When Function Signature Recovery Meets Compiler Optimization

Yan Lin
Singapore Management University
yanlin.2016@phdcs.smu.edu.sg

Debin Gao
Singapore Management University
dbgao@smu.edu.sg

***Abstract*—Matching indirect function callees and callers using function signatures recovered from binary executables (number of arguments and argument types) has been proposed to construct a more fine-grained control-flow graph (CFG) to help control-flow integrity (CFI) enforcement. However, various compiler optimizations may violate calling conventions and result in unmatched function signatures. In this paper, we present eight scenarios in which compiler optimizations impact function signature recovery, and report experimental results with 1,344 real-world applications of various optimization levels. Most interestingly, our experiments show that compiler optimizations have both positive and negative impacts on function signature recovery, e.g., its elimination of redundant instructions at callers makes counting of the number of arguments more accurate, while it hurts argument type matching as the compiler chooses the most efficient (but potentially different) types at callees and callers. To better deal with these compiler optimizations, we propose a set of improved policies and report our more accurate CFG models constructed from the 1,344 applications. We additionally compare our results recovered from binary executables with those extracted from program source and reveal scenarios where compiler optimization makes the task of accurate function signature recovery undecidable.**

I. Introduction

Control-Flow Integrity (CFI) [1] is a promising technique in defending against control-flow hijacking attacks [2], [5], [24], [30] by enforcing that runtime control flows follow valid paths in the program's Control-Flow Graph (CFG). Many approaches [20], [25], [26], [31] opt for fine-grained CFGs obtained at compilation time due to their high accuracy. However, it is difficult to precisely recover CFGs at the binary level since compilers do not preserve much information in the process of compilation [19]. Most existing approaches had to conservatively consider all functions as potential targets of an indirect caller, resulting in loosened CFI policies [35], [36] which make these approaches vulnerable to various attacks [4], [9], [11], [13], [29].

Latest approaches [23], [32] recover function signatures at the binary level by following calling conventions and only allow control flows between callees and callers with matching function signatures. Although generally good accuracy had been reported, e.g., TypeArmor [32] achieved 83.26% and 79.19% accuracy in identifying the number of arguments at callees and callers, respectively, in this paper, we challenge this belief of high accuracy when dealing with *optimized* binary executables. We subject TypeArmor to the same set of applications as chosen in the original paper, which are now compiled with different compiler versions with new optimization strategies enabled and find that the accuracy drops to 72.89% and 72.27%. The accuracy goes even lower to 63.74% and 69.36% when analyzing more complicated applications (e.g., Binutils) even with the same compiler version used in the original paper.

Our further investigation shows that this is because compiler optimizations may violate calling conventions and result in unmatched function signatures recovered at valid callees and callers. For example, modern compilers may not set or reset an argument register explicitly at the caller if the intended value is already in the corresponding register. The non-existence of the value assignment instruction therefore confuses the recovery process and results in underestimation on the number of function arguments. As shown in Listing 1, the indirect call at line 2 has 4 arguments, but the compiled binary code (with optimization flag `-O2` by `clang`) does not prepare for any argument as shown at Line 15 – 20. Similarly, the compiler only sets the first two arguments (`%edi`, `%esi`) for the indirect call at Line 25 while it requires 3 arguments as shown at Line 7. Such errors in function signature recovery could lead to invalid function calls being allowed or, even worse, valid calls being inadvertently blocked.

In this paper, we systematically study how compiler optimizations impact the accuracy of function signature recovery on x86-64 platform, with obfuscated binary out of our scope since existing work has clearly shown how obfuscated code complicates static binary analysis [15]. Specifically, we first theoretically analyze the possible ways in which compiler optimizations could impact the accuracy of two most recent approaches in function signature recovery for CFI, namely TypeArmor [32] and τCFI [23], and then experiment with a large number of applications including Binutils[1], LLVM test-suite[2], as well as C/C++ applications from Github to evaluate the extent to which such complications arise on real-world applications. We recover the ground truth of function signatures of 552 C and 792 C++ applications compiled with `gcc-8` and `clang-7` with optimization levels `-O0` to `-O3` and compare them with results of TypeArmor [32], τCFI [23], and Ghidra [12] in recovering the number of arguments and argument types.

[1]https://www.gnu.org/software/binutils/
[2]https://llvm.org/docs/TestSuiteGuide.html

```
long test(long a, long b, long c, long d, long e,
     long f) {
  long sum1 = (*fptr1)(a,b,c,d);
  ......
  //function ldiv returns a struct
  ldiv_t ldivrs;
  rs = ldiv (1000000L,132L);
  long sum2 = (*fptr2)(a,rs.quot, rs.rem);
  if (sum2 > sum1)
    return sum2;
  else
    return sum1;
}
0000000000400650 <test>:
......
40065b: mov    %r9,(%rsp)
40065f: mov    %r8,%r12
400662: mov    %rcx,%r13
400665: mov    %rdx,%rbp
400668: mov    %esi,%r15d
40066b: mov    %rdi,%r14
40066e: callq  *0x200e04(%rip) # 601478 <fptr1>
......
40069e: mov    %r14d,%edi
4006a1: mov    %eax,%esi
4006a3: callq  *0x2009b7(%rip) # 601060 <fptr2>
```

Listing 1: An example when function signature recovery meet compiler optimization.

Results show that compiler optimizations have both positive and negative impacts on function signature recovery. First, optimizations make the identification of variadic functions more accurate as arguments are more likely to be moved to callee-saved registers than being moved onto the stack. At the same time, the elimination of redundant instructions due to optimization also simplifies the argument analysis at callers. However, compiler optimization could make identification of the number of arguments and the type inferencing at callees less accurate, because of the elimination of unused arguments and promotion/demotion of argument types.

In order to mitigate these inaccuracies, we propose our improved policies to recover the function signatures more accurately from optimized binaries. We evaluate our proposed policies with the same set of real-world applications and compare our accuracy with that of existing ones. Results show that, e.g., the likelihood of misidentifying variadic functions in C is reduced from 3.3% to 1.2%. Moreover, our policy can mitigate all issues caused by argument type demotion at callers and argument type promotion at callees. Finally, we look at the bigger picture of CFI policies recovered from binary executables and program source, empirically analyze the implication of errors they make, and reveal scenarios in which compiler optimization makes the task of accurate function signature recovery undecidable.

In summary, this paper makes the following contributions:

- We study how compiler optimizations impact function signature recovery and perform our evaluation on 1,344 real-world applications;
- We propose improved inferencing policies which result in much higher accuracy when experimenting with real-world applications; and
- We empirically compare function signatures recovered from executables and program source and identify cases where compiler optimization makes the task undecidable.

II. Background and Unified Notation

In this section, we first briefly present C/C++ calling convention and introduce our notations used in this paper, and then present the CFI policies used by the two most recent approaches TypeArmor [32] and τCFI [23].

A. Basic Calling Conventions in C/C++

On Linux x86-64, all arguments of a function are passed from the caller to the callee who is assumed to process every argument. Integer arguments are passed in registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9` in sequence, while `%XMM0` – `%XMM7` are used to pass floating-point arguments [21]. Additional arguments are pushed onto the stack in reverse order. The return value is stored in `%rax` with potentially the higher 64 bits stored in `%rdx`. Floating-point return values are similarly stored in `%XMM0` and `%XMM1`. Both TypeArmor and τCFI adhere to these calling conventions and do not consider deviations from them.

Variadic functions (such as `printf` in the C library) are used to maximize flexibility in argument passing. These functions accept a variable number of arguments which do not necessarily have fixed types.

B. Unified Notation

TypeArmor [32] and τCFI [23] reconstruct both callee and caller signatures by performing static binary analysis and then use this information to enforce Control-Flow Integrity between callees and callers with similar signatures. TypeArmor uses the number of arguments as the signature, while width (number of bits $p \in \{64, 32, 16, 8\}$) of the argument-storing registers is used by τCFI. Just like in existing approaches, we focus on function signature recovery for integer arguments and use $i \in [1, 6]$ to index the six argument registers.

Here we introduce our unified notation to describe the CFI policies TypeArmor and τCFI employ as well as our improved policy (see Section V). Note that the notations we introduce are mainly for explaining *what* the policies are, and we discuss more on *how* the policies are extracted in Section V-E.

1) Analysis of callees: Analysis of a callee function typically starts from the function entry and continues in a forward manner until the end of the function. Here, the analysis focuses on the *first* instruction involving a parameter-passing register, which could have one of the following four possible states: $s^{EE} \in \{\dot{w}(), r\dot{w}(), r\dot{w}2s(), c\}$ (we use the dot above a state to denote that it's the analysis result of the *first* instruction involving the corresponding register).

Definition II.1. State $\dot{w}_i(p)$ if the first instruction involving register i is writing into the lower p bits of register i.

Definition II.2. State $r\dot{w}_i(p)$ if the first instruction involving register i is reading the lower p bits of it and writing to anther register or a non-stack address.

Definition II.3. State $r\dot{w}2s_i(p)$ if the first instruction involving register i is reading the lower p bits of it and writing to a stack address.

Definition II.4. State c_i if register i is not involved in any instructions.

For example, for function `test` in Listen 1, states of the first five argument registers are $riw_1(64)$, $riw_2(32)$, $riw_3(64)$, $riw_4(64)$, and $riw_5(64)$, since 64 or 32 bits of these argument registers are read before (potential) new data is written to them. The state of the sixth argument register $s_6^{EE} = r\dot{w}2s_6(64)$ since 64 bits of `%r9` are moved onto the stack.

Definition II.5. Argument register state vector observed at callee $P_{EE}^{OB} =< s_1^{EE}, s_2^{EE}, s_3^{EE}, s_4^{EE}, s_5^{EE}, s_6^{EE} >$ where $s_i^{EE} \in \{\dot{w}_i(), riw_i(), r\dot{w}2s_i(), c_i\}$ for $i \in [1, 6]$.

For the example in Listing 1, $P_{EE-400650}^{OB} =< riw_1(64), riw_2(32), riw_3(64), riw_4(64), riw_5(64), r\dot{w}2s_6(64) >$.

Definition II.6. $b2b_i$ is true if $s_i^{EE} = r\dot{w}2s_i()$ and $s_{i+1}^{EE} = r\dot{w}2s_{i+1}()$ and the corresponding instructions involving registers i and i +1 are back to back.

For the example in Listing 1, $s_5^{EE} = rw_5(64)$ and $s_6^{EE} = rw2s_6(64)$; therefore $b2b_5$ is false.

2) Analysis of callers: Analysis of a caller function starts at the indirect call instruction and continues in a backward manner until it hits another function call instruction. This backward analysis follows the CFG and focuses on *all instructions* involving the parameter-passing register instead of only the first instruction as in the analysis of callees.

Definition II.7. State $w_i(p)$ if there is an instruction writing to the lower p bits of register i.

Definition II.8. State $\hat{w}_i$ if there is no instruction writing to register i.

At the caller, a register can be in either state, i.e., $s^{ER} \in \{w(), \hat{w}\}$. For example, for Line 23 – 25 in Listing 1, $s_1^{ER} = s_2^{ER} = w_2(32)$.

Definition II.9. Argument register state vector observed at caller $P_{ER}^{OB} =< s_1^{ER}, s_2^{ER}, s_3^{ER}, s_4^{ER}, s_5^{ER}, s_6^{ER} >$ where $s_i^{ER} \in \{w_i(), \hat{w}_i\}$ for $i \in [1, 6]$.

The state vector at caller $0x4006a3$ in Listing 1 is $P_{ER-4006a3}^{OB} =< w_1(32), w_2(32), \hat{w}_3, \hat{w}_4, \hat{w}_5, \hat{w}_6 >$, since 32 bits of data are written to `%rdi` and `%rsi`.

C. TypeArmor's Policy on the Number of Arguments

1) Callee: TypeArmor [32] performs a forward recursive analysis from the entry block to find out states of the six argument registers. If the state of the sixth argument register (`%r9`) is $r\dot{w}2s_6()$, TypeArmor concludes that this function is variadic and the number of arguments is the maximal i that makes $b2b_i$ false. For example, in Listing 1, TypeArmor concludes that it is a variadic function with 5 arguments. If the state of `%r9` is not $r\dot{w}2s_6()$, the function is considered non-variadic and the number of arguments is the maximal i with state $r\dot{w}2s_i()$ or $riw_i()$.

Definition II.10. The observed number of arguments at callee $|P_{EE}^{OB}|$ is:

$$\begin{cases} \underset{i}{\text{argmax}}(\neg b2b_i) & \text{if } s_6^{EE} = r\dot{w}2s_6() \\ \max(\underset{i}{\text{argmax}}(r\dot{w}2s_i()), \underset{i}{\text{argmax}}(riw_i())) & \text{otherwise} \end{cases}$$

2) Caller: TypeArmor iterates over each indirect caller and performs a backward static analysis to detect the number of arguments prepared. If the states of all argument registers are $w()$, TypeArmor stops the analysis and considers that the caller prepares the maximum number of arguments. If some argument registers are neither $w()$ nor $\hat{w}$, TypeArmor performs a recursive backward analysis on incoming control flows. In cases where incoming control flows are via indirect calls and therefore backward analysis fails in identifying the caller function, TypeArmor assumes that the maximum number of arguments is prepared. It also assumes that the argument registers are always reset between two function calls, and therefore analysis is terminated when a return edge is encountered. In summary, the number of arguments at the caller is the minimal i with state $\hat{w}_i$ minus one.

Definition II.11. The observed number of arguments at caller $|P_{ER}^{OB}|$ is:

$$\begin{cases} \underset{i}{\text{argmin}}(s_i^{ER} = \hat{w}_i) - 1 & \text{if } \exists \hat{w}_i \in P_{ER}^{OB} \\ 6 & \text{otherwise} \end{cases}$$

Since there could be overestimation at callers and underestimation at callees, TypeArmor allows caller A to call callee B if and only if $|P_{ER-A}^{OB}| \geq |P_{EE-B}^{OB}|$.

D. τCFI's Policy on the Width of Arguments

τCFI [23] is the follower of TypeArmor that constructs a more fine-grained CFG by additionally considering the widths of argument registers as function signatures. It analyzes the number of bits of argument registers that are read or written to at callees and callers, respectively. We use $|s^{EE}|$ and $|s^{ER}|$ to represent the with of arguments at callees and callers, respectively. For example, if $P_{EE}^{OB} =< \dot{w}_1(), \dot{w}_2(), \dot{w}_3(), \dot{w}_4(), riw_5(64), riw_6(64) >$, then $|s_1^{EE}| = |s_2^{EE}| = |s_3^{EE}| = |s_4^{EE}| = 0$ and $|s_5^{EE}| = |s_6^{EE}| = 64$.

Since the analysis could cause overestimation at callers and underestimation at callees, the CFI policy of τCFI is: caller A can transfer control flow to callee B if and only if: $\forall i \in [1, |P_{ER}^{OB}|], |s_i^{ER}| >= |s_i^{EE}|$.

We also denote the ground truth for the states of argument registers at callees and callers as P_{EE}^{GT} and P_{ER}^{GT}, respectively. $|s^{EE,GT}|$ and $|s^{ER,GT}|$ are used to denote the ground truth on the width of arguments.

III. Eight Ways in which Compiler Optimization Impacts Function Signature Recovery

In this section, we present our analysis in binary optimization strategies and how they impact the accuracy of function signature recovery. Specifically, we study the source code of compilers (`gcc-8` and `clang-7`), paying special attention to the mechanism in which arguments are passed from callers

to callees under different optimization flags (`-O0`, `-O1`, `-O2`, `-O3`). We also consult the Intel instruction manual [14] on how each instruction could affect function signatures. Finally, we compile the following eight scenarios in which compiler optimization could impact function signature recovery by the two most recent work, namely TypeArmor and τCFI.

A. Complications at Callees

1) Misidentifying variadic functions: As outlined in Section II-C, TypeArmor uses $\dot{rw}2s_6()$ as the sole indicator of a variadic function. Interestingly, such a policy tends to introduce more errors in unoptimized binaries in which all arguments are moved onto the stack and any normal function with more than five arguments will be misidentified as variadic. We denote this complication as ***Nor2Var***. On the other hand, optimized binaries tend to move arguments to callee-saved registers, which reduces the chances of such errors. That said, normal functions in optimized binaries may still use the stack for parameter passing if the compiler determines that the argument will be reused after the call.

Listing 2a shows a function compiled with `clang -O2`. Since $s_6^{EE} = \dot{rw}2s_6()$, $b2b_5$ is true and $b2b_4$ is false. TypeArmor determines that `coff_write_symbol` is a variadic function with 4 arguments. However, $|P^{GT}_{EE-0x471a60}| = 7$ as shown at Line 1.

Another complication arises when a variadic function does not use some of the variadic arguments. An optimized binary will not explicitly read these arguments, which will cause the variadic function to be misidentified as normal (denoted as ***Var2Nor***). Note that this does not affect binaries compiled by `clang` since `clang` always explicitly reads all variadic arguments.

Listing 2b shows a variadic function `bfd_set_error` compiled by `gcc -O2`. As shown at Line 6 – 7, only the first two variadic arguments are used by this function, and therefore `gcc` only moves `%rsi` and `%rdx` onto the stack (Line 16 – 17). Current approaches would find that $P^{OB}_{EE-0x328c0} =< \dot{rw}_1(32), \dot{rw}2s_2(64), \dot{rw}2s_3(64), c_4, c_5, c_6 >$ and determine that $|P^{OB}_{EE-0x328c0}| = 3$ since `%r9` is not moved onto the stack. However, $|P^{GT}_{EE-0x328c0}| = 1$ as shown at Line 1.

Moreover, instructions that move the variadic arguments onto the stack in an optimized binary may not be back to back, which results in $b2b$ being unreliable in determining the number of arguments — an overestimation (denoted as ***VarOver***). Listing 2c shows the variadic function `concat_copy` compiled by `gcc -O2`. TypeArmor and τCFI find $b2b_5$ to be false and determine that it is a variadic function with 5 default arguments, but the ground truth is that it has only 2 default arguments as shown at Line 1.

2) Missing argument-reading instructions: When optimization is enabled, there may not be explicit reading of an argument if the function does not use it, leading the corresponding state of the argument to be c. We denote this complication as ***Unread***. As shown in Listing 3, since the first and third arguments of `jpeg_free_large` (compiled by `clang -O2`) are not used, TypeArmor and τCFI determine

```
1  static bfd_boolean coff_write_symbol (*,*,*,*,*,*,*,*)
2  0000000000471a60 <coff_write_symbol>:
3  ......
4  471a6e:    mov     %r9,0x40(%rsp)
5  471a73:    mov     %r8,0x10(%rsp)
6  471a78:    mov     %rcx,%r15
7  471a7b:    mov     %rdx,%r14
8  471a7e:    mov     %rsi,%rbp
9  471a81:    mov     %rdi,%r12
10 ......
11 471c4f:    mov     0x40(%rsp),%rbx
```

a: Normal function misidentified as variadic

```
1  void bfd_set_error (bfd_error_type error_tag,...) {
2    bfd_error = error_tag;
3    if (error_tag == bfd_error_on_input) {
4      va_list ap;
5      va_start (ap, error_tag);
6      input_bfd = va_arg (ap, bfd *);
7      input_error = (bfd_error_type)va_arg(ap,int);
8      ......
9    }
10 }
11 00000000000328c0 <bfd_set_error>:
12 ......
13 328c4:   mov     %edi,0x300186(%rip)
14 ......
15 328da:   cmp     $0x14,%edi
16 328dd:   mov     %rsi,0x28(%rsp)
17 328e2:   mov     %rdx,0x30(%rsp)
18 328e7:   je      32900
```

b: Variadic function misidentified as normal

```
1  char *concat_copy(char *dst, const char *first, ...)
2  00000000000dea00 <concat_copy>:
3  ......
4  dea25:     test %rsi,%rsi
5  dea28:     mov %rdx,0x30(%rsp)
6  dea2d:     mov %rcx,0x38(%rsp)
7  dea32:     mov %r8,0x40(%rsp)
8  dea37:     mov %rax,0x8(%rsp)
9  dea3c:     lea 0x20(%rsp),%rax
10 dea41:     mov %r9,0x48(%rsp)
```

c: Number of variadic arguments overestimated

Listing 2: Examples of variadic function misidentification

```
1  GLOBAL(void) jpeg_free_large (j_common_ptr cinfo,
       void FAR * object, size_t sizeofobject) {
2    free(object);
3  }
4  000000000041b6b0 <jpeg_free_large>:
5  41b6b0: mov     %rsi,%rdi
6  41b6b3: jmpq    400950 <free@plt>
7
8  caller site:
9  41b5a0: mov     0x70(%r14,%r15,8),%rsi
10 ......
11 41b5d3: mov     %r12,%rdi
12 41b5d6: mov     %rbp,%rdx
13 41b5d9: callq   41b6b0 <jpeg_free_large>
```

Listing 3: Not reading argument registers

that $P^{OB}_{EE-0x41b6b0} =< \dot{w}_1(64), \dot{rw}_2(64), c_3, c_4, c_5, c_6 >$. Note that compilers always set the argument registers at callers even if they are not used by the callee; see Line 11 – 12.

3) Misidentifying %rdx as an argument: Some registers have special usage in addition to passing arguments. For example, the third argument register `%rdx` can also be used to store return values when the size of the return value is larger than 64 bits. When there is a read operation on it, current approaches do not distinguish reading an argument

```
1  long test(long a, long b)
2  00000000004006a0 <test>:
3  ......
4  4006ae: callq  400490 <lldiv@plt>
5  4006b3: mov    %rbx,%rdi
6  4006b6: mov    %rdx,%rsi
7  4006b9: callq  *0x200db1(%rip)    # 601470 <fptr3>
8  4006bf: mov    %rax,%rbx
9  4006c2: callq  *0x2009a0(%rip)    # 601068 <fptr4>
```

Listing 4: Misidentifying %rdx as an argument

```
1  typedef unsigned int JDIMENSION;
2  void process_data_crank_post(j_decompress_ptr cinfo,
       JSAMPARRAY output_buf, JDIMENSION *out_row_ctr,
       JDIMENSION out_rows_avail) {
3    (*cinfo->post->post_process_data) (cinfo, NULL,
       NULL, 0, output_buf, out_row_ctr, out_rows_avail);
4  }
5  00000000000165c0 <process_data_crank_post>:
6  165c0:  sub    $0x10,%rsp
7  165c4:  mov    0x228(%rdi),%rax
8  165cb:  mov    %rsi,%r8
9  165ce:  mov    %rdx,%r9
10 165d1:  push   %rcx
11 165d2:  xor    %edx,%edx
12 165d4:  xor    %ecx,%ecx
13 165d6:  xor    %esi,%esi
14 165d8:  callq  *0x8(%rax)
```

Listing 5: Promoted argument pushed onto the stack

from reading the higher 64 bits of a return value. It could then result in an overestimation on the number of arguments. This complication is denoted as ***rdx***.

As shown in Listing 4, TypeArmor and τCFI determine that $P^{OB}_{EE-0x4006a0} =< \dot{rw}_1(64), \dot{w}_2(32), \dot{rw}_3(64), c_4, c_5, c_6 >$, and that it is a normal function with 3 arguments. However, $|P^{GT}_{EE-0x4006a0}| = 2$ and the reading of `%rdx` is for the higher 64 bits of the return value of function `lldiv`.

4) Argument (width) promotion: Some instructions may only work on 64-bit registers or memory, and optimization may prefer using 64-bit registers since using 32-bit registers would result in longer instructions. For example, the compiler uses `push` to pass arguments to callees (via the stack) when the flag "-mpush-arg" is enabled (e.g., when it is the 7th argument). However, `push` only allows 64-bit registers as operands, which leads to argument (width) promotion (denoted as ***Push***). Line 1 – 4 of Listing 5 shows that the fourth argument `out_row_avail`, whose type is unsigned int, is passed as the 7th argument at Line 3, and is pushed onto the stack at Line 10 (resulting in $\dot{rw}_4(64)$ instead of $\dot{rw}_4(32)$).

Another complication is due to the default width of operands of certain instructions, e.g., `lea` [14]. Compilers prefer reading a 64-bit register even if the width of the argument is 32 bits, since reading a 32-bit register requires a prefix `67H` (denoted as ***lea***). Appendix A shows an example of it.

B. Complications at Callers

1) Missing argument-writing instructions: Similar to missing argument reading instructions at callees as discussed above, compiler optimization may decide not to set or reset the value of a register explicitly at callers.

- Higher 64 bits of the return value used as the third argument (denoted as ***Ret***). `%rdx` is used to store the higher 64 bits of the return value. If the compiler finds that a function uses this value as the third argument, it will not explicitly reset `%rdx` again.
- Uninitialized variable as an argument (denoted as ***Uninit***). `clang` generates `undef` values for uninitialized variables and do not explicitly set these arguments [18], [22]. On the other hand, `gcc` initializes them to zero[3].
- Indirect calls in wrapper functions (denoted as ***Wrapper***). Indirect callers may not reset argument registers when their values are already in the corresponding registers especially for inlined functions.
- Argument values not modified between two calls (denoted as ***Unmodified***). `gcc-7` and above eliminates writing across functions when the argument register is set to the same value for two consecutive callers.

```
1  long test2(long a, long b){
2    //mesg and err are not initialized
3    char *mesg,*err;
4    lldiv_t res;
5    res = lldiv (31558149LL,3600LL);
6    long rl = (*fptr3)(a, res.quot, res.rem);
7    (*fptr4)(mesg,err);
8    printf("%s\n", buffer);
9    return rl;
10 }
11 000000000004006a0 <test2>:
12 ......
13 4006ae: callq  400490 <lldiv@plt>
14 4006b3: mov    %rbx,%rdi
15 4006b6: mov    %rax,%rsi
16 4006b9: callq  *0x200db1(%rip) # 601470 <fptr3>
17 4006bf: mov    %rax,%rbx
18 4006c2: callq  *0x2009a0(%rip) # 601068 <fptr4>
```

Listing 6: Missing argument-writing instructions

All the above except ***Wrapper*** leads to $\hat{w}$ and results in underestimation on the number of arguments. Here we present one example (Listing 6) in which the higher 64-bit return value and an uninitialized variable are used as arguments. The state vectors for the two indirect calls are $P^{OB}_{ER-0x4006b9} =< w_1(64), w_2(64), \hat{w}_3, \hat{w}_4, \hat{w}_5, \hat{w}_6 >$ and $P^{OB}_{ER-0x4006c2} =< \hat{w}_1, \hat{w}_2, \hat{w}_3, \hat{w}_4, \hat{w}_5, \hat{w}_6 >$, respectively, which lead to a finding of $|P^{OB}_{ER-0x4006b9}| = 2$ and $|P^{OB}_{ER-0x4006c2}| = 0$. However, by observing the source code at Line 6 – 7, we realize that $|P^{GT}_{ER-0x4006b9}| = 3$ and $|P^{GT}_{ER-0x4006c2}| = 2$. Additional examples can be found in Appendix B.

2) Registers storing temporary values: Since all argument registers are general-purpose registers, they could also be used as scratch registers to store temporary values, which could result in an overestimation on the number of arguments (denoted as ***Temp***). Listing 7a shows an example (compiled with `clang -O0`) with $P^{OB}_{ER-0x416015} =< w_1(64), w_2(64), w_3(64), w_4(64), \hat{w}_5, \hat{w}_6 >$ and $|P^{OB}_{ER-0x416015}| = 4$. However, according to the ground truth at Line 7, we can observe that $|P^{GT}_{ER-0x416015}| = 3$ and the write operation on `%rcx` is to store a temporary value. Note that compiler optimization can remove many redundant instructions that are used to store temporary values; and so it has a positive impact on

[3]https://github.com/gcc-mirror/gcc/blob/master/gcc/init-regs.c

```
1 ......
2 460ffc: mov     -0x18(%rbp),%rdi
3 461000: mov     -0xe8(%rbp),%rsi
4 461007: mov     -0xf0(%rbp),%rcx
5 46100e: add     $0x10,%rcx
6 461012: mov     %rcx,%rdx
7 #(*bed->elf_backend_reloc_type_class)(info, o, s->rela);
8 461015: callq   *%rax
```

a: Assembly compiled with clang -O0.

```
1 ......
2 438881: mov     0x30(%rsp),%rdi
3 438886: mov     %rbx,%rsi
4 438889: mov     %rbp,%rdx
5 43888c: mov     0x10(%rsp),%rax
6 #(*bed->elf_backend_reloc_type_class)(info, o, s->rela);
7 438891: callq   *0x208(%rax)
```

b: Assembly compiled with clang -O2.

Listing 7: Registers to store temporary values

```
1 546586:mov $0x8a01b0,%esi
2 54658b:mov $0x2000,%edx
3 546590:mov %r14,%rdi
4 #(*git_hash_update_fn)(*, *, size_t len);
5 546593: callq *0x28(%rax)
```

Listing 8: A constant and a pointer as arguments

this case; see the optimized binary in Listing 7b where $P^{OB}_{ER-0x438891} =< w_1(64), w_2(64), w_3(64), \hat{w}_4, \hat{w}_5, \hat{w}_6 >$ and $|P^{OB}_{ER-0x438891}| = |P^{GT}_{ER-0x438891}| = 3$.

3) Argument (width) demotion: To the opposite of argument promotion at callees, compilers may use a smaller-sized register (32-bit), since a 64-bit register may need a REX prefix [14] which increases the code size and affects the I-cache footprint. This applies to cases where

- Arguments are constants whose sizes are up to 32 bits (denoted as ***Imm***);
- Arguments are pointers pointing to .rodata, .bss, and .text sections (denoted as ***Pointer***); and
- Arguments are NULL pointers (denoted as ***Null***).

Listing 8 shows an example for these cases compiled by `clang -O2`. The ground truth at Line 4 shows $P^{GT}_{ER-0x546593} =< w_1(64), w_2(64), w_3(64), \hat{w}_4, \hat{w}_5, \hat{w}_6 >$, while TypeArmor and τCFI determine that $P^{OB}_{ER-0x546593} =< w_1(64), w_2(32), w_3(32), \hat{w}_4, \hat{w}_5, \hat{w}_6 >$ since the second argument (`0x8a01b0`) is a pointer pointing to the .rodata section, and the third argument (`0x2000`) is a 32-bit constant. Appendix C shows an example of the case of NULL pointers.

Appendix E presents a summary on the complications at both callees and callers with the last column indicating the consequences. Appendix D shows the case ***Prom*** which is a complication introduced by optimization although it does not result in unmatched function signatures.

IV. Experimental Results of the Eight Complications on Real-World Programs

Section III details our theoretical analysis by analyzing compiler optimization strategies. In this section, we test how the eight complications identified in Section III present themselves in real-world programs. Specifically, we use a test suite of programs comprising of 552 C and 792 C++ applications compiled with `gcc-8` and `clang-7` with optimization levels from `-O0` to `-O3` for x86-64, and compare analysis results of TypeArmor and τCFI with ground truths extracted. Since the source code of τCFI is not released, we implement it ourselves according to the description of the paper [23].

In addition to TypeArmor and τCFI which recover function signatures for the specific purpose of Control-Flow Integrity, we also include a well-known binary analysis framework, Ghidra [12] v9.1.1, into our experiments since it also performs function signature recovery for reverse engineering purposes. Besides its general-purpose nature which leads to less emphasis on precision of the function signature recovery, our preliminary analysis on its source code reveals the following distinctions when Ghidra is compared to TypeArmor and τCFI in their mechanisms of function signature recovery:

- Only functions with symbol information are correctly identified as variadic, while those without symbol information are simply assumed to be non-variadic;
- Only instructions immediately prior to (without control-flow transfers) a call instruction are considered potentially preparing for function arguments;
- Forward and backward analysis are constrained within the scope of a single function; and
- Width for each argument at callers is always 64 bits.

With this preliminary understanding, we expect Ghidra to perform less accurately compared to TypeArmor and τCFI in recovering function signatures.

Our test suite is composed of Binutils-2.26, LLVM test-suite, and a large number of C and C++ applications from Github. This composition ensures that (1) it contains a wide variety of realistic C and C++ binaries with sizes ranging from 0.07MB to slightly more than 100MB (see Appendix F for details of sizes of the binary executables); (2) it contains binaries used in the evaluation of previous work, making it possible to compare our results with the literature; (3) it includes real-world applications downloaded from Github which contain complex corner cases which "testbed" applications may not have (see Appendix G for details of the Github applications we choose — mainly those with many "stars").

A. Ground Truth and Statistics on the Ground Truth

Our objective of the experiments is to compare results from TypeArmor, τCFI, and Ghidra with ground truths to see how the complications identified in Section III present themselves in real-world applications. Here we first briefly explain how we obtain the ground truth in an automatic manner.

We base our ground truth on information collected by an LLVM [16] pass and on DWARF v4 debugging information [7] which is the default setting for `gcc` and `clang`. We use LLVM to collect source-level information, including the number and types of arguments for each function and indirect callers when the arguments are integers (using LLVM API `isIntegerTy(N)`) and pointers (using LLVM

APIs `isPointerTy()` and `isFunctionTy()`[4]). We also record the source line numbers of functions and indirect callers. We then compile the test applications with DWARF information and link the source-level line numbers with binary-level addresses using the DWARF line number table.

We implement the above with more than 500 lines of C++ code and more than 2,000 lines of python code. The result is a ground truth file for each binary in the test suite. With the ground truth collected, we perform statistical tests on our test suite to ensure that applications included could potentially present all variety of function signatures. Specifically, we count the number of arguments (ground truth) of all functions and make sure that there are sufficient numbers of functions with the number of arguments from 0 to 6; see Appendix H for details. We observe that there are more functions with between 1 and 3 arguments, and that C programs are more likely to have variadic functions. We also check the (ground truth) argument types for each function (see Appendix I). It appears that pointers are heavily used as function arguments, especially for C++ applications. This may imply that C++ applications are less likely to present complications on argument width demotion or promotion.

B. Metric Used and Overall Results

Since applications may have different numbers of functions and functions may have different numbers of indirect callers, we do not directly calculate the geometric mean as in TypeArmor [32] and τCFI [23]. Instead, we calculate the geometric mean of the *likelihood* that the callees and indirect callers present a complication in their function signature recognition. Specifically, we calculate the likelihood that the complications discussed in Section III cause under- and overestimation on the recovered function signatures. For example, application `addr2line` compiled with `clang -O0` has 2,019 normal functions among which 101 are misidentified as variadic and the identified number of arguments is underestimated. We first calculate the likelihood that a function is misidentified in this application ($101/2019$), and then use this number to compute the geometric mean for all applications in our test suite; see Figure 1[5] and Figure 2[6].

We discuss the detailed findings in the next two subsections. Note that complication case ***Unmodified*** only appears in one application (`mupdf`[7] compiled with `gcc`) and that ***Uninit*** and ***Ret*** do not appear at all in our test suite. We stress that this does not indicate insufficiency in our experiment, but rather the complications identified in our theoretical analysis (Section III) do not necessarily present themselves in real-world programs.

[4]We also check whether a struct argument has the attribute ByVal since clang will copy it onto the stack while considering it as a pointer.

[5]Likelihood is calculated against the number of normal functions for ***Nor2Var***, against the number of variadic functions for ***Var2Nor*** and ***VarOver***, against the total number of functions for ***rdx***, ***Unread***, ***Push***, ***lea*** and ***Prom***. See Appendix J for the number of various types of functions in our test suite.

[6]Likelihood is calculated against the total number of indirect calls. See Appendix J for the number of indirect calls in our test suite.

[7]https://mupdf.com/

C. Complications at Callees

***Unread*:** This is by far the biggest contributor to misidentification of function signatures at callees, where the fact that many functions do not read (some of) their arguments leads to underestimation of the number of arguments. It also potentially leads to underestimation of the width of an argument register whose evident reading instruction is missing while existence is implied (due to subsequent argument registers whose reading instructions being present). This complication presents more heavily in C++ programs due to the simplicity of many (callee) functions whose implementation does not require accessing the **this* argument. Another finding is that C++ applications compiled by `gcc` tend to have dead code eliminated, which makes them seemingly less vulnerable to this complication. Note that unoptimized binaries do not have this issue at all because compilers always insert argument reading instructions even if the callee function does not need them.

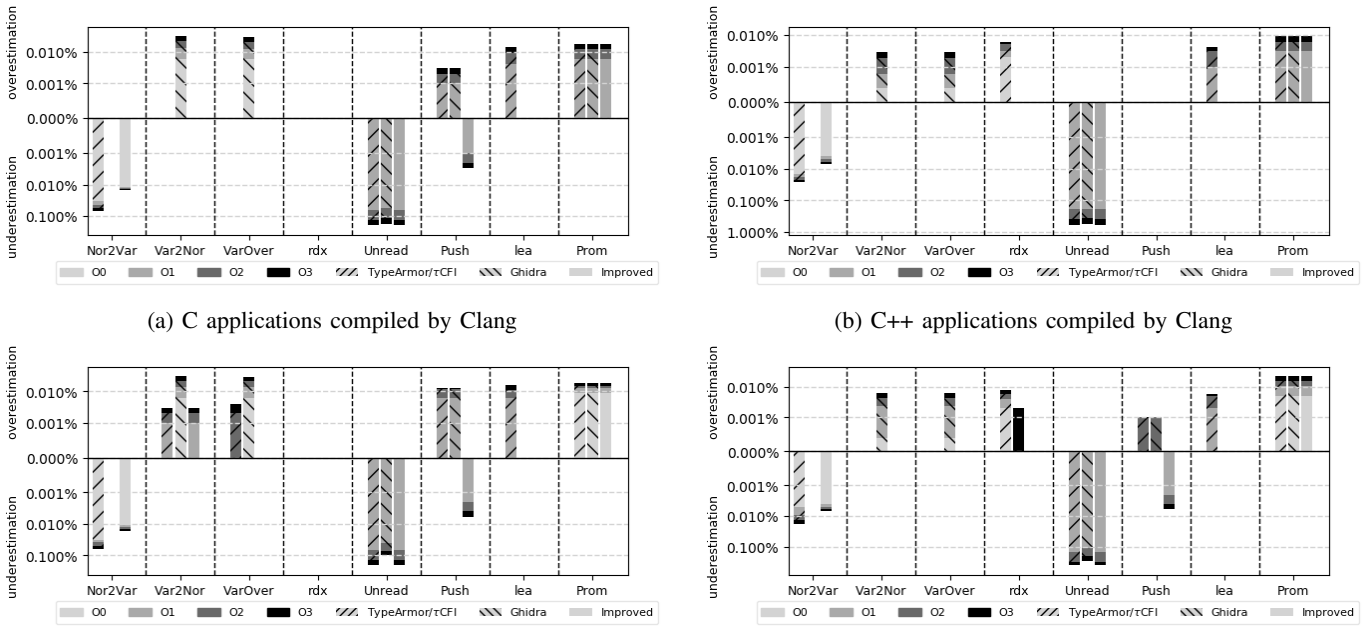
***Nor2Var*:** This also presents heavily in our test suite, leading to underestimation on the number of arguments, especially in C programs, except that compiler optimization actually helps mitigating it. As explained in Section III-A1, unoptimized binaries always move all arguments onto the stack, making it more likely to present more than 5 integer arguments at the callee which always leads to misidentification of variadic functions. Optimization helps "skipping" some of the arguments and reducing the likelihood of misidentification. Ghidra is immune to this complication since it simply considers all functions non-variadic.

lea*, *Push*, and *Prom*:** These three complications result in overestimation on the argument width, and together present a large thread to function signature identification of optimized binaries. Checking into the details, we find that C programs make heavier use of `lea` to perform simple computations and more often push arguments onto the stack (especially with `gcc`). Looking into the case of ***Prom, we find that `clang -O0` does not promote the argument width (it uses register `al` or `ax` to store the argument) while `gcc` does (it uses `eax`) even when optimization is turned off.

***rdx*:** This presents more on C++ programs and leads to overestimation on the number of arguments. Upon checking the details, we realize that the exception handling in C++ will call function `rethrow_exception`, which invokes function `_Unwind_RaiseException` that returns the unwind reason code in `%rdx` and the exception object in `%rax`.

***Var2Nor*:** As expected, Ghidra is vulnerable to this, although not that much due to compiler optimization but the simple treatment it employs (all functions are non-variadic). This complication presents to TypeArmor and τCFI, and is usually due to empty implementation of functions with more than five compulsory arguments. We find that C programs compiled by `gcc` suffer overestimation on the number of arguments on top of function type misidentification.

***VarOver*:** This only presents itself on binaries compiled with `gcc -O2` and `-O3`, where the instructions that move the variadic arguments onto the stack are not back to back. On the



(a) C applications compiled by Clang

(b) C++ applications compiled by Clang

(c) C applications compiled by GCC

(d) C++ applications compiled by GCC

Figure 1: Likelihood of complications at callees

other hand, all variadic functions are identified as non-variadic in Ghidra, so the number of arguments is overestimated.

D. Complications at Callers

Temp* and *Wrapper*:** These are clear examples in which compiler optimization helps TypeArmor and τCFI determining the number of function arguments. In the case of ***Temp, optimization eliminates redundant instructions as function arguments. ***Wrapper*** causes fewer complications in optimized binaries due to heavier applications of function inlining. Note that C++ applications are more vulnerable to ***Wrapper*** due to the large number of virtual functions being called indirectly. Ghidra generally performs worse here (considering the combined errors in both over- and underestimation) mainly due to its limited scope of backward analysis for indirect calls in wrapper functions. That said, Ghidra has superior mechanisms in dead code elimination and only the basic block which contains an indirect call is analyzed, which results in some argument registers that are used for temporary storage being correctly identified; see the complication of ***Temp*** (overestimation).

Imm* and *Null*:** C applications compiled with `clang` and `gcc` are both likely to pass immediate values to argument registers, which results in underestimation of the argument width by TypeArmor and τCFI. Interestingly, the likelihood increases upon increase of optimization levels. Digging into the details, we realize that this is actually just an artifact because higher optimization level results in heavier application of function inlining (`-O1` and `-O2` for `clang`, `-O1`, `-O2`, and `-O3` for `gcc`) and loop unrolling (`-O3` for both compilers), which leads to a larger number of callers of the same function; see Appendix K. Another interesting observation is that `gcc -O0` and `-O1` are more likely to move zero (Null***) to an argument register than using `xor`.

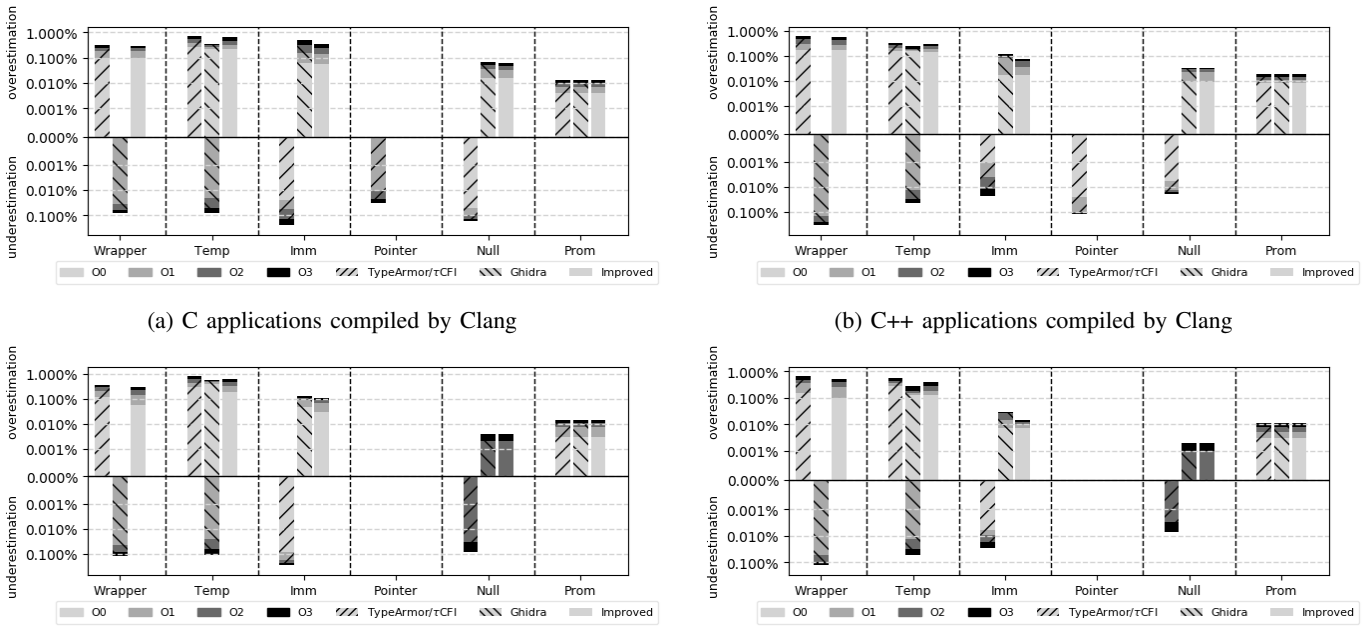
Ghidra, on the other hand, is not vulnerable to this underestimation but rather suffers on overestimation because it always uses the entire 64-bit memory range as the argument width.

***Pointer*:** This only affects applications compiled with `clang` especially on C++ programs as they are more likely to pass pointers to indirect callees. C programs compiled with `clang -O0` do not have this problem because it uses a 64-bit register to store the pointer by adding a prefix to denote the use of a 64-bit displacement or immediate source operand. C++ programs, on the other hand, set a 32-bit register to the pointer address and then move it to the argument register for some indirect calls. We also find that C++ applications compiled with `clang -O1` have a higher likelihood on this complication. This is because for some indirect calls that accept pointers as arguments, `clang` prepares them by moving 64-bit immediate values onto the stack first, and then after another indirect call instruction, the argument register is set by reading the 64-bit value from this stack address. As the number of indirect calls in binaries compiled with `-O2` and `-O3` is much larger, the likelihood for them becomes smaller.

Ghidra, again, is not vulnerable to this because it always uses the entire 64-bit memory range as the argument width.

Applications compiled by `gcc` do not use pointers that point to .text, .rodata, or .bss as arguments because `gcc-7` and above compile applications into position-independent code.

***Prom*:** This seems to be less sensitive to compiler optimization (compiler will always promote to the native type — 32 bits) and only affects a small number of indirect calls.



(a) C applications compiled by Clang

(b) C++ applications compiled by Clang

(c) C applications compiled by GCC

(d) C++ applications compiled by GCC

Figure 2: Likelihood of complications at callers

V. Our Compiler-Optimization-Friendly Policies

In an effort to properly handle the complications arisen due to compiler optimizations to more accurately recover function signatures, we propose a set of improved policies. In this section, we first discuss the details of these policies and then present our evaluation results of applying them to analyze our test suite of 1,344 real-world applications. Note that most of the policies proposed here are generally accurate for both optimized and unoptimized binaries, while others are more specifically targeting optimized binaries. Existing work [27], [28] and our experience (e.g., if values of all six argument registers are moved onto the stack, then it must be an unoptimized binary) show that detecting the compiler and the optimization level used in well-behaved binaries can be done accurately, and we take it as a prerequisite of enforcing our policies specifically targeting optimized binaries.

*A. Identifying Variadic Functions (Targeting **Nor2Var** and **VarOver**)*

The main problem in existing approaches is the identification of variadic arguments using "back-to-back value assigning instructions" (i.e., *b2b*) [32], which is not a sufficient condition as we analyzed (see Section III-A) and showed in experiments (see Section IV-C). We discover another more direct and sufficient condition for variadic argument identification when optimization is enabled, in which the stack addresses storing variadic arguments are consecutive, prepared using 64-bit registers, and read using pointers. More specifically,

Definition V.1. Let $@_i$ denote the stack address to which argument register i is moved given $\dot{rw2s_i}()$. Callee function f is a variadic function iff $\forall i \in \{5,4,3,2,1\}$,

- $|@_{i+1} - @_i| = 8$; and
- $s_{i+1}^{EE} = \dot{rw2s_{i+1}}(64)$ and $s_i^{EE} = \dot{rw2s_i}(64)$; and
- $@_{i+1}$ and $@_i$ are read via pointers.

with $|P_{EE-f}^{OB}|$ being the maximal i violating the above. Otherwise, f is a normal function and $|P_{EE-f}^{OB}|$ is:

$$\begin{cases} 6 & \text{if } \dot{rw2s_6}() \text{ and } @_6 \text{ is not read via a pointer} \\ \max(\underset{i}{\text{argmax}}(\dot{rw2s_i}()), \underset{i}{\text{argmax}}(rw_i())) & \text{if } s_6^{EE} \neq \dot{rw2s_6}() \end{cases}$$

We use the example in Listing 2a to show how our policy works. During analysis, we find that $P_{EE-0x471a60}^{OB} =< rw_1(64), rw_2(64), rw_3(64), rw_4(64), \dot{rw2s_5}(64), \dot{rw2s_6}(64) >$ and $@_6$ is not read via a pinter; therefore, we conclude that $|P_{EE-0x471a60}^{OB}| = 6$. Note that although $|P_{EE-0x471a60}^{GT}| = 7$, $|P_{EE-0x471a60}^{OB}| = 6$ is an accurate and best approximation based on the limited information present in the binary. The details about the analysis result by TypeArmor and our new policy can be found in Appendix L.

The policy described above does not work well when optimization is disabled, in which all arguments are copied onto the stack at consecutive addresses. Our policy to deal with unoptimized binaries is described in Appendix M.

*B. Argument (Width) Promotion and Demotion (Targeting **Push**, **lea**, **Imm**, **Pointer**, and **Null**)*

Our improved policy solves the argument promotion and demotion complications by analyzing the context of the instructions. More specifically,

- ***Push***: Let $p = 32$ in $rw_i(p)$ if the corresponding argument reading instruction is `push`.

- ***lea***: Let p in $riw_i(p)$ be the minimum of the width of the source and destination registers (instead of that of the source only as in TypeArmor and τCFI).
- ***Imm***: Let $p = 64$ in $riw_i(p)$ if register i holds a constant.
- ***Pointer***: Let $p = 64$ in $riw_i(p)$ if register i holds a pointer value pointing to .rodata, .bss, or .text section.
- ***Null***: Let $p = 64$ in $riw_i(p)$ if register i is involved in an `xor` instruction.

Note that this improved policy guarantees that all legal callers be matched with legal callees since there is no underestimation at callers or overestimation at callees, but could lead to some imprecise (but conservative) results. For example, demoting the argument width to 32 bit for a register read using `push` may result in underestimation; see the case of ***Push*** in Figure 1. We believe that this is a good tradeoff where an absolutely precise solution does not exist, especially since the intended control flow is never broken with our improved CFI policy.

C. Register Overloading (Targeting *rdx*)

Since the overloading of `rdx` is for storing function return values, we simply consider any first reading of `%rdx` after a call to a library function (let's denote the callee f) as $w_3()$. It may first sound counter-intuitive, but this must be reading the return value of f since the compiler has to make a conservative assumption that f has reset `%rdx`. This improved policy solves the complication ***rdx*** at callees with 100% accuracy.

D. Registers Storing Temporary Values (Targeting *Temp*)

Recall that the analysis of callers considers all instructions involving an argument-passing register instead of focusing on only the first instruction (Section II-B). Although that is technically correct, it also introduces complications since registers storing temporary values could be miscounted as passing parameters to a callee (***Temp***). Our improved policy takes into consideration the reading of registers (rather than focusing only on writing in the original policy) as well as the sequence of the instructions. More specifically, we let $s_i^{ER} = \hat{w}_i$ if register i is moved to another argument register after the write operation when the value of register i is not zero (a special case where the compiler will directly move register i to another argument register since the compiler does not prefer passing zeros to a register directly).

For example, as shown in Listing 7a, `%rcx` is moved to `%rdx` at Line 6 after the write operation at Line 4. With this, we conclude that `%rcx` is not used to pass arguments and $|P^{OB}_{ER-0x461015}| = 3$.

In order to be conservative, we only apply this policy to basic blocks where indirect calls are located. Note that this policy can also help correctly recover the number of arguments for indirect calls in wrapper functions.

E. Additional Binary Analysis to Extract our Policies

We have presented *what* our improved CFI policies are so far in this section. Here we briefly discuss *how* it is done with the additional binary analysis we perform.

Our improved policy for ***Nor2Var*** requires that we trace the data flow of a stack memory to check whether it is read without being overwritten. This is done by following the CFG of a function and check whether the stack memory is used as the source operand without being used as a destination operand.

Our improved policy for ***Imm*** requires that we identify whether one register holds a constant. Specifically, during the backward analysis, if we encounter a 32-bit argument register being written to, we will record its source recursively and check whether it is an immediate value. Our experiences show that this recursive tracing typically reports a success within the same basic block and does not result in excessive overhead.

F. Evaluation of our Improved Policies

We apply our new policies on the same test suite consisting of 1,344 C and C++ applications and use the same metric as described in Section IV-B to evaluate it; see the bars named "Improved" in Figure 1 and Figure 2. The comparison shows that our new policies result in significant improvement over most of the complication cases. In particular, we completely mitigate the complication cases of ***VarOver***, ***rdx***, ***lea***, and ***Pointer***, and significantly reduce the chances of running into ***Nor2Var***.

For cases of ***Imm***, ***Null***, and ***Push***, our policy guarantees that valid calls are never inadvertently blocked, but it could also potentially make the recovered function signatures more conservative. For example, we promote the argument width at indirect callers for cases ***Imm*** and ***Null***, which may result in overestimation on argument widths as shown in Figure 2 with likelihood less than 10.1% and 1.7%, respectively. Similarly, our policy to deal with ***Push*** may cause argument width underestimation at the callees, and the likelihood is about 0.2%. This raises an interesting question whether it is possible for CFI policies recovered from binary executables to be more accurate and approach the accuracy of source-based solutions; we discuss this in Section VI-A.

For ***Nor2Var***, the likelihood of misidentifying normal functions to variadic for unoptimized binaries is reduced from 3.3% to 1.2%, with that for optimized binaries dropped to 0.1%.

Since we only apply the policy for ***Temp*** to basic blocks where indirect calls are located, there can be overestimations if the argument registers storing temporary values are in other predecessors. The same policy also helps identify the number of arguments for indirect calls in wrapper functions as shown in the case of ***Wrapper*** in Figure 2 — the likelihood of overestimation on the number of arguments is reduced from 11.5% to 5.4% for C applications compiled by `gcc -O0`.

G. Potential revisions to deal with other complications

To handle ***Var2Nor***, we could revise our policy on identifying variadic functions to find the argument register with the highest index i that is moved onto the stack. However, this will result in (potentially unnecessary) checking of registers at a smaller index, and lead to substantially higher overhead in the processing. Since we only observe one variadic function

```
1 5a0d32: xor    %esi,%esi
2 5a0d34: xor    %edx,%edx
3 5a0d36: mov    %rbp,%rdi
4 #struct ref *(*get_refs_list)(struct transport *
      transport, int for_push, const struct argv_array
      *ref_prefixes);
5 5a0d39: callq  *0x10(%rax)
```

Listing 9: Immediate zero and NULL as arguments

(`bfd_set_error` in `Binutils`) being misidentified as a normal function and causing overestimation on the number of arguments in our large test suite, we do not suggest enforcing this policy.

Similarly for ***Unmodified***, we could perform backward analysis from the indirect caller until another indirect call is encountered. We do not enforce this policy because there is only one application in our test suite that has this problem (with only two indirect calls), and this policy could result in a large number of overestimation on the number of arguments at indirect callers.

VI. Discussions and Security Implications

In this section, we first discuss an interesting question whether policies recovered from binary executables could approach the accuracy of source-based solutions, and then further evaluate the security implications of having inaccurate CFI policies.

A. Comparison with Source-Level Solutions

Section V-F shows that even our improved policy inevitably results in some over- and underestimation, which raises an interesting question whether it is possible to further improve the policies so that their accuracy approaches that of source-level solutions. Here we present three scenarios where a compiler makes the task of accurately recovering function signatures undecidable, and therefore show that binary-level techniques can never achieve the accuracy of source-based solutions.

1) Immediate value zero vs. NULL pointer: A simple example demonstrating the limitation of binary analysis in this context is the differentiation between an immediate value zero and the NULL pointer. Line 4 of Listing 9 shows a callee function with the second and third arguments being integer and pointer type, respectively, while Line 1 – 2 show the caller preparation with identical instructions for these two arguments. It clearly demonstrates that binary analysis is unable to distinguish the two cases and would have to make approximations in recovering the caller signature.

2) Arguments unused: Another scenario arises in the case of unused arguments at the callee (corresponding to complication case ***Unread***), where binary analysis cannot differentiate

- Listing 10a: a callee function with an argument passed in but the argument is not used; and
- Listing 10b: a callee function without arguments.

Binary analysis would not be able to differentiate the two cases as observations on their parameter-passing registers are identical.

```
1 bfd_plugin_core_file_failing_signal (bfd *abfd )
2 482000: push   %rax
3 482001: mov    $0x4dc9e1,%edi
4 482006: mov    $0x1ac,%esi
5 48200b: callq  405230 <bfd_assert>
```

a: Argument passed in but not used

```
1 void bfd_section_already_linked_table_free ()
2 48aa60: mov    $0x7172f8,%edi
3 48aa65: jmpq   406860 <bfd_hash_table_free>
```

b: No argument

Listing 10: Function argument unused

```
1 51e199:mov  %eax,%esi
2 51e19b:test %r15,%r15
3 51e19e:je   51e1ad
4 51e1a0:lea  0xe0(%rsp),%rdi
5 #(fptr_T)(func_one(&cc, c));
6
7 51e1a8: callq *%r15
```

a: %esi used to pass argument

```
1 43ae62: mov   %ebp,%esi
2 43ae64: test  %rax,%rax
3 43ae67: je    43ae6f
4 43ae69: mov   %ebp,%edi
5 #get_elf_backend_data(abfd)
      ->obj_attrs_order(i);
6 43ae6b: callq *%rax
```

b: %esi used to store temporary

Listing 11: Example of argument register usage

3) Registers overloading: Registers are used for passing arguments as well as any other general purposes (corresponding to complication case ***Temp***), and binary analysis usually cannot distinguish the two cases. Listing 11 shows two indirect callers with

- Listing 11a: a caller that uses `%esi` to pass the second argument to callee.
- Listing 11b: a caller that uses `%esi` to store a temporary value.

Again, binary analysis would not be able to tell apart these two cases and an approximation has to be made in extracting function signatures.

We stress that this is not an exhaustive list of cases where binary analysis may fail, but the three scenarios identified are specific to funciton signature recovery where compiler optimization makes binary analysis *undecidable*.

B. Security Implication with Imprecise Function Signature Recovered

The undecidability in binary analysis results in inevitable errors in function signature recovery from (optimized) binary executables. An immediate question, therefore, is on the extent to which such errors impact security applications. In this subsection, we evaluate this security implication from two perspectives.

a) Imprecision on the set of callees allowed: Our first evaluation focuses on the number of callees allowed in a CFI enforcement, and here we consider six solutions:

- AT [36]: A binary-level solution that allows indirect callers to target any "Address-Taken" functions;
- TypeArmor [32]: A binary-level solution with function signatures capturing the number of arguments;

Table I: Number of callees allowed by different policies

		Opt	AT	TypeArmor	τCFI	Improved	IFCC	LLVM-CFI
clang	C	O0	543	412	290	246	114	7
		O1	540	446	242	213	124	8
		O2	394	318	147	147	93	7
		O3	380	300	130	120	99	8
	C++	O0	3,379	2,734	2,343	2,186	1052	37
		O1	3,290	2,631	1,879	1,805	998	35
		O2	702	552	304	270	251	44
		O3	710	543	296	284	247	44
gcc	C	O0	546	499	336	257		
		O1	446	373	272	239		
		O2	418	318	147	147		
		O3	406	332	231	200		
	C++	O0	4,505	3,920	3,278	3,219		
		O1	686	498	314	301		
		O2	698	477	294	281		
		O3	656	527	315	299		
Geomean			767	612	395	353	232	19

- τCFI [23]: A binary-level solution with function signatures capturing the number of arguments and width of arguments;
- Our improved policy: A binary-level solution with function signatures capturing the number of arguments and width of arguments, targeting optimized binaries; and
- IFCC [31]: A (relatively old) source-level solution with function signatures capturing the number of arguments; in LLVM-3.4.
- LLVM-CFI[8]: A (latest) source-level solution with more precise function signatures (the number of arguments and their primitive types, function return type) captured; in LLVM-10.0.

Table I shows the median of the number of callees allowed for each indirect caller for the 1,344 applications in our test suite under different policies. We can see that compared to AT, TypeArmor, τCFI, and our improved policies reduce the number of legal control-transfer targets by about 20%, 49%, and 54%, respectively, while none of the binary-level solutions could achieve precision of source-level techniques. In particular, LLVM-CFI achieves much better accuracy because it uses finer-grained types of arguments — char* and const char*, struct A* and struct B* are considered different types — which cannot be differentiated at binary level.

b) Effectiveness in allowing/disallowing COOP gadgets: With Table I showing the number of mistakes each solution makes, we next evaluate the extent to which these mistakes result in initial COOP gadgets an attacker could use to construct code-reuse attacks. This time, we only focus on τCFI and our improved policy as they run relatively close in the previous evaluation. We use the same heuristics proposed in the corresponding papers to find potential Main-Loop Gadgets (ML-G) [29] and RECursive Gadgets (REC-G) [8] for all C++ applications in our test suite. Table II shows the total number of such gadgets as well as the number of such gadgets whose

[8]https://clang.llvm.org/docs/ControlFlowIntegrity.html

Table II: Potential ML-G and REC-G gadgets

	Opt	ML-G			REC-G		
		icall	τCFI	Improved	icall	τCFI	Improved
clang	O0	93	53	64	73	41	45
	O1	58	50	50	56	44	44
	O2	70	46	52	60	41	44
	O3	70	42	53	49	35	39
gcc	O0	96	50	68	71	32	46
	O1	98	71	80	74	50	56
	O2	113	100	103	33	21	30
	O3	106	79	84	22	15	17
Geomean		83	56	65	58	37	43

function signatures are correctly identified by τCFI and our improved policy. Bigger numbers indicate better effectiveness of CFI in disallowing the corresponding code-reuse attacks.

As we can see, τCFI correctly identifies 68% and 64% ML- and REC- gadgets, respectively, while our improved policy achieves 78% and 74% effectiveness, respectively. We believe that this evaluation provides a good indicator on the security impact of our improved CFI policies.

c) Severity of each mistake: For each mistake in recovering function signature of the caller, we check how far the mistake is from the ground truth, which also has a direct implication on the amount of flexibility an attacker has when using the corresponding caller to construct an code-reuse attack. Figure 3 shows the result of this evaluation, again, on our test suite of 1,344 applications, with x-axis labels being:

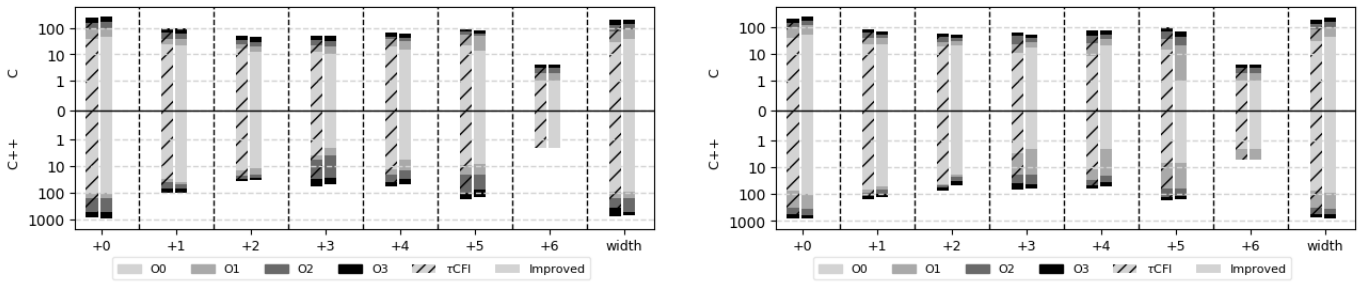
- $+t$: the average number of indirect callers whose number of arguments is overestimated by t; and
- width: the average number of indirect callers whose function signature (number and width of arguments) is correctly recovered.

Besides showing the consistently better results from our improved policy compared to those from τCFI, we also notice that our improved policy performs most significantly better on "+5", which means our improved policies manage to correct a larger number of more severe mistakes made by τCFI.

VII. Related Work and Limitations

A. Control-Flow Integrity

Control-Flow Integrity forces control-flow transfers in the program to follow policy presented by the CFG. Due to the difficulty in accurately recovering the CFG from the binary, most approaches enforce a coarse-grained policy by conservatively considering all functions as potential targets of an indirect caller. Usually, they mark valid targets of indirect control transfers with unique identifiers (ID) and then insert ID-checks into the program before each indirect branch transfer. An indirect branch is allowed to jump to any destination with the correct ID. For example, CFIMon [33] makes use of static analysis and online training to get valid targets for return, indirect call, and indirect jump instructions. Branch Trace Store (BTS) is used to collect in-flight control transfers to perform CFI check. BinCFI [36] uses two IDs for all indirect branch transfers: one for return and indirect jump instructions, and the other for indirect call instructions. All



(a) Applications compiled by Clang

(b) Applications compiled by GCC

Figure 3: Amount of flexibility of code-reuse attacks in each mistake in function signature recovery of indirect callers

indirect branches are instrumented to jump to the corresponding address translation routine that determines the targets of the transfers. CCFIR [35] implements a 3-ID approach which extended the 2-ID approach by further separating returns into sensitive and non-sensitive functions. All control-flow targets of indirect branches are collected and randomly allocated on a springboard section, and indirect branches are only allowed to use control flow targets contained in the springboard section. These approaches allow an indirect call to target any function, which makes them vulnerable to many state-of-the-art code-reuse attacks [4], [9], [11], [13], [29].

Fine-grained CFI approaches based on function signature matching are proposed and they rely on the availability of source code to obtain function signature. MCFI [25] and πCFI [26] instrument each indirect branch transfer during compile time to consult tables that store legitimate targets. These tables are updated when modules are dynamically loaded by making use of the auxiliary type information obtained at compilation. Forwarding CFI [31] protects binaries by inserting checks before all forward edge control flow transfers to check whether the function signature (the number of arguments) is correct. Cryptographically enforced CFI [20] enforces another form of fine-grained CFI by adding a message authentication code (MAC) that is computed with type information to control flow elements, which prevents the usage of unintended control-flow transfers in the CFG.

B. Function Signature Recovery

Besides TypeArmor [32], liveness analysis and heuristic methods based on calling conventions and idioms were used to recover function signatures. EIWazeer et al. [10] apply liveness analysis to recover arguments, variables, and their types. TIE [17] infers variable types in binaries through formulating the usage of different data types. Caballero et al. [3] make use of dynamic liveness analysis to recover function arguments for execution traces. Since it is a dynamic analysis, it cannot guarantee the full coverage of unused arguments during an execution trace. Recently, Zeng et al. [34] propose to perform type inference based on debugging information generated by the compiler so that a high-precision CFG can be constructed to help CFI enforcement. Another direction is to make use of machine learning approaches to recover function signatures. For example, EKLAVYA [6] uses a three layers Recurrent Neural Network to learn the number and types of arguments from disassembled binary code.

C. Limitations

Currently, we only focus on function signature recovery for integer arguments with floating-point arguments passed via `XMM` registers not taken into consideration. This may give the attacker more chances to find valid gadgets that can be used to construct code-reuse attacks. Current CFI policies based on argument width cannot be directly used when floating-point arguments are analyzed since static analysis cannot reveal the order between integer and floating-point arguments. We leave it as our future work to extend our static analysis to include floating-point registers.

New optimization strategies employed by compilers makes function signature recovery more difficult and the analysis engines need to be continuously updated so that they can be used to analyze binaries compiled by these new versions of compilers.

VIII. Conclusion

In this paper, we study how compiler optimization impacts function signature recovery implemented TypeArmor and τCFI. Our study shows that compiler optimization has important impact on function signature recovery and potentially results in unmatched function signatures at callees and callers. In order to better deal with these optimizations, a set of improved policies is proposed, with results showing that most complications identified earlier being mitigated.

Acknowledgement

We thank our shepherd, Christo Wilson and the anonymous reviewers for their valuable comments and suggestions that have helped improve our paper. This research has been supported by National Research Foundation (NRF) Singapore under its National Satellite of Excellence in Trustworthy Software Systems program (Award No: NSOE-TSS2019-02).

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009.

[2] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[3] Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. Technical report, California Univ Berkeley Dept of Electrical Engineering and Computer Science, 2009.

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

[5] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[6] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. Neural nets can learn function type signatures from binaries. In *Proceedings of the 26th USENIX Security Symposium*, pages 99–116, 2017.

[7] DWARF Debugging Information Format Committee et al. Dwarf debugging information format, version 4. *Free Standards Group*, 2010.

[8] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 243–255, 2015.

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

[10] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 51–60, 2013.

[11] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 901–913. ACM, 2015.

[12] Ghidra. The ghidra decompiler. https://ghidra-sre.org/, 2019.

[13] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, pages 575–589. IEEE, 2014.

[14] INC INTEL. Intel® 64 and ia-32 architectures software developers manual. 2018.

[15] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*. USENIX Association, 2004.

[16] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004.

[17] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[18] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P Lopes. Taming undefined behavior in llvm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 633–647. ACM, 2017.

[19] Christian Lindig. Random testing of c calling conventions. In *Proceedings of the 6th international symposium on Automated analysis-driven debugging*, pages 3–12. ACM, 2005.

[20] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 941–951. ACM, 2015.

[21] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System v application binary interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2014.

[22] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. Safelnit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, pages 1–15, 2017.

[23] Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. τcfi: Type-assisted control flow integrity for x86-64 binaries. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 423–444. Springer, 2018.

[24] Nergal. The advanced return-into-lib(c) exploits. http://phrack.org/issues/58/4.html, 2001.

[25] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*, pages 577–587. ACM, 2014.

[26] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 914–926. ACM, 2015.

[27] Nathan Rosenblum, Barton P Miller, and Xiaojin Zhu. Recovering the toolchain provenance of binary code. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 100–110. ACM, 2011.

[28] Nathan E Rosenblum, Barton P Miller, and Xiaojin Zhu. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 21–28. ACM, 2010.

[29] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pages 745–762. IEEE, 2015.

[30] Hovav Shacham et al. 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*, pages 552–561. ACM, 2007.

[31] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*, pages 941–955, 2014.

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

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

[34] Dongrui Zeng and Gang Tan. From debugging-information based binary-level type inference to cfg generation. In *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy*, pages 366–376. ACM, 2018.

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

[36] Mingwei Zhang and R Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*, pages 337–352, 2013.

APPENDIX

A. Example of complication *lea*

Here we show an example where instruction `lea` takes a promoted operant. In Listing 12, the state of the second argument is $\dot{rw}_2(64)$; however, the ground truth is a 32-bit parameter (unsigned int).

```
1 bfd_check_overflow (enum complain_overflow how,
2  unsigned int bitsize, unsigned int rightshift, unsigned int
       addrsize, bfd_vma relocation)
3
4 000000000048ca60 <bfd_check_overflow>:
5 48ca60: mov    %ecx,%eax
6 48ca62: mov    %edx,%r9d
7 48ca65: lea    -0x1(%rsi),%ecx
8 48ca68: mov    $0xfffffffffffffffe,%rdx
```

Listing 12: Promoted operand of instruction lea

B. Additional examples of missing argument-writing instructions at callers

Listing 13 shows indirect calls in a wrapper function. Since there is no direct caller for function `bfd_elf64_swap_dyn_in`, TypeArmor and τCFI determine that $P^{OB}_{ER-0x416845} =< w_1(64), w_2(64), w_3(64), w_4(64), w_5(64), w_6(64) >$, which results in an overestimation on the number of arguments while $|P^{GT}_{ER-0x416845}| = 1$.

```
1 0000000000416830 <bfd_elf64_swap_dyn_in>:
2 416830: push   %r15
3 ......
4 416835: mov    %rdx,%r14
5 416838: mov    %rsi,%r15
6 41683b: mov    %rdi,%rbx
7 41683e: mov    0x8(%rdi),%rax
8 416842: mov    %rsi,%rdi
9 416845: callq  *0x68(%rax)
```

Listing 13: An indirect call in a wrapper function

Listing 14 shows that $P^{GT}_{ER-0x1aae2c} =< w_1(64), w_2(64), w_3(64), \hat{w}_4, \hat{w}_5, \hat{w}_6 >$. However, $P^{OB}_{ER-0x1aae2c} =< w_1(64), \hat{w}_2, w_3(64), \hat{w}_4, \hat{w}_5, \hat{w}_6 >$ and $|P^{OB}_{ER-0x1aae2c}| = 1$ since the value of `%rsi` is not changed by the function at `0x1a95f0`, and the compiler does not reset it explicitly.

```
1 1aae0a: mov    0xb38(%r13,%r14,1),%rdi
2 1aae12: mov    %rbp,%rsi
3 1aae15: callq  1a95f0
4 1aae1a: mov    (%rsp),%rax
5 1aae1e: lea    (%rax,%r14,1),%rdx
6 1aae22: mov    (%rbx),%rax
7 1aae25: mov    0xb8(%rax),%rdi
8 #call funcs->create( cffsize->face->memory, &priv, &
      internal->subfonts[i - 1] )
9 1aae2c: callq  *(%r12)
```

Listing 14: Arguments not modified between two calls

C. Exmample of a NULL pointer as an argument

The example with a NULL pointer being an argument is shown in Listing 15. According to the ground truth at Line 5, the second argument should be a pointer; but a NULL pointer is passed at the caller, and the compiler uses `xor` to prepare for it.

*D. Argument (width) promotion at both callees and callers (**Prom**)*

There are other argument (width) promotions at both callees and callers that would not result in inaccuracies in matching

```
1 57f50e: test %rbp,%rbp
2 57f511: je 57f531
3 57f513: mov 0x333a46(%rip),%rdi
4 57f51a: xor %esi,%esi
5 #(*advertise)(*r, *);
6 57f51c: callq *0x8(%rbp)
```

Listing 15: A NULL pointer as an argument

function callees with callers since the argument promotion happens in a matching manner. This refers to promotions of types smaller than the native type of the target platform's Arithmetic Logic Unit (ALU) to make arithmetic and logical operations possible or more efficient. C and C++ perform such promotions for objects of boolean, character, wide character, enumeration, and short integer types. As shown in Listing 16, the type of the third argument is unsigned char (8-bits) as shown at Line 1, but the analysis engine would determine its state being $rw_3(32)$ due to the promotion performed by the compiler.

```
1 static bfd_boolean add_line_info (struct
      line_info_table *table, bfd_vma address, unsigned
       char op_index, char *filename,      unsigned int
       line, unsigned int column, unsigned int
      discriminator, int end_sequence)
2
3 000000000044c2d0 <add_line_info>:
4 44c2d0: push   %rbp
5 ......
6 44c2e7: mov    %edx,%r12d
7 44c2ea: mov    %rsi,%r13
8 44c2ed: mov    %rdi,%rax
9 44c2f0: mov    (%rdi),%rdi
10 44c2f3: mov   %rax,0x8(%rsp)
11 44c2f8: mov   0x30(%rax),%rax
12 44c2fc: mov   %rax,0x10(%rsp)
13 44c301: mov   $0x28,%esi
14 44c306: callq 408a80 <bfd_alloc>
```

Listing 16: Promotion of small integral types

E. Summary of complications at callees and callers

Table III summaries the complications at both callees and callers with the last column indicating the impact that these cases can cause.

F. Sizes of binary executables in our test suite

Table IV shows the sizes of the binary executables in our test suite under various optimization flags for both C and C++ programs. Note that the C++ programs are typically larger than the C programs.

G. Github applications in our test suite

Table V shows the Github applications we include in our test suite. We typically choose those with a large number of stars.

H. Number of arguments in functions in our test suite

Table VI shows the percentage of functions with specific number of arguments, as well as the geometric mean of the number of variadic functions in each application.

Table III: Summary of complications introduced by compiler optimization

Site	Category	Complication	Impact
Callee	Misidentifying variadic functions	Normal to variadic (***Nor2Var***)	$\|P_{EE}^{OB}\| < \|P_{EE}^{GT}\|$
		Variadic to Normal (***Var2Nor***)	$\|P_{EE}^{OB}\| > \|P_{EE}^{GT}\|$
		Back-to-back condition unreliable (***VarOver***)	$\|P_{EE}^{OB}\| > \|P_{EE}^{GT}\|$
	Missing argument reading instructions	Arguments are not used by a function (***Unread***)	$\|P_{EE}^{OB}\| < \|P_{EE}^{GT}\|$ $\|s_i^{EE}\| < \|s_i^{EE,GT}\|$
	Misidentifying %rdx as an argument	Reading the higher 64 bits of a return value (***rdx***)	$\|P_{EE}^{OB}\| > \|P_{EE}^{GT}\|$
	Argument (width) promotion	Arguments are pushed onto the stack (***Push***)	$\|s_i^{EE}\| > \|s_i^{EE,GT}\|$
		Default width of the operand of certain instructions is 64-bit (***lea***)	$\|s_i^{EE}\| > \|s_i^{EE,GT}\|$
Caller	Missing argument writing instructions	Higher 64 bits of a return value as the third argument (***Ret***)	$\|P_{ER}^{OB}\| < \|P_{ER}^{GT}\|$
		Uninitialized variables as arguments (***Uninit***)	$\|P_{ER}^{OB}\| < \|P_{ER}^{GT}\|$
		Indirect calls in wrapper functions (***Wrapper***)	$\|P_{ER}^{OB}\| > \|P_{ER}^{GT}\|$
		Argument values not modified between two calls (***Unmodified***)	$\|P_{ER}^{OB}\| < \|P_{ER}^{GT}\|$
	Registers storing temporary values	Argument registers are used to store temporary values (***Temp***)	$\|P_{ER}^{OB}\| > \|P_{ER}^{GT}\|$
	Argument (width) demotion	Argumets are constant whose sizes are up to 32-bit (***Imm***)	$\|s_i^{ER}\| < \|s_i^{ER,GT}\|$
		Argument are pointers pointing to data and text sections (***Pointer***)	$\|s_i^{ER}\| < \|s_i^{ER,GT}\|$
		Arguments are NULL pointers (***Null***)	$\|s_i^{ER}\| < \|s_i^{ER,GT}\|$
Both	Small integral type promotion	Small integral types are promoted to native types (***Prom***)	$\|s_i^{EE}\| > \|s_i^{EE,GT}\|$ $\|s_i^{ER}\| > \|s_i^{ER,GT}\|$

Table IV: Sizes of the binary executables in our test suite

Language	Opt	Size (MB)					
		clang			gcc		
		min	median	max	min	median	max
C	O0	0.07	0.69	44.75	0.08	0.68	44.72
	O1	0.07	0.71	45.61	0.12	0.98	50.52
	O2	0.08	0.84	50.09	0.11	1.02	51.79
	O3	0.08	0.84	48.95	0.13	1.55	54.30
C++	O0	0.11	7.51	65.77	0.12	14.60	73.22
	O1	0.11	7.22	68.82	0.17	10.32	99.95 9
	O2	0.13	6.31	65.70	0.18	16.96	105.50
	O3	0.13	6.15	66.79	0.19	17.12	109.83

Table V: Github applications in our test suite

App	Language	description
git	C	Distributed version control system
darknet	C	An open source neural network framework
netdata	C	A real-time performance monitoring
redis	C	An in-memory database
sqlite	C	SQL database engine
vim	C	UNIX text editor
gnupg	C	Complete implementation of the OpenPGP standard
openssl	C	TLS/SSL and crypto library
mupdf	C & C++	A lightweight PDF, XPS, and E-book viewer
vorbis	C	A general purpose audio and music encoding format
aria2c	C++	A lightweight multi-protocol download utility
cppcheck	C++	Static analysis of C/C++ code
hpx	C++	C++ Standard Library for Parallelism and Concurrency
xpdf	C++	A PDF viewer and toolkit

I. Argument types of functions in our test suite

Table VII shows the percentage of functions having a specific type as its arguments.

Table VI: Number of arguments of functions in our test suite

Language	Opt	Number of Arguments (%)							# variadic
		0	1	2	3	4	5	6	
C	O0	6.92	29.35	29.73	17.46	7.47	4.33	1.77	8.45
	O1	6.01	28.64	29.87	17.32	7.73	4.71	1.95	
	O2	6.78	28.05	27.85	18.11	8.22	4.92	1.88	
	O3	5.99	26.52	29.04	18.20	8.55	5.17	2.11	
C++	O0	4.31	47.84	26.78	12.97	3.64	2.47	0.64	2.43
	O1	4.44	46.06	27.76	13.34	3.80	2.54	0.67	
	O2	3.09	45.27	20.77	12.58	7.09	5.48	1.87	
	O3	3.13	45.84	20.88	12.45	6.95	5.09	1.86	

J. Number of various types of functions and indirect calls in our test suite

Table VIII shows the number of various types of functions and indirect calls in our test suite.

K. Likelihood that indirect calls in C programs use immediate values as arguments

Table IX shows the likelihood that indirect calls use immediate values as arguments for different reasons.

L. Analysis of variadic function in Binutils

Table X shows the details about the analysis in identifying a variadic function by TypeArmor and our new policy.

M. Our improved policy for identifying variadic functions in unoptimized binaries

Definition A.1. Callee function f is a variadic function iff $\forall i \in \{5, 4, 3, 2, 1\}$,

- $|@_{i+1} - @_i| = 8$; and

Table VII: Argument types of functions in our test suite

Type	Opt	Arg for C (%)						Arg for C++ (%)					
		1st	2nd	3rd	4th	5th	6th	1st	2nd	3rd	4th	5th	6th
8-bits	O0	0.186	0.374	0.123	0.261	0.338	0.353	0.024	0.318	1.517	0.271	0.831	0.944
	O1	0.140	0.307	0.225	0.286	0.535	0.555	0.080	0.563	1.596	0.384	0.431	0.546
	O2	0.106	0.242	0.301	0.225	0.753	0.779	0.024	0.310	1.475	0.258	0.786	0.920
	O3	0.103	0.252	0.261	0.291	0.598	0.609	0.024	0.319	1.517	0.271	0.831	0.944
16-bits	O0	0.091	0.188	0.127	0.135	0.169	0.294	0.003	0.038	0.224	0.242	0.444	0.498
	O1	0.108	0.235	0.168	0.119	0.134	0.266	0.043	0.075	0.112	0.216	0.392	0.307
	O2	0.097	0.205	0.100	0.164	0.143	0.325	0.003	0.037	0.215	0.231	0.420	0.486
	O3	0.113	0.243	0.149	0.134	0.0.136	0.292	0.003	0.038	0.224	0.242	0.444	0.499
32-bits	O0	9.382	19.582	25.330	29.654	33.498	28.794	0.823	9.663	19.566	26.018	29.271	17.364
	O1	8.549	19.747	25.292	30.307	37.600	32.971	0.554	4.602	15.191	15.572	21.027	16.804
	O2	8.305	18.407	24.213	28.380	32.706	25.609	0.809	9.438	19.095	24.890	27.834	16.979
	O3	7.480	19.233	24.910	30.479	38.844	33.333	0.823	9.663	19.566	26.018	29.271	17.364
64-bits	O0	2.314	7.248	11.359	10.373	10.853	10.000	0.144	5.131	10.820	17.138	13.026	10.472
	O1	1.971	6.166	10.929	9.175	9.005	7.744	0.834	7.540	14.866	11.212	7.373	6.294
	O2	2.240	6.994	12.193	11.079	10.771	9.834	0.144	4.956	10.920	17.207	13.042	10.312
	O3	1.938	5.952	10.653	9.309	8.688	7.572	0.144	5.131	10.820	17.138	13.026	10.472
ptr	O0	88.023	72.633	62.610	59.425	54.988	60.441	98.031	84.445	67.557	54.700	56.154	70.205
	O1	89.223	73.398	62.936	59.839	52.482	58.221	98.280	86.986	67.613	70.732	70.199	75.771
	O2	89.240	73.958	62.766	59.793	55.448	63.389	98.056	84.866	67.992	55.853	57.659	70.799
	O3	90.355	74.140	63.576	59.482	51.475	57.950	98.031	84.445	67.557	54.700	56.154	70.205

Table X: Analysis of the non-variadic function in Binutils

Line Number	Operation	TypeArmor	Our improved policy
4	Move %r9 to stack 0x40(%rsp)	%r9 is a variadic argument	May be a variadic argument
5	Move %r8 to stack 0x10(%rsp)	%r8 is a variadic argument	Non-consecutive stack addresses; not a variadic argument
11	0x40(%rsp) is read not overwritten		Not a variadic argument
Conclusion		Variadic function with 4 arguments	Normal function with 6 arguments

Table IX: Likelihood that indirect calls in C programs use immediate values as arguments

Compiler	Opt	Non-inline	Inline	Loop-unroll	Func-copy
clang	O0	24	0	0	3
	O1	23	0	0	6
	O2	13	52	0	3
	O3	13	49	45	4
gcc	O0	81 (25)	0	0	3
	O1	75 (23)	59	0	5
	O2	26	28	0	3
	O3	22	33	3	3

Numbers in brackets correspond to functions that pass the value 0 to an argument register.
"Func copy" refers to multiple copies of the same function called from different modules.
Results shown are likelihood results multiplied by 1,000, rounded to the nearest integer.

Table VIII: Number of functions and indirect calls

		Opt	#func	#normal func	#variadic	#icalls
clang	C	O0	543	486	21	64
		O1	540	495	19	60
		O2	394	346	19	85
		O3	380	352	19	93
	C++	O0	3,379	3,229	15	121
		O1	3,290	3,085	13	74
		O2	702	652	13	439
		O3	710	640	13	452
gcc	C	O0	546	483	24	70
		O1	446	420	19	69
		O2	418	386	21	67
		O3	406	370	22	83
	C++	O0	4,505	4,113	22	152
		O1	686	608	13	336
		O2	698	613	13	312
		O3	656	606	13	299

- $s_{i+1}^{EE} = r\dot{w}2s_{i+1}(64)$ and $s_i^{EE} = r\dot{w}2s_i(64)$.

with $|P_{EE-f}^{OB}|$ being the maximal i violating the above. Otherwise, f is a normal function with 6 arguments.