaxElements[1200]; } *img;
3  int writeMBLayer (int rdopt) {
4    /* index is the constraining data to determine currSE */
5    SyntaxElement *currSE = &img->MB_SyntaxElements[index];
6    writeSyntaxElement_UVLC(currSE, ...);
7  }
8  int writeSyntaxElement_UVLC(SyntaxElement *se, ...) {
9    se->mapping(se->value1, ...);
10 }
```

Figure 6: Simplified h264ref code snippet that retrieves a function pointer from a large structure array.

we use its concrete value at runtime as an under-approximation. Finally, the `counter` value of the code pointer is the number of allowed targets for the ICT instruction. Note that our goal here is not to get an accurate number of unique allowed targets. Instead, we mainly use the `counter` value to estimate the attacker's flexibility on building control-hijacking attacks.

Column (`w/o c-data`) shows our estimation results, in which 4 SPEC benchmarks and both real world applications will permit significantly more targets if the analysis does not use constraining data. sjeng always permits 7 targets for its only ICT instruction, while another 5 applications allow targets varying from small counts (e.g., 2) to the maximum integer (i.e., `ULONG_MAX` on Linux). We draw the distribution of allowed target number for gobmk, h264ref, nginx and vsftpd in Figure 5, where the X-axis shows the binary logarithm of the allowed target number. The distribution of allowed target numbers for gobmk varies from input to input, in which ICTs with more than one target range from 5% to 35%. Most ICTs have less than 64 (2^6) targets, but some may permit 2^{64} targets. For h264ref, 99.7% of ICTs have only one target. Other ICTs have either 400 targets (0.15%) or 1200 targets (0.15%). Requesting a 1KB file from nginx can trigger ICTs with 2^{23} targets, while ICTs with one target only account for 45%. The allowed target numbers for vsftpd is simpler, like one target (55%) or two (40%), and a few with 128 targets. Our estimation shows that without constraining data, attackers have substantial flexibility to divert the control flow, even if the victim is protected with known CFI solutions. For soplex and namd, μCFI monitor detects operations on constraining data (the last column on Table 3), but no ICT instructions use the operation results. Therefore, the analysis without constraining data also enforces one target for all invoked ICT instructions. However, we find execution paths in these programs that really use constraining data for ICT instructions, where the analysis has to use constraining data to achieve the UCT property.

```
1  typedef int (*autohelper_fn_ptr)(...);
2  struct pattern { autohelper_fn_ptr autohelper; };
3  struct matched_pattern_data { struct pattern *pattern; };
4  struct matched_patterns_list_data
5    { struct matched_pattern_data *pattern_list; };
6  int get_next_move_from_list(
7      struct matched_patterns_list_data *list, ...) {
8    for (...) {
9      struct matched_pattern_data tmp;
10     tmp = list->pattern_list[index1];
11     list->pattern_list[index2] = tmp;
12
13     tmp = list->pattern_list[index3];
14     check_pattern_hard(..., tmp.pattern, ...);
15 }}
16 int check_pattern_hard(..., struct pattern *pattern, ...) {
17   pattern->autohelper(...);
18 }
```

Figure 7: Simplified gobmk code snippet that cascades memory access with constraining data.

Case study 1: reading code pointer from a huge table. h264ref permits up to 1200 targets if the constraining data is not available. We inspect its execution trace and figure out that the large number is caused by reading a code pointer from a huge table with a variable index, as shown in Figure 6. Structure `ImageParameters` contains an array of 1200 `SyntaxElement` instances, while structure `SyntaxElement` has a function pointer `mapping`. h264ref gets the structure pointer `currSE` from that array (line 5), and uses the function pointer in the pointed structure for an indirect function call (line 9). The `index` used to retrieve the structure pointer is constraining data, without which the analysis has to conservatively take all 1200 elements as the potentially retrieved pointer. When `currSE` is dereferenced to get the function pointer, there are up to 1200 candidate locations, leading to the large allowed target number.

Case study 2: cascading access. gobmk has the largest allowed target number (the maximum 8-byte value), due to the cascading of constraining data. Specifically, one value derived from constraining data is used to calculate a second value together with other constraining data, in which the `counter` is multiplied by two maximum values. Figure 7 shows one example of cascaded access, in which `index1`, `index2`, and `index3` are constraining data. Here we use the concrete value to estimate the maximum constraining data. At line 10, `tmp` is retrieved from `pattern_list` with `index1`, and its `counter` will be multiplied by `index1`. When `tmp` is saved into the list, each list element may have `counter*index1` sources. Then after another iteration, each element will have `counter*index1*index1` sources. In this way, the `counter` value increases quickly. When `tmp` is used at line 17, the allowed target number is very large.

Table 4: Real-world exploits prevented by μCFI.

Prog	CVE	Type	Exploit	Blocked?
ffmpeg	2016-10191	heap overflow	[2]	✓
	2016-10190	heap overflow	[6]	✓
php	2015-8617	format string	[20]	✓
nginx	2013-2028	stack overflow	[63]	✓
sudo	2012-0809	format string	[22]	✓

μCFI guarantees the UCT property regardless of large tables and cascading accesses, as it uses constraining data to get a unique target for each node, avoiding `counter` increase from the beginning.

5.2 Preventing attacks

We evaluate the effectiveness of μCFI at preventing real-world exploits, recently proposed advanced attacks, and synthesized attacks that bypass known defenses (including PittyPat).

We first collect 5 publicly available exploits against 4 vulnerable programs as listed in Table 4. ffmpeg is a popular multimedia framework for encoding and decoding videos and audios. It is vulnerable to two heap-based buffer overflow bugs, CVE-2016-10190 and CVE-2016-10191, which are exploitable to attackers to construct control-hijacking attacks. php is the interpreter of the PHP language, while sudo is a utility program on Unix-like systems for users to run programs with the privilege of other users. Both of them are vulnerable to format string vulnerabilities, i.e, CVE-2015-8617 for php and CVE-2012-0809 for sudo. As this type of vulnerability is highly exploitable, attackers simply launch control-hijacking attacks by corrupting code pointers. Nginx web server has a stack-based buffer overflow (CVE-2013-2028). We modify the exploit from [53] to carefully overwrite return addresses with their original values, and finally corrupt a sensitive structure pointer on the stack to launch forward-edge attacks. μCFI successfully detects all these CFI violations and halts their executions.

We also apply μCFI to protect the program introduced in PittyPat [21] that is vulnerable to COOP attack [55]. COOP is a Turing-complete attack method via fake object construction. As it corrupts forward-edge control-flow transfers to function entries, COOP poses a big challenge to coarse-grained CFIs. μCFI prevents COOP attacks by protecting all control data, which allows it to accurately track the function pointers in memory. When the program is fed a malicious input, μCFI successfully discriminates between legitimate and counterfeit objects to detect the attack.

At last, we evaluate μCFI on synthesized attacks that can bypass analysis without constraining data, like in PittyPat. We modify the source code of sjeng and gobmk to introduce two bugs, and build attacks to corrupt function pointers retrieved from large arrays. As we demonstrate in §2.2, existing CFI solutions cannot prevent such attacks because they overestimate all array elements as allowed targets. μCFI detects the inconsistency between the real target and the result of our analysis, and thus blocks both attacks.

5.3 Overhead measurement

Table 3 summarizes the overhead of μCFI in terms of execution time, peak memory use and compiled code size.

Performance overhead. On average, μCFI introduces 7.88% execution overhead to evaluated SPEC benchmarks. We break down

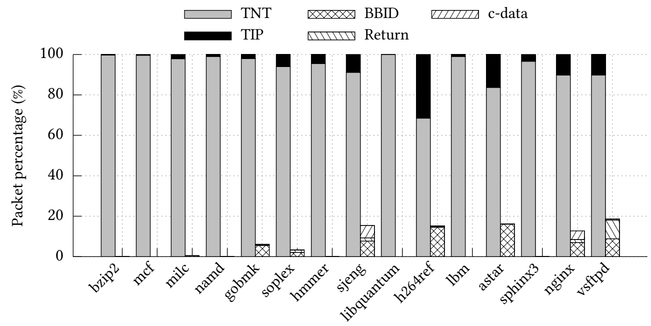

Figure 8: Trace reduction. Complete path construction uses TNT and TIP packets, while μCFI only requires BBID, Return and c-data.

the overhead to two components: that by instrumentation for efficient tracing and that by synchronization for CFI validation. As shown in the `instru` column under `Time Overhead` in Table 3, code instrumentation leads to less than 3% overhead, while the `+monior` column shows that monitoring further increases overhead by 5.23%. The overall overhead can be less than the instrumentation overhead. We believe this is due to non-determinism like caching and paging. We also calculate the average overhead for SPEC by excluding benchmarks without any ICT instructions, specifically, milc, libquantum and lbm. The result shows that μCFI still performs efficiently, introducing 10.10% overhead.

For real-world applications, μCFI introduces 4.05% overhead to nginx, and 0.75% overhead to vsftpd for requesting 1K files. We also measure the request for larger files, and find the overhead is negligible. Requesting large files invokes more `write` system call, and thus triggers more synchronizations between the monitor and the protected execution. However, as there is no pending CFI checks between system calls, μCFI immediately resumes the execution. In fact, such heavy I/O operations amortize the instrumentation in the main program, and thus lead to less overhead.

μCFI has less overhead than PittyPat (17.9% for SPEC and 11.9% for nginx) for two reasons. First, enforcing the UCT property makes our analysis more efficient. For example, for an assignment operation, the analysis copies the target set from the source to the destination. μCFI only copies one target, whereas PittyPat has to copy a large set (e.g., 1200 targets in h264ref). Second, our method of path reconstruction avoids generating and parsing the TNT packets that predominate PT traces. Figure 8 compares the necessary packets for complete control-path construction (TNT, TIP) against our partial path construction (BBID, Return, c-data). The TNT packets account for over 90% of the whole PT trace in most cases. Our trace size is negligible in comparison.

μCFI introduces relatively higher overhead to some benchmarks, like 25% for h264ref and 50% for perlbench. We examine the code of h264ref and find that it performs a large number of indirect function calls within a small time window, which creates a burden on the kernel task processing the PT trace. We can address this problem by allocating more kernel tasks for PT parsing, or moving our analysis into the kernel space, like in Griffin [23]. Overhead on perlbench mainly comes from two aspects: high percentage of sensitive instructions and frequent forking of child processes. About

20% of perlbench instructions are considered sensitive and half of basic blocks are instrumented for dumping their BBIDs. Further, perlbench creates 66 child processes with the heavy `fork` system call, which triggers the monitor forks in the same way, thus slowing down the execution. We can reduce the overhead as follows.

Optimization opportunity. We identify a promising direction to reduce our overhead with the new hardware feature — the `PTWrite` instruction from Intel PT. The `PTWrite` instruction directly writes user-provided data as a TIP packet into the PT trace. μCFI can utilize this instruction to log BBID and constraining data. Compared to our `write_data` function which contains a bunch of instructions, `PTWrite` is more compact and thus more efficient. With this instruction, we can significantly reduce the performance overhead.

Memory overhead. We measure the memory usage of the protected program and present the results in the `Memory Ovrhd` column of Table 3. At 0.81%, the memory increase for SPEC benchmarks is negligible. For nginx and vsftpd, μCFI introduces less than 6% overhead. Considering the large amount of memory on contemporary devices, such increase is acceptable.

Code overhead. μCFI compiler introduces extra code into the protected binary, including a fixed-size part and a program-dependent part. The fixed part contains functions for data collection and trace reduction, which is the same for any program. We allocate about 4MB for the fixed part to support logging constraining data in the range of [-1024, 4M-1024]. Considering the large code base in modern programs (e.g., browsers) and advanced memory sharing techniques (e.g., memory deduplication [29, 41]), this size overhead is acceptable. Another part of the overhead comes from the instrumented calls and return redirection, and is shown in the `vCode Ovrhd` column of Table 3. μCFI introduces little code overhead in this part for most SPEC benchmarks (4.04% on average). For perlbench, the code size overhead is 32.13%. The reason is that half of its basic blocks contain sensitive instructions and thus are instrumented with extra calls to `write_data` to record their BBID.

5.4 Compatibility with shadow stack

We measure the compatibility of μCFI with integrated parallel shadow stack (PSS) protection. We compile each program with the μCFI compiler and PSS and measure the execution correctness and performance overhead. All test programs including SPEC benchmarks, nginx, and vsftpd, work well with benign inputs, demonstrating the strong compatibility of μCFI. The overhead of integrating PSS is shown in the `+stack` column in Table 3. On average, PSS introduces a 2.07% overhead to evaluated SPEC benchmarks and negligible overhead to nginx and vsftpd. Although parallel shadow stack works well with μCFI, we do not claim any contribution nor provide any guarantee on backward-edge CFI. By showing the compatibility of μCFI with shadow stacks, we clarify that any alternative solutions with various security guarantees, like randomization-based SafeStack [36] and hardware-based Intel CET technique [33], can be integrated with μCFI to provide UCT property on both directions.

6 DISCUSSION & FUTURE WORK

In this section, we discuss several important topics regarding Intel PT and μCFI. First, we analyze the security guarantee of μCFI and the attacks it can prevent. Then we compare μCFI with a closely related work, code-pointer integrity (CPI) [36]. Next we present the data-loss problem of Intel PT, and discuss the missed SPEC benchmarks due to data-loss. At last, we list the future work of handling the less common but challenging ICTs, like exceptions.

6.1 Security promise by μCFI.

As μCFI monitor asynchronously checks the target of each ICT instruction after its execution, it is possible that the attack has been launched for a while before we detect it. However, μCFI still provides a strong security guarantee as follows. First, attackers cannot make significant damage through security-critical system calls. μCFI synchronizes the monitor and the protected execution before the latter enters the high-privileged kernel space through security-critical system calls. The synchronization temporarily pauses the protected execution until the monitor finishes all CFI checks. Therefore, the monitor verifies the targets of all executed ICT instructions, and detects any individual unexpected behavior.

Second, attackers cannot clean its attack trace to bypass detection. Intel PT logs each ICT target (including the corrupted one) into the kernel space immediately after the instruction's execution. Without invoking security-critical system calls, attackers cannot touch the PT trace. But once they invoke such system calls, the monitor pauses the execution before entering kernel, checks all executed ICT instructions with the clean PT trace and detects the attack.

At last, attackers cannot overflow PT trace in kernel to bypass detection. A smart attacker may keep running ICT instructions in user-space to generate a huge number of PT packets, aiming to overflow the PT trace. However, this attack does not work on μCFI. We set a limit of kernel memory used for PT trace. Once the limit is hit, μCFI suspends the protected execution until the monitor completes the checking phase and creates new memory quota. Since the normal execution usually does not trigger the limit, we treat frequent limit hit as a hint of attack.

6.2 Security analysis and CPI

Whether μCFI can prevent a concrete attack depends on the type of the attack-corrupted data. Specifically, (1) attacks corrupting code addresses (i.e., return address and function pointer) can be detected and blocked. (2) Attacks corrupting data with no relationship to control-flow can survive. In the case when the corrupted data indirectly affects the control-flow, μCFI can detect the attack if (3) the corrupted data is a pointer that affects the control-flow; (4) otherwise, μCFI cannot detect it. For example, μCFI cannot detect non-control-attacks [14, 56] because they fall into either case (2), such as corrupting the user identity variable; or case (4), such as corrupting the authenticated flag. For real-world control-hijacking attacks, like the ones in §5.2, which fall into either case (1) or (3), μCFI can detect and block them.

μCFI prevents the same set of attacks as code-pointer integrity, which enforces memory safety on control-data to prevent corruption in the first place. Both works protect the same set of program data, i.e., the *sensitive pointer* in CPI, and the union of the sensitive instruction and the constraining data in this paper. The main difference is the mechanism used to realize the protection. CPI uses in-process isolation to prevent low-privileged code from accessing sensitive data, and checks access boundary in high-privileged code to avoid corruptions. Instead, μCFI relies on out-of-process monitor to remove the low-privileged code from the attack surface, and further leverages an "infinite" memory model to preclude possible memory errors in the monitor. Regarding implementations, currently CPI uses code segment to provide isolation on x86 system, and relies on information hiding to secure the sensitive data on x86_64 system. We implement μCFI on x86_64 system with process isolation and Intel PT, and the idea is also applicable on x86 system.

6.3 Reliability of Intel PT

Intel PT has been explored in several works [21, 23, 28] to improve control-flow integrity. However, we find that PT packets can be lost at the hardware level. Specifically, some packets are dropped from the PT trace, even if the software driver faithfully copies every bit from the hardware buffer. This problem is more severe if one program generates a large number of PT packets within a short time window. As μCFI requires all necessary PT packets shown in their generation order to reconstruct the control flow, any data-loss renders our protection fail due to missed operations. Originally, we find this problem in almost all SPEC benchmarks, in which we generate traces to include all types of packets.

We inspect this problem and identify that TNT packets, which usually dominate the PT trace, contribute the most to the data-loss problem. Therefore, we mitigate this problem with novel encoding techniques to circumvent TNT packets and some others, as we discuss in §4. Finally, we did not see any packet loss for evaluated SPEC programs and real-world applications. However, data-loss problem still exists for some SPEC programs, especially for C++ ones which keep generating a lot of PT packets, like xalancbmk. We have to skip these benchmarks in our evaluation.

We report the data-loss problem to the corresponding team in Intel. They acknowledge this problem, and express supportive attitude to use PT for security. We believe our work shows the promising security benefit from complete PT trace, and will help Intel accurately measure the value of fixing the data-loss problem. Although Intel does not provide a concrete plan of fix, we do observe that newer generations of Intel CPUs have fewer lost packets on the same workload. We will keep eyes on Intel's progress of solving this problem, and expect a complete fix in the near future.

6.4 Unsupported benchmarks

We skip several SPEC benchmarks in our evaluation due to the data-loss problem, including gcc, dealII, povray, omnetpp and xalancbmk. We will evaluate μCFI on these benchmarks once Intel fixes the data-loss problem, or releases newer productions with minimal lost data. Among them, C++ benchmarks usually contain more sensitive data due to the C++ polymorphism, and are likely to have higher overhead with μCFI. Polymorphism introduces a pointer of one function pointer table to each object, which is considered as sensitive data in μCFI. Therefore, μCFI executes more instructions in the monitor to capture all valid operations on control-data to enforce the UCT property.

However, we believe the performance number of μCFI on these benchmarks will be better than that reported in PittyPat, and will be more efficient than existing memory safety solutions. As we discuss in §5.3, μCFI takes the same structure as PittyPat to perform online point-to analysis, but with significantly less PT packets and accurate (unique) target for each control data (shown in Figure 8). Therefore, the enforcement by μCFI is more lightweight and robust. μCFI also performs better than existing memory safety solutions. For example, Softbound [42], the commonly referred memory safety solution, introduces about 250% overhead to the benchmark h264ref, while μCFI only introduces 25% slow down. For benchmark sjeng, the overhead is about 80% for Softbound, while only about 18% for μCFI. Therefore, we strongly believe even in our missed benchmarks, μCFI is likely to be efficient than existing memory safety solutions.

6.5 Future work

A common challenge for CFI systems is validating control-flow changes caused by signals [9] and exceptions [55]. Considering that signal handling and exception handling are OS-dependent, we leave them as future work. μCFI can be extended to handle these cases, as Intel PT by default records the targets of signals and exceptions in *FUP* packets. We can label the structures used to register and store handler data as constraining data, and record them with our technique for arbitrary data collection. In the monitor, we can save these data structures, and use them to check with the FUP packets to validate the control-flow transfers caused by signals and exceptions.

Another challenge for CFI systems is validating the edges in dynamically loaded code (e.g., shared libraries). Other works address this problem using modular CFI [46, 47]. We choose to focus on protecting the main binary in μCFI, and model a set of well-known library functions to guarantee the correctness of the points-to analysis, like `memcpy` and `malloc`. Our techniques also apply to libraries, and we leave the stitching of our models at runtime to future work.

7 RELATED WORK

Control-flow attacks are the predominate method to exploit memory errors. The attack method has evolved from code injection to code reuse, in which code snippets in the victim program are chained to achieve expressive attacks, like ret2libc [45] and return-oriented programming (ROP) [7–9, 13, 56, 57]. Researchers have proposed randomization techniques to mitigate code-reuse attacks [4, 5, 16, 19, 39, 40, 64]. For example, address space layout randomization (ASLR) is widely deployed in modern operating systems [51]. However, recent works [26, 27, 49] demonstrate that randomization-based solutions have inherent weaknesses and can still be bypassed. CFI is a principled solution to prevent control-hijacking attacks [1]. The idea is to statically draw a control-flow graph (CFG) to define all legitimate control-flow transfers and dynamically check the execution against the CFG. μCFI follows the idea of CFI, and proposes online points-to analysis with full execution context to achieve the strongest CFI enforcement.

Coarse-grained CFI solutions, like CCFIR [68] and BinCFI [69], achieve strong compatibility and good performance, but fail to provide strong security guarantee to eliminate all control-hijacking attacks [10, 25, 55]. Fine-grained CFI, like type-based CFI [46, 59, 62], significantly reduces the number of allowed targets. However, none of them can guarantee the UCT property, due to the missing execution context. Our system μCFI is the first work that guarantees the UCT property while introducing small performance overhead.

Several hardware features are used to provide efficient CFI enforcement, like branch tracing store [65], and last branch record [15, 50]. However, following works [11, 25] have demonstrated attacks against these efficient CFI solutions. Recent works [23, 28] use PT to record the complete execution path and validate the ICT with a static control-flow graph. However, these solutions are best-effort and over-approximate the set of valid targets due to the limitation of static analysis. PittyPat [21] performs online points-to analysis using the PT trace, but fails to enforce the UCT property due to the missing constraining data. μCFI utilizes full execution context to perform the points-to analysis, and thus is able to get the unique code target for each invocation of each ICT instruction.

Memory safety detects memory errors at runtime and thus prevents subsequent exploitation. Spatial memory safety guarantees that each memory access is within the expected boundary and prevents errors like buffer overflow and NULL-pointer dereference [3, 35, 42, 44, 67], while temporal memory safety detects access violations due to incorrect memory release and reuse, like user-after-free [18, 37, 60, 66]. Unfortunately, memory safety solutions introduce high overhead (usually over 100%) and make runtime hardening impractical. μCFI is a lightweight solution focusing on control data for better performance.

8 CONCLUSION

In this paper, we present the Unique Code Target (UCT) property for CFI, which guarantees that for each invocation of any indirect control-transfer instruction, there is one and only one allowed target. A CFI implementation enforcing the UCT property can stop all control-flow hijacking attacks that compromise control data. We prototype the first CFI system that satisfies the UCT property. Our system, μCFI, combines static program instrumentation with online points-to analysis to infer the unique code target. The evaluation shows that μCFI successfully enforces the UCT property for all protected programs, and stops real-world and advanced control-hijacking attacks while incurring less than 10% overhead.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their helpful feedback. This research was supported in part by the DARPA Transparent Computing program under contract DARPA-15-15-TC-FP006, by the ONR under grants N00014-17-1-2895, N00014-15-1-2162 and N00014-18-1-2662. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA and ONR.

REFERENCES

[1] 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.*

[2] Ali Juanquan. 2017. FFmpeg CVE-2016-10191. http://www.freebuf.com/vuls/148389.html.

[3] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation.*

[4] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making Fine-grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Conference on Security Symposium.*

[5] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.*

[6] bird. 2017. CVE-2016-10190 FFmpeg Heap Overflow. https://security.tencent.com/index.php/blog/msg/116.

[7] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy.*

[8] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.*

[9] Erik Bosman and Herbert Bos. 2014. Framing Signals - A Return to Portable Shellcode. In *Proceedings of the 35th IEEE Symposium on Security and Privacy.*

[10] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium.*

[11] Nicholas Carlini and David Wagner. 2014. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium.*

[12] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation.*

[13] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-Oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security.*

[14] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium.*

[15] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H Deng. 2014. ROPecker: A generic and practical approach for defending against ROP attacks. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium.*

[16] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy.*

[17] 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.*

[18] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium.*

[19] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium.*

[20] dctf. 2017. sploit.php. https://github.com/dctf/exploits/blob/master/CVE-2015-8617/sploit.php.

[21] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the 26th USENIX Security Symposium.*

[22] Aeon Flux. 2013. sudo 1.8.0 < 1.8.3p1 - 'sudo_debug' glibc FORTIFY_SOURCE Bypass + Privilege Escalation. https://www.exploit-db.com/exploits/25134/.

[23] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems.*

[24] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. Griffin Trace. https://github.com/TJAndHisStudents/Griffin-Trace.

[25] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Gerogios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy.*

[26] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining Information Hiding (and What to Do about It). In *Proceedings of the 25th USENIX Security Symposium.*

[27] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium.*

[28] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *Proceedings of the 7th ACM on Conference on Data and Application Security and Privacy.*

[29] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John C. S. Lui. 2017. SmartMD: A High Performance Deduplication Engine with Mixed Pages. In *Proceedings of the 2017 USENIX Annual Technical Conference.*

[30] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. 2016. TypeSan: Practical Type Confusion Detection. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security.*

[31] Michael Hicks. 2014. What is memory safety? http://www.pl-enthusiast.net/2014/07/21/memory-safety/.

[32] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy.*

[33] Intel. 2016. Intel Releases New Technology Specifications to Protect Against ROP attacks. https://software.intel.com/en-us/blogs/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks.

[34] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. 2013. ShadowReplica: Efficient Parallelization of Dynamic Data Flow Tracking. In *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security.*

[35] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference.*

[36] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation.*

[37] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium.*

[38] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. 2015. Type Casting Verification: Stopping an Emerging Attack Vector. In *Proceedings of the 24th USENIX Security Symposium.*

[39] Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium.*

[40] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.*

[41] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. 2013. XLH: More Effective Memory Deduplication Scanners Through Cross-layer Hints. In *Proceedings of the 2013 USENIX Annual Technical Conference.*

[42] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 30th ACM SIG-PLAN Conference on Programming Language Design and Implementation.*

[43] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 9th International Symposium on Memory Management.*

[44] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

[45] Nergal. 2001. The Advanced Return-into-lib(c) Exploits. http://phrack.org/issues/58/4.html.

[46] Ben Niu and Gang Tan. 2014. Modular Control-flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[47] Ben Niu and Gang Tan. 2014. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security.*

[48] Ben Niu and Gang Tan. 2015. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.*

[49] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. 2016. Poking Holes in Information Hiding. In *Proceedings of the 25th USENIX Security Symposium.*

[50] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Security Symposium.*

[51] PaX Team. 2003. PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[52] Tristan Ravitch. 2017. Whole Program LLVM. https://github.com/travitch/whole-program-llvm.

[53] Karlsruhe Institute of Technology. 2016. Exploitation Training – CVE-2013-2028: Nginx Stack Based Buffer Overflow . https://github.com/kitctf/nginxpwn.

[54] Tencent Xuanwu Lab. 2016. Return Flow Guard. http://xlab.tencent.com/en/2016/11/02/return-flow-guard/.

[55] 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 *Proceedings of the 36th IEEE Symposium on Security and Privacy.*

[56] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security.*

[57] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy.*

[58] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. 2016. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 23th Annual Network and Distributed System Security Symposium.*

[59] 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 *Proceedings of the 23rd USENIX Security Symposium.*

[60] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. 2017. DangSan: Scalable Use-after-free Detection. In *Proceedings of the Twelfth European Conference on Computer Systems.*

[61] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.*

[62] Victor van der Veen, Enes Goktas, Moritz Contag, Andre Pawlowski, 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 *Proceedings of the 37th IEEE Symposium on Security and Privacy.*

[63] w00d. 2013. Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028). http://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html.

[64] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM SIGSAC Conference on Computer and Communications Security.*

[65] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters. In *Proceedings of the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks.*

[66] Yves Younan. 2015. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium.*

[67] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security.*

[68] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy.*

[69] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium.*

A FORMAL RESULTS

In this section, we formally express and prove the correctness of our approach. In particular, we define the syntax (§A.1) and semantics (§A.2) of a core low-level language that we use to define our approach. We then formally define the problem that we address (§A.3) and our mechanism for protecting control security (§A.4).

A.1 Syntax

Figure 9 contains the syntax of a space of program instructions, `Instrs`. `Instrs` is defined over a space of disjoint sets of data registers $\texttt{Regs}_D$, code-pointer registers $\texttt{Regs}_C$, and data-pointer registers $\texttt{Regs}_P$. An instruction may operate over data values (Equation 1), set a value as an offset for pointer arithmetic (Equation 2), load data from memory to a register (Equation 3), store data in a register to memory (Equation 4), branch to the address in a code pointer (Equation 5), load a code pointer from memory into a register (Equation 6), store a code pointer in a register to memory (Equation 7),

$$
\begin{aligned}
\texttt{Instrs} := {} & \texttt{x := y + z} && (1)\\
& \mid \texttt{offset:=x} && (2)\\
& \mid \texttt{x := *p} && (3)\\
& \mid \texttt{*p := x} && (4)\\
& \mid \texttt{br c} && (5)\\
& \mid \texttt{c := *p} && (6)\\
& \mid \texttt{*p := c} && (7)\\
& \mid \texttt{p := q + offset} && (8)\\
& \mid \texttt{p := alloc x} && (9)\\
& \mid \texttt{p := *q} && (10)\\
& \mid \texttt{*p := q} && (11)
\end{aligned}
$$

Figure 9: The instructions, Instrs, of our target language. x, y, and z range over data registers $\texttt{Regs}_D$, c ranges over code-pointer registers $\texttt{Regs}_C$, and p and q range over data-pointer registers $\texttt{Regs}_P$.

compute a data pointer by adding a data pointer to an integer offset (Equation 8), allocate a memory object with size in a given data register (Equation 9), load a data pointer from memory into a register, or store a data pointer in a register to memory (Equation 11). Although all arithmetic operations are assumed to be binary, when convenient we will depict operations as using fewer registers.

A program is a map from instruction addresses to instructions. That is, for space of instruction addresses Addrs_C containing a designated *initial* address $\iota \in \mathsf{Addrs}_C$, the language of programs is $\texttt{Lang} = \mathsf{Addrs}_C \to \texttt{Instrs}$.

`Instrs` does not contain instructions similar to those in an architecture with a complex instruction-set, which may, e.g., perform operations directly on memory or call to and return from a procedure. The design of μCFI directly generalizes to analyze programs that use such an instruction set. In particular, the actual implementation of μCFI monitors programs compiled for the `x86` architecture.

A.2 Semantics

Each program $P \in \texttt{Lang}$ defines a language of sequences of program states, called runs, that are generated by executing a sequence of instructions in P from an initial state. A state is a frame that binds registers to values, paired with a heap that maps data addresses to values. Let Words be a space of data words, let Objs be a space of data objects. A *data address* is a data object paired with an integer offset; i.e., the space of data addresses is denoted $\mathsf{Addrs}_D = \mathsf{Objs} \times \mathsf{Words}$. The space of *values* is denoted $\mathsf{Values} = \mathsf{Words} \cup \mathsf{Addrs}_C \cup \mathsf{Addrs}_D$.

A *data frame* is a map from data registers to words paired with an offset value; i.e, the space of data-register frames is denoted $\mathsf{Frames}_D = (\texttt{Regs}_D \to \mathsf{Words}) \times \mathsf{Words}$. The spaces of *code-pointer frames* ($\mathsf{Frames}_C = \texttt{Regs}_C \to \mathsf{Addrs}_C$) and *data-pointer frames* ($\mathsf{Frames}_P = \texttt{Regs}_P \to \mathsf{Addrs}_D$) are defined similarly. The code-pointer and data-pointer frames of each initial program state map each code and data-pointer register to null.

A heap is a pair consisting of **(1)** a map $s : \mathsf{Objs} \to \mathbb{Z}$ from each object to its size and **(2)** a map $m : \mathsf{Addrs}_D \hookrightarrow \mathsf{Values}$ that for each $o \in \mathsf{Objs}$ and $i \in \mathbb{Z}$ with $0 \leq i < s(o)$, maps address (o, i) to some value. The space of all heaps is denoted Heaps.

A state is a tuple consisting of **(1)** the address of the current instruction, **(2)** a data frame, **(3)** a code-pointer frame, **(4)** a data-pointer frame, and **(5)** a heap. The space of states is denoted States.

Each instruction in `Instrs` implements a *transition function* τ that maps each pre-state and instruction to the resulting post-state. The definition of τ is standard and is thus omitted in order to simplify the presentation; the instructions that refer to symbol `offset` update and use the offset value in the data frame of the current state. For each program $\texttt{P} \in \texttt{Lang}$ and sequence of states r such that each pair of adjacent states in r are in the transition relation of a corresponding instruction in P, r is a *run* of P. The runs of P are denoted Runs(P).

A.3 Problem definition

In this section, we define the problem that we address in this work in formal detail. We first define spaces of program instrumenters and precise monitors, and then define conditions under which they are valid control-security monitors.

A procedure that, given a program, generates another program is a program *instrumenter*; i.e., the space of all program instrumenters is denoted $\mathsf{Instrumeters} = \texttt{Lang} \to \texttt{Lang}$. A procedure that, given a program and sequence of instruction addresses, outputs an instruction address is a *control monitor*; i.e., the space of control monitors is denoted $\mathsf{Mons} = \texttt{Lang} \times \mathsf{Addrs}_C^* \to \mathsf{Addrs}_C$.

The control trace of a run r is the sequence of targets of indirect branches taken by r. Our formal definition of a valid control monitor is expressed in terms of the code addresses visited over program runs and conditions under which one run of program simulates a run of another program. For each $\texttt{P} \in \texttt{Lang}$ and $r \in \mathsf{Runs}(\texttt{P})$, the sequence of targets of indirect control branches taken at states of r is the *control trace* of r, and is denoted $\mathsf{Trace}(r)$.

A run r is simulated by a run r' if for each state in r, there is a corresponding state in r' that maintains all state of r, and may optionally maintain additional state. Let r be a sequence of states $q_0, q_1, \ldots, q_n \in \mathsf{States}$. For $\texttt{P}' \in \texttt{Lang}$, let $r' \in \mathsf{Runs}(\texttt{P}')$ be a concatenation of n non-empty subsequences of states such that for each $0 \leq i < n$ and each state q' in the ith subsequence, q' has the data-register frame, code-pointer frame, and data-pointer frames of q_i over all registers bound in q_i (although q' may bind registers that are unbound in q_i), and q' has the heap of q_i. Then r is *simulated* by r', denoted $r \sim r'$.

Our work is intended to be applied in a security context in which one may both instrument a program before it is run and monitor the control branches taken by the instrumented program. Thus, we will address the problem of designing *control frameworks*, consisting of both a static program instrumenter and a dynamic control monitor, which may read the control branches chosen by an instrumented program and output a single instruction address to which the instrumented program may next transfer control.

Definition 1. *Let* $\texttt{P} \in \texttt{Lang}$, $r \in \mathsf{Runs}(P)$, $q \in \mathsf{States}$ *be such that* $r \cdot q \in \mathsf{Runs}(P)$, *and let* $a \in \mathsf{Addrs}_C$ *be such that* $\mathsf{Trace}(r \cdot q) = \mathsf{Trace}(r) \cdot a$. *Let* $I \in \mathsf{Instrumeters}$ *and* $M \in \mathsf{Mons}$ *be such that there is some* $r' \in \mathsf{Runs}(I(P))$ *such that* ***(1)*** $r \sim r'$; ***(2)*** $M(\mathsf{Trace}(r')) = a$. *Then I and M are a* valid control framework *for* P *and* $r \cdot q$.

The definition of a valid control framework ensures that any framework that exists is precise. In particular, because each control monitor is a function, each monitor in a framework, given a control trace of a program, may output only a *single* code address that may be the valid target of the monitored program's not indirect branch. Such a definition is not satisfied by previous approaches to online control-security enforcement, such as PittyPat [21]. Such approaches, after reading a control trace, may potentially allow any control target from a *non-singular set*. Thus, the analysis cannot implement a *function* from each trace to a single control target allowed next.

The problem that we address in this work is to develop a valid control framework for each program and its runs. To address this problem, we define a program instrumenter Instrumenter (§A.4.1) and a program monitor Monitor (§A.4.2) such that Instrumenter and Monitor are a valid control framework for each program and each of its runs.

A.4 Protecting control security

In this section, we formally define a program instrumenter (§A.4.1) and program monitor (§A.4.2).

A.4.1 Program instrumentation. Instrumenter, given a program `P`, generates a program `P`' such that each offset used to compute a pointer in `P` determines the target of an indirect branch in `P`'. `P`' uses two code registers that we assume, without loss of generality, are not used by `P`, namely `offsetTgt` and `nextInstr`. `P`' also contains a region of code starting at a fixed address `offsetCode`; the size of the region is the maximum value of a data word, denoted |Words|. The range of instruction addresses between `offsetCode` and `offsetCode` + |Words| is called the *offset-code region*. Each instruction in the code region is a noop, except for the final instruction, which indirectly branches to the code address stored in `nextInstr`.

`P`' is an instrumentation of `P` that, before each instruction that performs pointer arithmetic, translates the offset used to compute the new pointer to a corresponding indirect branch. Let $\texttt{p}, \texttt{q} \in \texttt{Regs}_P$ be such that `q:=p+offset` $\in$ `P`. `P`' includes the following additional instructions immediately before `P`:

1. An instruction that computes a branch target of `offsetCode` plus `x`: `offsetTgt:=offsetCode+offset`.
2. An instruction that binds the address of the next instruction to `nextInstr`: `nextInstr:=ip+1`.
3. An instruction that transfers control to the instruction at `offsetTgt`: `br offsetTgt`.

Instrumenter is a valid program instrumenter: given a program `P`, Instrumenter generates a program that simulates each run of `P`.

Lemma 1. *For each* $\texttt{P} \in \texttt{Lang}$ *and* $r \in \mathsf{Runs}(P)$, *there is some* $r' \in \mathsf{Runs}(\textit{Instrumenter}(P))$ *such that* $r \sim r'$.

Proof. Proof by induction on r. For the base case, r is an initial state σ, which is a data frame paired with an empty heap. σ is also an initial state of Instrumenter(`P`). Thus r is a run of Instrumenter(`P`) such that $r \sim r$.

For the inductive case, r is a initial run s, followed by states σ and σ'. By the inductive hypothesis, $s \cdot \sigma$ is simulated by some $s' \in \mathsf{Runs}(\textsc{Instrumenter}(\texttt{P}))$. The instruction $\texttt{i} \in \texttt{Instrs}$ on which σ transitions to σ' may have one of the following forms. If `i` is any instruction other than one that performs pointer arithmetic, then $s' \cdot \sigma'$ is in Runs(Instrumenter(`P`)) and $s \cdot \sigma \cdot \sigma' \sim s' \cdot \sigma'$. Otherwise, if `i` performs pointer arithmetic, then from σ, Instrumenter(`P`) steps through a bounded sequence of states Σ with instruction addresses in the offset-code region, and then the state σ'. $s \cdot \Sigma$ is in Runs(Instrumenter(`P`)) and $s \cdot \sigma \cdot \sigma' \sim s' \cdot \Sigma$. □

The key idea behind our approach is to monitor programs in a specific form that reflect values used in pointer arithmetic as targets of control branches. In particular, if each instruction that performs pointer arithmetic in each run of program `P` is a preceded by an instruction that branches to a target that encodes that encodes the offset, then we say that `P` *reflects* pointer-arithmetic offsets.

Definition 2. *Let* $\texttt{P} \in \texttt{Lang}$ *be such that for each* $r \in \mathsf{Runs}(P)$ *and each* $\sigma \in r$ *that steps using an instruction that performs pointer arithmetic, there is some* $\sigma' \in r$ *before* σ *and* $\texttt{c} \in \texttt{Regs}_C$ *such that* σ' *steps on instruction* `br c`, $\sigma(\texttt{offset}) = \sigma'(\texttt{offsetCode}+\texttt{c})$, *and there is no* $\sigma'' \in r$ *between* σ' *and* σ *and* $\texttt{d} \in \texttt{Regs}_C$ *such that* σ'' *steps on instruction* `br d` *and* $\texttt{offsetCode} \leq \sigma''(d) < \texttt{offsetCode} + |\mathsf{Words}|$. *Then* `P` reflects *pointer-arithmetic offsets.*

Instrumenter only generates programs that reflect pointer-arithmetic offsets.

Lemma 2. *For each* $\texttt{P} \in \texttt{Lang}$, *Instrumenter(P) reflects pointer-arithmetic offsets.*

Lemma 2 follows directly from the definition of Instrumenter.

A.4.2 Control monitoring. Monitor, given a program `P` and control trace T, outputs the only control target that may be taken next by a valid run of `P` that has executed T. We now define the domain of information about program states maintained by Monitor (§A.4.2) and then define how domain information is updated by simulating each instruction executed by `P` (§A.4.3).

Monitor States. Monitor maintains a code-pointer frame, data-pointer frame, and heap that soundly models the structure of data objects and code pointers in the `P`' heap, while retaining no information about the data values in the heap. I.e., let the countably-infinite space of *monitor objects* (used by the monitor to model data objects allocated by the monitored program) be denoted Objs_M. Let a *monitor address* be a monitor object paired with an offset; i.e., the space of monitor addresses is denoted $\mathsf{Addrs}_M = \mathsf{Objs}_M \times \mathbb{Z}$. A *monitor pointer frame* binds data pointer to monitor addresses; i.e., the space of monitor pointer frames is denoted $\mathsf{Frames}_M = \texttt{Regs}_P \to \mathsf{Addrs}_M$. A monitor heap is a partial map from monitor addresses to code addresses and monitor address; i.e, the space of monitor heaps is denoted $\mathsf{Heaps}_M = \mathsf{Addrs}_M \hookrightarrow \mathsf{Addrs}_C \cup \mathsf{Addrs}_M$. A monitor state is a tuple (a, o, C, D, H), where

1. The address $a \in \mathsf{Addrs}_C$ of the current instruction.
2. The offset $o \in \mathbb{Z}$ to be used in the next pointer-arithmetic instruction.
3. The code-pointer frame $C \in \mathsf{Frames}_C$ (§A.2) of the monitored program.
4. A monitor-pointer frame $D \in \mathsf{Frames}_M$ (§A.2) isomorphic (as defined below) to the data-pointer frame of the monitored program.
5. A monitor heap $H \in \mathsf{Heaps}_M$ isomorphic (as defined below) to the heap of the monitored program.

The space of monitor states is denoted States_M.

Each monitor state represents an infinite set of program states with identical code-pointer frames and isomorphic heaps. Let $Q \in \text{Frames}_P$, $Q' \in \text{Frames}_M$, $H \in \text{Heaps}$, $H' \in \text{Heaps}_M$, and $f : \text{Objs} \to \text{Objs}_M$ be such that **(1)** f is one-to-one, **(2)** for each $\texttt{p} \in \texttt{Regs}_P$, $f(Q(\texttt{p})) = Q'(\texttt{p})$, **(3)** for each $o \in \text{Objs}$ and $i \in \mathbb{Z}$, if $H(o,i) \in \text{Addrs}_C$, then $H'(o,i) = H'(f(o),i)$ and if $H(o,i) \in \text{Addrs}_D$, then $f(H(o,i)) = H'(f(o),i)$. Then for $a \in \text{Addrs}_C$, $D \in \text{Frames}_D$, $C \in \text{Frames}_C$, state $q = (a, D, C, Q, H)$ is *object-abstracted* by monitor state $q' = (a, D, C, Q', H')$, denoted $q \preceq_O q'$. If in addition, $q(\texttt{offset}) = q'(\texttt{offset})$, then q is *fully abstracted* by q', denoted $q \preceq q'$.

For each $\texttt{P} \in \texttt{Lang}$, the initial monitor state $\iota_M^{\texttt{P}} \in \text{States}_M$ consists of the initial instruction pointer of P, a default offset value, the initial code-pointer and data-pointer frames, and an empty monitor heap. For each initial state $\sigma \in \text{States}$ of P, $\sigma \preceq \iota_M^{\texttt{P}}$.

A.4.3 Interpretations of instructions over monitor states. Monitor, given P and a sequence of code addresses T, determines the sequence of instructions I that must be executed by any run of P that branches to the addresses in T in sequence. After reading the sequence, the monitor outputs the single valid control target of the next indirect branch of the monitored run.

To do so, Monitor models the effect of each instruction in I on the current state of P by appropriately updating a monitor state that it maintains. For $a \in \text{Addrs}_C$, $o \in \mathbb{Z}$, $C \in \text{Frames}_C$, $D \in \text{Frames}_M$, and $H \in \text{Heaps}_M$, each monitor pre-state $q = (a, o, C, D, H)$ and instruction $\texttt{i} \in \texttt{Instrs}$ define a unique monitor post-state, as follows.

- Each instruction `i` that performs arithmetic on data, loads from memory, stores data to memory only updates the instruction address of q to be the address following `i`.
- For each $\texttt{c} \in \texttt{Regs}_C$, instruction `br c` updates the instruction address of the maintained monitor to be $C(\texttt{c})$. If the branch target is within the offset code region (i.e., if $\texttt{offsetCode} \leq C(\texttt{c}) < \texttt{offsetCode} + |\text{Words}|$), then o is updated to be $C(\texttt{c}) - \texttt{offsetCode}$.
- For each $\texttt{c} \in \texttt{Regs}_C$ and $p \in \texttt{Regs}_P$, instruction `c:=*p` updates C to bind `c` to $H(\texttt{p})$. Instruction `*p:=c` updates H to map address $H(D(\texttt{p})$ to $C(\texttt{c})$.
- For all $\texttt{p}, \texttt{q} \in \texttt{Regs}_P$ and each $x \in \texttt{Regs}_D$, instruction `p:=q+x` updates D to bind `p` to $D(\texttt{q}) + o$.
- For each $\texttt{x} \in \texttt{Regs}_D$, instruction `p:=alloc x` updates D to bind a monitor object not in the domain of H to `p`.
- For all $\texttt{p}, \texttt{q} \in \texttt{Regs}_P$, instruction `q:=*p` updates D to bind `q` to $H(D(\texttt{p}))$. Instruction `*q:=p` updates H to map $H(D(\texttt{q}))$ to $D(\texttt{p})$.

For each $q \in \text{States}_M$ and $\texttt{i} \in \texttt{Instrs}$, the monitor state resulting from executing `i` on q is denoted $\tau_M(q, \texttt{i})$.

Interpretations of instructions over monitors preserve two key properties. First, the interpretation of each instruction preserves object abstraction.

Lemma 3. *For each $q \in \text{States}$, $q_M \in \text{States}_M$ such that $q \preceq_O q_M$, and $\texttt{i} \in \texttt{Instrs}$, it holds that $\tau(q, \texttt{i}) \preceq_O \tau_M(q_M, \texttt{i})$.*

Proof. Proof by cases on the structure of `i`. The proof follows directly from the definition of object abstraction offsets and τ_M on `i`. □

Second, interpretation of pointer arithmetic preserves full abstraction.

Lemma 4. *For each $q \in \text{States}$, $q_M \in \text{States}_M$ such that $q \preceq q_M$, $\texttt{p}, \texttt{q} \in \texttt{Regs}_D$, it holds that $\tau(q, \texttt{i}) \preceq \tau_M(q_M, \texttt{i})$, where $\texttt{i} \equiv \texttt{p:=q+offset}$.*

Proof. The proof follows immediately from the definition of full abstraction and the interpretations of pointer arithmetic over concrete states and monitor states. □

The fact that the transformer over monitor states soundly models the effect of each instruction supports the fact that the analysis monitor always soundly determines the unique next valid control-transfer target.

Lemma 5. *Let $\texttt{P} \in \texttt{Lang}$ be such that P reflects pointer-arithmetic offsets, $r \in \text{Runs}(P)$, $T \in \text{Addrs}_C^*$, and $a \in \text{Addrs}_C$ be such that $\text{Trace}(r) = T \cdot a$. Then Monitor$(T) = a$.*

Proof. Proof by induction on r. The key property established by induction on r is that after Monitor analyzes r with final state σ, monitor maintains a monitor state σ_M such that $\sigma \preceq_O \sigma_M$; if σ has the instruction pointer of an instruction that performs pointer arithmetic, then $\sigma \preceq \sigma_M$. For the base case, where r is a sequence consisting of only an initial state σ, the monitor state maintained by Monitor is $\iota_M^{\texttt{P}}$, and $\sigma \preceq \iota_M^{\texttt{P}}$.

For the inductive case, r is a sequence of states s, followed by states σ and then σ'. Monitor, given $\text{Trace}(s \cdot \sigma)$, maintains a monitor state σ_M such that $\sigma \preceq \sigma_m$, by the inductive hypothesis. If σ has the instruction pointer of an instruction that does not perform pointer arithmetic, then for $\sigma'_M = \tau_M(\sigma, \texttt{i})$ the monitor state maintained by Monitor, it holds that $\sigma' \preceq_O \sigma'_M$, by Lemma 3. Otherwise, if σ has the instruction pointer of an instruction `i` that performs pointer arithmetic, then s contains a step on a branch instruction into the offset code region, and the most recent such instruction branches to `offsetCode + offset`, by the assumption that P reflects pointer-arithmetic offsets. Thus the offset in σ_M is `offset` in σ, by the definition of τ_M for branch instructions. Thus $\sigma' \preceq \sigma'_M$, by Lemma 4.

The property proved by induction, combined with the definition of object abstraction, entail that Monitor$(T) = a$. □

The components of μCFI correctly enforce control security.

Theorem 1. *For each $P \in \texttt{Lang}$ and $r \in \text{Runs}P$, the system (Instrumenter, Monitor) is a valid control framework.*

Proof. The fact that (Instrumenter, Monitor) is a valid control framework follows directly from the definition of a valid control framework (Defn. 1), the fact that Instrumenter is a valid instrumenter (Lemma 1) that only generates programs that reflect pointer-arithmetic offsets (Lemma 2), and the fact that Monitor soundly determines each control target (Lemma 5). □