TypeSqueezer: When Static Recovery of Function Signatures for Binary Executables Meets Dynamic Analysis

Ziyi Lin
Xidian University
Xi'an, China

Jinku Li
Xidian University
Xi'an, China

Bowen Li
Xidian University
Xi'an, China

Haoyu Ma*
Zhejiang Lab
Hangzhou, China

Debin Gao
Singapore Management University
Singapore, Singapore

Jianfeng Ma
Xidian University
Xi'an, China

ABSTRACT

Control-Flow Integrity (CFI) is considered a promising solution in thwarting advanced code-reuse attacks. While the problem of backward-edge protection in CFI is nearly closed, effective forward-edge protection is still a major challenge. The keystone of protecting the forward edge is to resolve indirect call targets, which although can be done quite accurately using type-based solutions given the program source code, it faces difficulties when carried out at the binary level. Since the actual type information is unavailable in COTS binaries, type-based indirect call target matching typically resorts to approximate function signatures inferred using the arity and argument width of indirect callsites and calltargets. Doing so with static analysis, therefore, forces the existing solutions to assume the arity/width boundaries in a too-permissive way to defeat sophisticated attacks.

In this paper, we propose a novel hybrid approach to recover fine-grained function signatures at the binary level, called TypeSqueezer. By observing program behaviors dynamically, TypeSqueezer combines the static analysis results on indirect callsites and calltargets together, so that both the lower and the upper bounds of their arity/width can be computed according to a philosophy similar to the squeeze theorem. Moreover, the introduction of dynamic analysis also enables TypeSqueezer to approximate the actual type of function arguments instead of only representing them using their widths. These together allow TypeSqueezer to significantly refine the capability of indirect call target resolving, and generate the approximate CFGs with better accuracy. We have evaluated TypeSqueezer on the SPEC CPU2006 benchmarks as well as several real-world applications. The experimental results suggest that TypeSqueezer achieves higher type-matching precision compared to existing binary-level type-based solutions. Moreover, we also discuss the intrinsic limitations of static analysis and show that it is not enough in defeating certain type of practical attacks; while on the other hand, the same attacks can be successfully thwarted with the hybrid analysis result of our approach.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; **Operating systems security**.

KEYWORDS

Control-flow integrity; Type inference; Binary executables

ACM Reference Format:
Ziyi Lin, Jinku Li, Bowen Li, Haoyu Ma, Debin Gao, and Jianfeng Ma. 2023. TypeSqueezer: When Static Recovery of Function Signatures for Binary Executables Meets Dynamic Analysis. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623214

1 INTRODUCTION

With advanced code-reuse attacks [4, 5, 9, 12, 18, 34, 35, 40] being a major threat to today's software systems, control-flow integrity (CFI) [6] had become one of the most promising countermeasures. While an ideal CFI should be able to enforce the exact control-flow graph (CFG) of a program so that each control transfer is only allowed to target its intended destination(s) [1], it is hard to achieve such an ideal security goal without source code because extracting the CFG to such degree of accuracy is an undecidable problem. The alternative (and more practical) strategy is to enforce an approximation of the program's CFG, but should this approximation be established too loosely, the resulting CFI defense may be bypassed by carefully orchestrated attacks, e.g., Counterfeit Object-Oriented Programming (COOP) [12, 34], Control JuJutsu [13], Control-Flow Bending (CFB) [8], etc. Therefore, how to generate precise CFGs for programs is a key question for binary-level CFI schemes.

Existing binary-level CFI solutions mainly guard against forward-edge violations by enforcing certain CFI policies on indirect callsites (e.g., virtual calls, indirect calls, etc., henceforth we refer these as *icalls* for short), so that they are only allowed to target a limited set of possible destinations. This begins with bin-CFI [46], which scans the subject binary to find all address-taken functions (i.e., functions that have their addresses assigned to function pointers, henceforth *AT functions* for short) and restricts icalls from targeting any functions other than them. However, a CFI policy of this

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '23, November 26–30, 2023, Copenhagen, Denmark

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623214

granularity is not enough to defend against aforementioned advanced code-reuse attacks [8, 12, 13, 34]. Therefore, subsequent solutions adopt a variety of strategies to refine their capabilities of CFG inference. For instance, vfGuard [32] and VTint [44] enforce the integrity of icalls derived from C++ virtual calls by restricting their virtual table accesses, but such approaches cannot be applied to all icalls in general. BPA [20] uses a block-based points-to analysis that scales to large 32-bit binary executables to resolve targets of icalls, but it suffers from significant overhead which makes it difficult to extend to 64-bit binaries.

Enforcing type-based matching for CFI is a popular (and probably the most generalized) way of inferring correlations between icalls and AT functions. Representative works like TypeArmor [37] and τCFI [28] leverage static analysis to infer type-related features of icalls and AT functions, such as the number of parameters (i.e., the arity) and the width of arguments, and accordingly they allow icalls to target only the compatible AT functions. Lin et al. [23] further study how compiler optimization mechanisms affect TypeArmor and τCFI, and propose their countermeasure. Fundamentally, both TypeArmor and τCFI are established on top of static binary analysis. Unfortunately, unlike the case with program source code, COTS (Commercial Off-The-Shelf) binaries provide almost no supportive information for conducting type-based matching. Specifically, two major challenges can be identified:

- First, for any icall or AT function, static analysis can only assume a possible range of its arity. Worse still, by evaluating loading/storing operations on a specific set of registers used for parameter passing, *static analysis can only assume the upper bound of the arity of an icall.* While *given an AT function, it can only infer the lower bound of its arity.*
- Second, it is an undecidable task to recover the precise type information from binary executables, and it is too permissive to infer parameter types with the width of arguments because *distinguishing between values and address references is not always feasible just by looking at the argument widths.*

As a result, current binary-level CFI schemes leveraging type-based matching are in fact less effective than expected. To give an example, assuming an icall which prepares 6 arguments (System V ABI [27] uses at most 6 registers to pass integer arguments), TypeArmor can only conclude that this icall prepares *at most* 6 arguments and would therefore allow it to target *any* AT functions. This is because any AT function consumes 0 to 6 arguments from registers, and is therefore considered as compatible. Further, in the case that all the 6 arguments of the mentioned icall are address references (i.e., 64-bit long in the case of x86_64), then even τCFI or Lin et al.'s work [23] would allow it to target any AT functions because a 64-bit argument would be considered as compatible with an argument in arbitrary width at the calltarget. Even worse, address reference is the most common type of arguments in real-world programs (see §4.1.3 for a detailed evaluation). Though static analysis may also distinguish between variables and address references, e.g., by investigating the instructions that process the corresponding arguments, it usually requires sophisticated binary-level alias analysis to do so, which is therefore of high complexity and overhead. For the same reason, it is also difficult to statically determine the exact structure of an object pointed by an address reference.

Noticing that *binary-level CFI is currently encumbered by intrinsic limitations of static program analysis*, this paper proposes a novel hybrid style binary analysis approach, called TypeSqueezer, to provide fine-grained function signature recovery that serves as the basis for binary-level CFI (and to improve their effectiveness). We take one more step to explore the potential (and perhaps the limit) of type-based matching applied on binary executables while further raising the bar for code-reuse attacks. On top of the strategies adopted by existing schemes, TypeSqueezer further enriches the inferred prototype signature of icalls and AT functions by carrying out a tailor-made combination of static/dynamic analysis, which inspects the actual destination of direct/indirect calls as well as runtime register and memory states related to the observed invocations. Specifically, TypeSqueezer makes progress mainly in three aspects compared to existing binary-level type-based solutions:

- First, according to how icalls and AT functions are paired at runtime, TypeSqueezer works with a philosophy similar to the *squeeze theorem* (hence the name we give it) and is able to determine *both the lower and upper bound* of their possible arities and argument widths.
- Second, by statically searching for AT functions that are also invoked by direct calls, TypeSqueezer is able to integrate assumed prototype signatures of direct calls into the inference of icalls and/or AT functions and further polish the results.
- Third, by dynamically inspecting content within registers when an icall or AT function is reached, TypeSqueezer can tell *whether a potential parameter of the subject is a value or an address reference* without involving complicated alias analysis. By further looking into the memory locations referenced by registers, TypeSqueezer can even *distinguish between address references that index values and addresses.*

These together allow TypeSqueezer to generate more accurate signatures for both icalls and AT functions in subject binaries. Consequently, it is able to enforce indirect control flows with rules that fit more closely with the supposed CFG.

To validate our approach, we implement a proof-of-concept prototype of TypeSqueezer on top of Intel Pin [26] and Dyninst [3], a pair of binary instrumentation frameworks that enable customized static and dynamic analysis of COTS executables. We evaluate the prototype with both SPEC CPU2006 and several real-world applications. The experimental results suggest that TypeSqueezer significantly reduces the average indirect call targets (AICT) compared to TypeArmor [37], τCFI [28], and the scheme of Lin et al.'s [23]. Furthermore, with TypeSqueezer's analysis result, a binary-level CFI would be able to prevent certain targeted attacks that may bypass the aforementioned solutions. To engage the community, we release the source code of our proof-of-concept implementation at https://github.com/XDU-SysSec/TypeSqueezer. In summary, this paper makes the following contributions:

- We propose the first retrofitted type-based matching approach using hybrid static/dynamic binary analysis for forward-edge CFI, which leverages clustering of dependent callers/callees and nested investigation of in-register arguments to make more precise arity and type inferences compared to the state-of-the-art solutions [23, 28, 37].

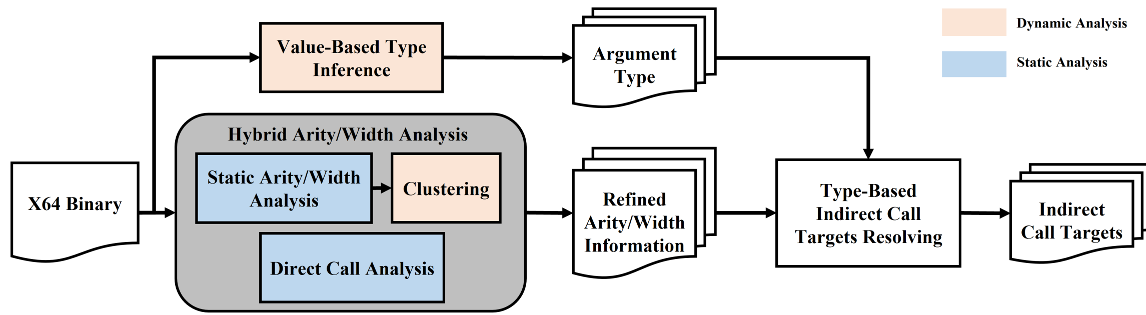

Figure 1: The Overall Workflow of TypeSqueezer

- We implement a proof-of-concept prototype that carries out the proposed type-based matching, namely TypeSqueezer. The experimental results demonstrate that TypeSqueezer outperforms existing solutions built on top of static type-based matching in enforcing forward-edge invariants.
- We show that our improved type-based matching strategy is able to mitigate carefully crafted forward-edge attacks against binary-level CFIs. For example, TypeSqueezer can detect and prohibit a variant of Control JuJutsu attack [13] which cannot be guarded against by prior solutions.

2 SYSTEM DESIGN

In this section, we first present a high-level overview of the proposed TypeSqueezer system, and then introduce design details of the key techniques of our approach. We assume that programs to be analyzed and protected have been built into x86_64 binaries via a compiler that adheres to System V ABI with regards to calling convention, and our approach only pays attention to function parameters passed via registers. In addition, obfuscated or hand-crafted binaries are considered out-of-scope, and so are floating-point arguments that are passed using xmm registers. These assumptions put our work on the same starting line with state-of-the-art systems [23, 28, 37] so that the effectiveness of our approach can be evaluated without introducing unfair advantages.

2.1 System Overview

As mentioned in §1, the accuracy of inferring the arity of both icalls and AT functions using static analysis is intrinsically limited by the missing type-related information which is only available at the source level. Static analysis resorts to reaching-definition analysis on icalls and liveness analysis on AT functions to assume their arities. However, it is difficult to distinguish an actual argument preparation from the storing of a temporary value (both of which can exist in the form of write to argument registers). In addition, it is also impractical to assert that an argument register is not used to pass a parameter just because it is not read in the entire function. As a result, the static analysis would inevitably *overestimate* the arity of icalls and *underestimate* the arity of AT functions. This issue, together with the incapability of statically distinguishing between full-width variables and address references, are the main causes of the inaccuracy of binary CFI schemes using type-based matching.

A key takeaway from the above observations is that *static analysis happens to be over/underestimating the arity of icalls and AT functions, respectively.* This suggests that if those icalls and AT functions can be explicitly paired, the incomplete assumptions on their arities could then compensate for each other and end up improving the overall accuracy of the generated CFG. Such an idea is very close to the philosophy of the *squeeze theorem* in mathematics, which determines the limit of a function with the known limits of two other functions between which the subject function is trapped[1]. Real execution traces obtained using dynamic analysis could provide the exact kind of evidence regarding the dependencies between icalls and AT functions. Moreover, dynamic analysis could also leverage the runtime context to achieve a more comprehensive investigation of the possible type of identified function arguments (rather than just looking at their widths). Considering these advantages, we build TypeSqueezer into a hybrid framework that combines the capabilities of static and dynamic binary analysis to provide more accurate type-based matching for CFI.

Figure 1 illustrates the overall workflow of TypeSqueezer. On top of the existing arity/width analysis adopted by state-of-the-art (Lin et al. [23] in specific), our approach statically analyzes direct callsites within the subject binary to obtain explicit dependencies between them and AT functions. We also run a dynamic simulation to observe actual indirect control transfers. This information is then used to cluster icalls and AT functions into groups according to their dependencies, so that the arity and argument width of each group can be refined to have both an upper and a lower bound (see §2.2). Furthermore, TypeSqueezer performs a nested investigation of the value of assumed arguments for both icalls and AT functions to observe if any valid address within the runtime context may be referenced, resulting in detailed inferencing on the possible type of arguments (see §2.3). The aforementioned analyses together allow TypeSqueezer to generate much richer profiles for the icalls and AT functions when resolving indirect control transfer targets, to further reduce the allowed targets, and consequently recover a more accurate CFG.

2.2 Hybrid Arity/Width Analysis

As described earlier, a key insight regarding indirect control flow in type-based solutions is that *during benign program execution, icalls will always invoke AT functions that have the same function signatures.* Therefore, given a subject binary on which a static arity/width analysis has been performed, TypeSqueezer dynamically

[1]The actual squeeze theorem requires the known limits to be equal, which is not needed in this paper.

Algorithm 1 Clustering

Input: ICG: The indirect call graph, $Variadics$: The set of variadic functions
Output: $CLTs$: The set of clusters

```
1:  CLTs ← ∅
2:  function Cluster(ICG,Variadics)
3:      for each < cs, ct >∈ ICG ∧ ct ∉ Variadics do
4:          if cs and ct do not belong to any cluster then
5:              Add a new cluster{cs,ct} to CLTs
6:          else
7:              Unify cs's cluster and ct's cluster
8:          end if
9:      end for
10: end function
```

observes the control flow between icalls and AT functions to gain the knowledge on which AT functions are actually targeted by specific icalls, and therefrom groups them into a cluster. Once the clustering is complete, the static arity/width inferences on the members of the same cluster become a set of flawed assumptions on the same function prototype. Since the inferences on icalls and AT functions flaw in different ways (as mentioned in §2.1, they are respectively over/underestimated), together they help TypeSqueezer to compute a unanimous (and closer to the optimized) assumption of the arity/width for the cluster.

2.2.1 Dependency-Based Clustering of Icalls and AT Functions. Our approach clusters icalls and AT functions by monitoring its dynamic control flows. When the monitored program executes an indirect call, TypeSqueezer records the addresses of the indirect call and the target function in a `<cs(callsite), ct(calltarget)>` pair, where cs represents an icall and ct indicats an AT function. We refer to the recorded execution traces as the *indirect call graph* (ICG). As illustrated by Algorithm 1, for each `<cs, ct>` pair, if both the callsite and calltarget have not yet been included in an existing cluster, TypeSqueezer creates a new one (lines 4-5); Otherwise, if one item of the `<cs, ct>` pair is already included in a cluster, or if the two items belong to different clusters, TypeSqueezer unifies these clusters since their members all share the same signature (lines 6-7). In this way, TypeSqueezer progressively reduces the number of remaining clusters so that it achieves the best possible overall accuracy for subsequent arity/width inferencing.

2.2.2 Refined Arity Analysis. According to the clustering and the static analysis results, TypeSqueezer continues to refine the arity in each cluster. Note that within the same cluster, the prototype of all icalls and the signature of all AT functions should be the same. Therefore, the static analysis results for all members in the same cluster can be collectively consumed to arrive at a more accurate and stringent upper- and lower-bounds. Algorithm 2 outlines the workflow of our refined arity analysis, which is to iterate over the clusters in $CLTs$ returned by Algorithm 1, and for each cluster, to intersect the static analysis result of its individual members into a set of shared arity boundaries. This is done by iteratively selecting the minimum upper bound of arity over all icall members (lines 4-7) while selecting the maximum lower bound of arity among AT

Algorithm 2 Refined arity analysis

Input: $CLTs$: The clustering result, DCG: The direct call graph, $Arity$: Arity information obtained from static analysis
Output: Refined arity information

```
1:  function Refined Arity Analysis(CLTs, Arity)
2:      for each callsite clt ∈ CLTs do
3:          ub ← 6, lb ← 0, ret ← −1   ▷ -1 indicates the presence of a return value is unknown
4:          for each cs ∈ clt do
5:              if cs.ub < ub then
6:                  ub ← cs.ub
7:              end if
8:              if cs.ret = 1 then
9:                  ret ← 1
10:             end if
11:         end for
12:         for each calltarget ct ∈ clt do
13:             if ct.lb > lb then
14:                 lb ← ct.lb
15:             end if
16:             if ct.ret = 0 then
17:                 ret ← 0
18:             end if
19:         end for
20:         for each item ∈ clt do
21:             item.ub ← ub, item.lb ← lb, item.ret ← ret
22:         end for
23:     end for
24: end function
25: function Direct Call Analysis(DCG,Arity,Variadics)
26:     for each < cs, ct >∈ DCG ∧ ct ∉ Variadics do
27:         if cs.ub < ct.ub then
28:             ct.ub ← cs.ub
29:         end if
30:         if cs.ret = 1 then
31:             ct.ret ← 1
32:         end if
33:     end for
34: end function
```

functions (lines 12-14). Once the shared arity boundary is determined, it is assigned to each member of the cluster (lines 20-22). Similarly, if the static analysis phase of TypeSqueezer is able to assert on any member of a cluster that a return value must be expected (lines 8-10) or is guaranteed to not be prepared (lines 16-18), all members of the cluster are assigned with the same conclusion.

Figure 2 shows an example from SPEC CPU2006's 445.gobmk benchmark where an icall given in Figure 2a can be dynamically observed to target two AT functions, `autohelperbarrierspat66()` and `autohelperbarrierspat171()` (which are shown respectively in Figure 2b and Figure 2c). The icall here has argument registers **edi**, **esi**, **edx**, and **ecx** set, with static analysis concluding that the indirect call prepares at most 4 arguments. Meanwhile, we statically observe that `autohelperbarrierspat171()` conducts register reading on all four argument registers mentioned above, while the other AT function, `autohelperbarrierspat66()`, reads **edi**,

```
1  41dbab: jmpq   41d8a0
2  41dbb0: xor    %ecx,%ecx    ; set the 4th argument
3  41dbb2: mov    %r9d,%edx    ; set the 3rd argument
4  41dbb5: mov    %r9d,0x8(%rsp)
5  41dbba: mov    %ebp,%esi    ; set the 2nd argument
6  41dbbc: mov    %r15d,%edi   ; set the 1st argument
7  41dbbf: callq  *0xa8(%rbx)
```

(a) Indirect call which prepares at most 4 arguments

```
1  0000000000487700 <autohelperbarrierspat66>:
2  487700: movslq %edi,%rax    ; read the 1st argument
3  487703: mov    %edx,%r10d   ; read the 3rd argument
4  487706: mov    $0x2,%edx
5  48770b: mov    0xb96e20(,%rax,4),%r8d
6  487713: mov    0xb96980(,%rax,4),%ecx
7  48771a: mov    %r10d,%edi
8  48771d: xor    %eax,%eax
9  48771f: add    %esi,%r8d    ; read the 2nd argument
10 487722: add    %esi,%ecx
11 487724: mov    $0x1,%esi
12 487729: mov    %r8d,%r9d
13 48772c: jmpq   44d390
14 487731: nopl   0x0(%rax,%rax,1)
15 487736: nopw   %cs:0x0(%rax,%rax,1)
```

(b) Function `autohelperbarrierspat66()` which consumes at least 3 arguments

```
1  0000000000489c40 <autohelperbarrierspat171>:
2  489c40: movslq %edi,%rax    ; read the 1st argument
3  489c43: test   %ecx,%ecx    ; read the 4th argument
4  489c45: mov    0xb96e60(,%rax,4),%r8d
5  489c4d: mov    0xb97300(,%rax,4),%edi
6  489c54: je     489c78
7  489c56: mov    $0x3,%eax
8  489c5b: sub    $0x8,%rsp
9  489c5f: add    %esi,%edi    ; read the 2nd argument
10 489c61: sub    %edx,%eax    ; read the 3rd argument
11 489c63: mov    %eax,%esi
12 489c65: callq  41e2a0
13 489c6a: xor    %eax,%eax
14 489c6c: add    $0x8,%rsp
15 489c70: retq
16 489c71: nopl   0x0(%rax)
17 489c78: lea    (%r8,%rsi,1),%edi
18 489c7c: mov    %edx,%esi
19 489c7e: jmpq   445240
```

(c) Function `autohelperbarrierspat171()` which consumes at least 4 arguments

Figure 2: An example that shows how dynamic profiling can help refine arity information

esi and **edx** only. This means that the two functions are assumed to consume at least 3 and 4 arguments, respectively. Since the control flows between the icall and the two AT functions are dynamically captured, TypeSqueezer puts them into the same cluster and deduces that all members within this cluster share the upper bound of 4 (obtained from the icall) and the lower bound of 4 (obtained from `autohelperbarrierspat171()`). In contrast, existing work with static analysis only would have the arity boundary of the aforementioned icall and AT functions be [0,4], [3,6], and [4,6], respectively — a much looser bound than what we achieve with TypeSqueezer.

The return value is a special case in arity analysis, i.e., while it can be seen as an extra argument, inferences of it made by the static analysis are flawed in different ways for icalls and AT functions. Specifically, static analysis can only tell if an icall expects a return value, or, if an AT function prepares no return value. In the case where an icall never reads the return value prepared by its callee, static analysis would determine the lower bound of the icall's return value number as 0 (which is in fact incorrect). To deal with return values in the arity analysis, TypeSqueezer is designed to intersect the following rules to all members of a cluster:

- members in a cluster must have a return value if any icall within the cluster is statically determined to be expecting a return value; and
- members in a cluster must have no return value if any AT function within the cluster is guaranteed to prepare no return value at the static phase.

On top of the above mechanisms, TypeSqueezer further improves the accuracy of the recovered CFG by exploiting the observation that *AT functions can be targeted by both indirect and direct control transfers.* For example, in C++, static polymorphism (or compile-time polymorphism) will use a direct call instruction to implement a virtual call if the object type can be determined by the compiler. Therefore, by intersecting the static inferences regarding arguments and return values of such direct calls to those of the directly targeted AT functions, TypeSqueezer introduces one additional source of information for further refining its arity analysis. Specifically, TypeSqueezer traverses all direct calls in the binary. If a direct call targets an AT function, TypeSqueezer performs reaching-definition analysis (similar to the callsite analysis of TypeArmor) to compute the upper bound of arity as well as the lower bound of return value number for the direct call, and uses the result to constrain the corresponding inferences on the AT function (lines 26-33 of Algorithm 2).

2.2.3 *Argument Width Analysis.*

Our refined argument width analysis leverages similar approaches as in the arity analysis, i.e., we seek to compute the lower bound of width for the argument of icalls and the upper bound of width for AT functions. Again, in each cluster, the static width analysis results on different members are intersected to reach a unanimous set of upper and lower bounds. Specifically, given the refined arity and return value inferencing result of a cluster, TypeSqueezer traverses each cluster member and finds the maximum lower bound and minimum upper bound for the width of each argument. The results are then applied to all members in the cluster, so that a CFI scheme based on TypeSqueezer's analysis result can enforce a stricter forward-edge policy. Note that width analysis for return values is also in contrast with that for arguments, in which the upper bound of the return value width is determined at the calltarget, while the lower bound is determined at the callsite. Other than this, the boundary intersection of return value widths follows the same general procedure as that for argument widths.

To demonstrate that our width analysis helps refine the assumed width of arguments, consider the example given in Figure 3 in which a function pointer in `splay_tree_splay_helper()` that prepares two parameters (see line 18 of Figure 3a) is found to be targeting the AT function `splay_tree_compare_ints()`, with the latter consumes the two arguments after converting them to type `int` rather than their original type `unsigned long` (lines 6 and 8 in Figure 3c). From the assembly of the caller and callee functions, we can see that the icall corresponding to the given function pointer indeed prepares its arguments in full 64-bit width (lines 2-3 of Figure 3b), so the statically assumed argument width should be of the range [0, 64]. However, after being paired with the AT function, TypeSqueezer finds that the latter only reads from two 32-bit registers (line 3 of Figure 3d). Therefore, by analyzing the given icall and AT function (that are guaranteed to be dynamically clustered due

```
1  static splay_tree_node
2  splay_tree_splay_helper (sp, key, node, parent, grandparent)
3       splay_tree sp;
4       splay_tree_key key;    // typedef unsigned long splay_tree_key
5       splay_tree_node *node;
6       splay_tree_node *parent;
7       splay_tree_node *grandparent;
8  {
9    splay_tree_node *next;
10   splay_tree_node n;
11   int comparison;
12
13   n = *node;
14
15   if (!n)
16     return *parent;
17
18   comparison = (*sp->comp) (key, n->key);
19   ...
20 }
```

(a) Source of an icall in function `splay_tree_splay_helper()`

```
1    ...
2    638419: mov    (%rbx),%rsi    ; set 2nd argument
3    63841c: mov    %r15,%rdi      ; set 1st argument
4    63841f: callq  *0x8(%r12)
```

(b) Assembly of the corresponding icall

```
1  int
2  splay_tree_compare_ints (k1, k2)
3       splay_tree_key k1;     //typedef unsigned long splay_tree_key
4       splay_tree_key k2;
5  {
6    if ((int) k1 < (int) k2)  // unsigned long is cast to int
7      return -1;
8    else if ((int) k1 > (int) k2)
9      return 1;
10   else
11     return 0;
12 }
```

(c) Source of the callee AT function `splay_tree_compare_ints()`

```
1  0000000000638a70 <splay_tree_compare_ints>:
2    638a70: xor    %eax,%eax
3    638a72: cmp    %esi,%edi       ; read 1st and 2nd argument
4    638a74: mov    $0xffffffff,%edx
5    638a79: setg   %al
6    638a7c: cmovl  %edx,%eax
7    638a7f: retq
```

(d) Assembly of the corresponding AT function

Figure 3: An example from 403.gcc in which the argument width analysis of TypeSqueezer has impact.

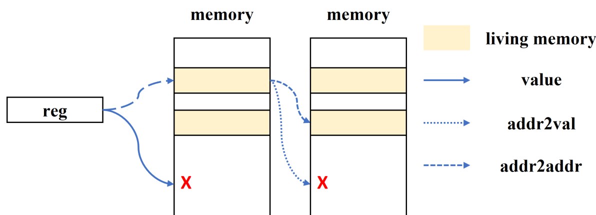

Figure 4: Different scenarios considered by the value-based type inference of TypeSqueezer.

to their dependency) together rather than inspecting each of them individually, TypeSqueezer could more accurately determine that the range of their argument width be [32, 64].

2.3 Dynamic Value-Based Type Inference

Pointers are a fundamental concept of C/C++ programming language and are widely used to support memory manipulation, efficient data access, and other program behaviors. In x86_64, pointers are translated to address references whose widths are 64-bit at the binary level. Therefore, even an ideal width analysis that is able to reconstruct the precise width of arguments cannot tell the difference between two pointer arguments at binary level. Worse still, existing static width analysis is conservative and approximates argument widths with their possible ranges. To give an example, when an argument of an icall is a pointer (which would hence be assumed to be at most 64 bits long), it would be considered compatible with any arguments that are of an arbitrary type because the maximum width of an argument register is 64-bit in x86_64.

With the capability of conducting dynamic analysis on the subject binaries, TypeSqueezer is able to produce signatures of icalls and AT functions with richer type-related profiles. This is done by inspecting registers and memory locations involved in argument passing at runtime. In particular, at each indirect callsite, TypeSqueezer inspects the content stored in the argument registers to infer their types. As illustrated in Figure 4, if an argument register points to the living address space of the process, TypeSqueezer concludes that it is an address reference; otherwise, it is considered a value. Furthermore, when an argument is recognized as an address reference, TypeSqueezer examines the memory location where it points, and differentiates an address reference of another address from an address reference of a value[2]. The above strategy enables TypeSqueezer to distinguish between integers, pointers, and pointers to pointers, such as `uint64_t`, `char *`, and `char **`, as different types. Similar to §2.2, TypeSqueezer determines the type of arguments via the aggregation of inferences on all individual members of the same cluster. In addition, by observing runtime context at the location of return instructions, TypeSqueezer also recovers the type of return values for icalls and AT functions.

Although TypeSqueezer is powerful in recognizing the parameter types at a fine-grained level, we also recognize three complexities in its type inference, which we briefly explain below.

Intended Type Polymorphism. Note that during the execution of a program, a function might be executed several times with arguments assigned with different values, or even with different types in some special scenarios. For example, a union is a composite data type that allows different data types to share the same memory location, thus it may cause the same argument to be assigned with different types at runtime, which affects the inference made by our approach. We call this issue the *intended type polymorphism*. For correctness concerns, when a potential type polymorphism is detected during dynamic analysis, TypeSqueezer is designed to act prudently and identify the corresponding argument as a value rather than an address reference. In particular,

- if an argument is observed to be both a value and an address reference, it is considered a value; and
- if an argument is observed to be both a reference of address and a reference of value, it is considered an address reference indexing a value.

TypeSqueezer adopts the above policy based on the following reasons. First, an argument holding a value may coincidentally fall into the living address space of the process. Second, it ensures that

[2]Note that for arguments observed to be pointing to an executable memory region, TypeSqueezer only considers those whose values align with the legal instruction boundaries of that region as address references.

```
/* RTL expression ("rtx"). */
struct rtx_def
{
  ENUM_BITFIELD(rtx_code) code: 16;
  ENUM_BITFIELD(machine_mode) mode : 8;
  unsigned int jump : 1;
  unsigned int call : 1;
  unsigned int unchanging : 1;
  unsigned int volatil : 1;
  unsigned int in_struct : 1;
  unsigned int used : 1;
  unsigned integrated : 1;
  unsigned frame_related : 1;
  rtunion fld[1];
};
```

Figure 5: Definition of `struct rtx_def`.

when TypeSqueezer determines the unanimous function signature of a specific cluster with the presence of type polymorphism among the cluster's members, it does so conservatively such that the resulting CFI protection would not reject legal control flows.

Leveraging ASLR. In some special cases, an argument of value type might accidentally point to a valid address. To give an example, Figure 5 shows the declaration of `struct rtx_def` used by the 403.gcc benchmark from SPEC CPU2006, which is by all means a bit string encoding a configuration list. Such a variable would unlikely have its value changed across multiple executions of the program. We observed that a function named `c_safe_from_p()` consumed a parameter of this type, which was always of the same value yet pointing to the program's heap region. To handle this issue, we leverage the fact that *real address references are affected by ASLR (Address Space Layout Randomization) and would have their values changed for each individual execution of the program.* Specifically, when a particular argument is observed to be pointing to an address within a position-independent memory region (e.g., stack, heap, or any section of a shared library), TypeSqueezer would run the subject program multiple times with the same input, and see if the value changes — if not, then the argument will be conservatively inferred as a value.

Arguments of Zero Value. A special case that TypeSqueezer needs to deal with is that arguments might hold zero value at runtime, leading to an ambiguity of pointers and integers. Therefore, when TypeSqueezer dynamically detects that the value of an argument (or the memory entry indexed by the argument) is zero, it considers the argument as an *unknown* type (or an *addr2unknown* type). To maintain robustness, TypeSqueezer considers the unknown type to be always compatible with any other type, and the addr2unknown type to be compatible with any address reference (including addr2val and addr2addr types). As such, given that TypeSqueezer leverages dependency-based clustering to finalize the assumed type for each argument, it can be expected that the aforementioned handling of zero values does not affect the overall accuracy of TypeSqueezer in argument type inferencing.

3 IMPLEMENTATION

We have implemented a proof-of-concept prototype of our TypeSqueezer approach on top of Dyninst [3] (v9.3.1) and Intel Pin [26] (v3.27). Currently, our prototype only supports x86_64 executables in the ELF format that comply with the System V ABI [27] calling convention.

3.1 Static Analysis

The static analysis phase of our prototype mainly carries out two tasks, namely to perform static arity/width analysis on subject binaries as in existing work [23, 28, 37], and to run direct call analysis of which the result assists our refined arity analysis (see the last paragraph of §2.2.2).

Static Arity/Width Analysis. In order to avoid the complications brought by compiler optimization, our prototype leverages the improved static type-based matching technique proposed by Lin et al. [23] to perform static arity/width analysis, given that this more recent solution has fixed a number of complier-optimization-related shortcomings of previous schemes. We omit more details regarding the mechanism of such static analysis since they are not the contribution of this work.

Direct Call Analysis. The purpose of direct call analysis is to discover AT functions that are directly invoked, such that more constraints can be applied in the arity/width analysis of TypeSqueezer. This begins by consulting the disassembly code of the subject binary to retrieve all direct call instructions and the corresponding call targets. Upon identifying any callsite that targets an AT function, a reaching-definition analysis is performed to determine the upper bound of arity as well as argument widths for its prototype. For the return value, TypeSqueezer performs liveness analysis to determine its existence and, if it exists, the lower bound of its width. The two analyses here adopt the same algorithm in TypeArmor and τCFI, except that TypeSqueezer directly uses the API provided by Dyninst to decode instructions, inspect each instruction's read/write effect on certain registers, and obtain the width information of those registers.

3.2 Dynamic Analysis

Note that the main goal of TypeSqueezer's dynamic analysis is to capture dependencies between icalls and AT functions during actual executions of the subject binary, while also inspecting the runtime context at the location of each icall. To fulfill these requirements, we use Intel Pin, a well-known dynamic instrumentation framework as the basis of our prototype because it provides the desired execution interception and instrumentation capabilities while being sufficiently flexible, scalable, and customizable. Specifically, we write a customized Pin tool to instrument icalls and return instructions of the subject binary, so that during its execution, key execution traces as well as the read operations of the runtime snapshots of both CPU registers and the process's memory space, can be collected.

Input Generation. As the functionality of TypeSqueezer relies on dynamic analysis, we note that code coverage would definitely impact the effectiveness of our approach. However, improving code coverage for dynamic binary analysis is out of the scope of this paper. Thus, the PoC implementation of TypeSqueezer simply leverages an open-source coverage-guided fuzzer, AFL++ [15], to generate test cases for subject binaries. Specifically, we use handwritten test suites provided by developers as corpora to feed the fuzzer and use the fuzzer's output to run the binaries.

Collecting Execution Traces. To understand how icalls and AT functions are paired during execution, TypeSqueezer records the runtime control flows with an instrumentation that adds an

analysis routine before each icall at the instruction granularity. An icall is attached with the analysis routine only when it executes for the first time. After the instrumentation, the address of the hooked icall can be retrieved by passing an `IARG_INST_PTR` argument to the analysis routine, whereas the target address can be retrieved by passing an `IARG_BRANCH_TARGET_ADDR` argument. The above two arguments together then form the execution traces for an icall in the form of a `<cs, ct>` pair. It is worth pointing that a particular `<cs, ct>` pair could occur millions of times throughout the execution of a subject binary. To avoid recording duplicate traces and to reduce performance overhead, TypeSqueezer uses a map to filter out previously recorded traces.

Reading Registers And Memories. The value-based type inference of TypeSqueezer requires both in-register and in-memory values. The consulting of a register value is done by passing an `IARG_REG_VALUE` argument and an additional `REG` argument (to specify the target register) to the analysis routine. The consulting of a memory location is fulfilled using the `Pin_SafeCopy()` API. Note that when `Pin_SafeCopy()` returns, a zero return value indicates that the accessed memory location is invalid (i.e., not in the living address space of the process). We leverage this feature to determine whether an argument is an address reference or not, and to distinguish address references that index values and other addresses.

Instrumenting Multi-Threaded Applications. When Pin is stitched to a multi-threaded program, the instrumented code residing in multiple threads might write the output stream simultaneously, resulting in asynchronous or even invalid information captured. We leverage the locking primitives that Pin provides to lock the output stream when an analysis routine is writing to it.

3.3 Resolving Type-Based Indirect Call Targets

Given the results of hybrid arity/width analysis (§2.2) and value-based type inference (§2.3), TypeSqueezer leverages type-based matching to resolve indirect call targets. In addition to the policies regarding arity and argument width that are adopted by existing solutions [23, 28, 37], TypeSqueezer further examines whether each argument type is compatible. In particular, an unknown type is always compatible with any type, and an addr2unknown type is compatible with any address reference (including addr2val and addr2addr types), while an address reference type is not compatible with a value type. Once all the checks are passed, TypeSqueezer considers the AT function a potential target for the indirect call.

3.4 CFI Hardening

Our current prototype of TypeSqueezer is able to fortify COTS binaries with label-based CFI enhancement, which is done in a way akin to the Tary and Bary table implementation in MCFI [29]. In particular, TypeSqueezer embeds two maps into the subject binary, which are associated respectively with icalls and AT functions. Each entry of the maps is a key-value pair where the key is the address of an icall or AT function and the label being the corresponding function signature recovered via our hybrid analysis. TypeSqueezer leverages Dyninst [3] as the foundation for static binary rewriting, incorporating integrity checks before icalls within the subject binary to retrieve and compare the labels of the callsite and calltarget before executing an indirect control-flow transfer. A control transfer is allowed when the two labels provide compatible signatures, hence achieving CFI enforcement.

A CFI scheme must maintain the correctness of fortified programs. However, due to the limitation of dynamic analysis w.r.t. completeness of the test cases, value-based type inference might, to be admitted, not be entirely foolproof in some unforeseen scenarios. We address this concern by embedding a runtime review component as part of its fortification, which double checks forward-edges that violate CFI only due to inferred argument types. Specifically, we maintain two versions of CFG inferences, with CFG_a built using its full capability while CFG_b produced based on the hybrid arity/width analysis. An indirect flow complying with CFG_b but not with CFG_a is reviewed on-the-fly by checking if the call target is potentially abused as a forward-edge gadget.

To the best of our knowledge, there could be three types of control-flow hijacking attacks that divert the execution of an icall: JOP [4], COOP [12, 34], and Control Jujutsu [13]. Specifically, JOP and COOP rely on a dispatcher gadget to chain other gadgets, leading to *repeated violations of indirect control-flow integrity.* The Control Jujutsu attack corrupts the in-memory function pointer as well as argument of the Argument Corruptible Indirect Call Site to divert the control flow to a function close to a powerful system call, and achieves remote code execution via a *malicious system call.*

As such, TypeSqueezer simply checks: 1) if the currently reviewed callsite belongs to a previously reviewed call target (JOP), 2) if the same callsite is repeatedly reviewed (COOP), and 3) if the first sensitive system call within the currently reviewed call target (if any) is to be abused should the program run with the current execution context. Suspected control flows are deemed safe and allowed once these checks pass, and the correctness of fortified programs is ensured.

4 EVALUATION

To evaluate our approach, we applied TypeSqueezer on the SPEC CPU2006 benchmarks and a number of popular real-world applications for a thorough analysis. We mainly evaluated TypeSqueezer in the following aspects.

- **Effectiveness.** The main goal of TypeSqueezer is to improve the precision of forward-edge CFG recovered, i.e., to refine indirect call target resolving for binary executables. Thus, we used the AICT (average indirect call targets) vector for evaluation. Compared to type-based CFG generation techniques built on pure static analysis, the evaluation results show the extent to which dynamic analysis helps in refining indirect call targets.
- **Soundness.** CFI policy must be sound and free of correctness concerns. We assessed the soundness of TypeSqueezer by comparing CFGs it generates against those collected using dynamic profiling. To evaluate the impact of potential value collisions, we further conducted a dedicated analysis on the dynamic value-based type inference of our approach.
- **Case Studies.** We discussed some insightful cases we discovered when evaluating TypeSqueezer. Specifically, we showed how TypeSqueezer thwarts a practical attack, and provided an example to demonstrate the inherent limitations of static analysis.

Table 1: Fuzzing duration for each SPEC CPU2006 benchmark program.

Benchmark	401.bzip2	458.sjeng	433.milc	482.sphinx3	456.hmmer	464.h264ref	445.gobmk	400.perlbench
Time (hours)	96	117	41	135	139	126	88	189
Benchmark	403.gcc	483.xalancbmk	471.omnetpp	447.dealII	473.astar	450.soplex	453.povray	444.namd
Time (hours)	130	186	14	35	22	72	95	113

Table 2: The AICT comparison results of TypeSqueezer with representative binary-level approaches on SPEC CPU2006 benchmarks and real-world applications.

Benchmark	icalls	funcs	Coverage		AICT					
			icall	AT	AT [46]	TypeArmor [37]	τCFI [37]	Lin et al. [23]	TypeSqueezer	TypeSqueezer*
401.bzip2	20	73	60%	66.67%	3	2.05	2.05	1.80	1.65	1.50
458.sjeng	1	138	100%	66.67%	9	9	8	9	8	8
433.milc	4	234	75%	66.67%	3	3	3	3	3	3
482.sphinx3	7	323	0%	0%	7	2	2	2	2	-
456.hmmer	10	487	100%	8.70%	23	23	20	20	19.90	19.90
464.h264ref	352	515	16.48%	50%	42	41.01	16.74	19.35	16.21	10.62
445.gobmk	43	2507	86.05%	76.31%	1790	1645.30	1607.3	1635.63	658.07	519.35
400.perlbench	114	1697	62.28%	44.54%	723	652.10	618.33	624.47	429.97	349.35
403.gcc	460	4559	54.35%	35.62%	1269	885.87	676.42	693.64	505.35	386.52
483.xalancbmk	11958	13465	13.56%	9.93%	8030	6643.15	6232.26	6139.43	5999.81	5248.25
471.omnetpp	873	2013	45.02%	28.32%	1243	1021.21	987.54	968.96	880.84	739.87
447.dealII	1777	7200	5.63%	4.42%	2510	2116.39	1989.55	1990.83	1986.09	1986.80
473.astar	1	89	100%	25%	4	4	4	4	2	2
450.soplex	583	917	28.99%	33.95%	436	314.75	290.46	286.44	267.85	240.07
453.povray	149	1605	27.52%	12.84%	522	402.24	365.68	350.80	334.24	320.81
444.namd	14	96	85.71%	24.49%	49	48.43	48.43	48.43	48.43	48.50
Nginx	396	1869	74%	81.33%	884	653.08	513.05	530.56	202.11	148.18
ProFTPD	142	1986	44.37%	82.78%	389	324.78	316.67	309.08	227.64	133.65
lighttpd	91	1222	63.74%	87.34%	79	32.35	29.25	29.52	16.88	10.10
Memcached	107	431	91.59%	67.68%	99	93.73	72.36	70.06	43.34	41.60
Redis	389	928	37.53%	23.71%	928	672.92	579.38	483.59	351.44	232.05
geomean	83.26	822.44	47.67%	34.23%	145.17	111.26	100.51	100.04	76.82	65.02

Table 3: The AICT comparison results of TypeSqueezer with representative binary-level approaches on covered icalls.

Approach	AT[46]	TypeArmor[37]	τCFI[37]	Lin et al.[23]	TypeSqueezer
geomean	145.17	112.87	102.98	100.76	65.02

- **Performance.** Finally, we tested the overhead introduced by applying TypeSqueezer, and verified the compatibility of our approach.

4.1 Effectiveness

4.1.1 AICT comparison. Our evaluation dataset includes C/C++ benchmarks of the SPEC CPU2006 suite and a number of real-world applications, namely two web servers (Nginx v1.20.2 and lighttpd v1.4.68), an FTP server (ProFTPD v1.3.6), an in-memory data store (Redis v5.0.14), and a general-purpose distributed memory caching system (Memcached v1.6.19). We evaluated TypeSqueezer on binaries compiled in standard compiling options with **-O2** optimization level. Note that benchmark programs that do not contain indirect callsites, i.e., 429.mcf, 470.lbm, and 462.libquantum, were excluded from our evaluation. When calculating AICT, we also excluded compiler-generated icalls and AT functions which are used to set up the runtime environment. The test cases of the SPEC benchmark programs were generated via coverage-guided fuzzing (see §3.2) running on a server powered by a 32-core Intel E5-2630 CPU. The duration of fuzzing for each SPEC benchmark program was given in Table 1, with the average, maximum, and minimum duration being 100 hours, 189 hours (on 400.perlbench), and 14 hours (on 471.omnetpp), respectively. The selected real-world applications all come with comprehensive hand-written test suites, which were directly employed as inputs for the dynamic analysis of TypeSqueezer.

We compared TypeSqueezer with representative binary-level indirect call target resolving approaches on SPEC CPU2006 benchmarks and real-world applications, and the results are shown in Table 2. Note that we mainly compare TypeSqueezer with state-of-the-art type-based techniques, including TypeArmor [37], τCFI [28], and Lin et al. [23]. We excluded BPA [20] from the comparison for

Table 4: Breakdowns on the contribution of TypeSqueezer's functional components to the achieved AICT reduction.

Component	Static Arity	+Static Width	+Hybrid Arity/Width	+Direct Call	+Value-Based Type Inference
geomean	111.39	100.04	83.89	79.64	76.82

the following reasons: (1) it only targets x86_32 binaries, (2) it is a points-to analysis rather than a type-based approach, and (3) its source code is not publicly available.

The second (*icalls*) and third columns (*funcs*) of Table 2 indicate the number of indirect callsites and the number of functions in each benchmark or application, respectively. In the columns of *Coverage*, we gave the coverage ratios of indirect call (*icall*) and AT function (*AT*) covered by our dynamic analysis to reflect the impact of code coverage on the AICT results. Note that we leveraged the coverage of indirect call and AT function to represent code coverage instead of the coverage of line/branch which is commonly used in software testing, as the coverage of executed indirect calls and AT functions is more relavant to TypeSqueezer. In addition, in the columns of *AICT*, *AT* gives the AICT results for approaches that only allow an icall to target all AT functions (e.g., binCFI [46]), whereas *TypeArmor*, *τCFI*, *Lin et al.[23]*, *TypeSqueezer* and *TypeSqueezer** give the AICT results for the three representative binary-level approaches as well as ours. Note that we show two levels of AICT results for our approach, namely, the overall result (column *TypeSqueezer*) and that for dynamically covered icalls in specific (column *TypeSqueezer**). We also presented in Table 3 the geometric mean of AICT results regarding the collection of icalls covered by our dynamic analysis. To give a fair comparison, the AICT for TypeSqueezer and several representative type-based solutions on this specific collection of icalls were all listed.

As shown in Table 2, compared to TypeArmor, τCFI, and Lin et al. [23], TypeSqueezer reduces the geometric mean of AICT by 30.95% (111.26 vs 76.82), 23.57% (100.51 vs 76.82), and 23.21% (100.04 vs 76.82), respectively. Moreover, as shown in Table 3, for those icalls covered by the dynamic analysis, TypeSqueezer further reduces the geometric mean of AICT by 55.21% (145.17 vs 65.02), 42.39% (112.87 vs 65.02), 36.86% (102.98 vs 65.02), and 35.47% (100.76 vs 65.02), respectively. These suggest that TypeSqueezer introduces a significant reduction of overall AICT and outperforms state-of-the-art techniques in refining indirect calltargets.

Among the results presented in Table 2, we found that the AICT reductions are highly correlated with the coverage ratio of dynamic analysis. Specifically, on programs with high coverage of icalls and AT functions like 445.gobmk and Nginx, TypeSqueezer generates impressive results. For example, compared to Lin et al. [23], TypeSqueezer reduces the AICT of Nginx by 61.91% (530.56 vs 202.11); while in contrast, as our dynamic analysis cannot reach any indirect calls in 482.sphinx3, the AICT remains the same as Lin et al. [23], which highlight the importance of high code coverage of dynamic analysis in TypeSqueezer. It is worth pointing out that high and balanced coverage of both icall and AT function is important in TypeSqueezer. For instance, although the coverage of AT function for ProFTPD is extremely high (i.e., 82.78%), the relatively low coverage of icall (i.e., 44.37%) limits TypeSqueezer's ability in reducing AICT. We believe the reason is that while most AT functions' signatures were indeed refined, TypeSqueezer does not bring extra confinement to the signatures of half of the icalls; as a result, although the AICT for dynamically covered icalls decreases dramatically (133.65 on average for this particular benchmark), the same effect was unfortunately not extended to the overall CFG.

Particularly, in small benchmark programs that have only a few or tens of icalls, e.g., 401.bzip2, 458.sjeng, and 433.milc, TypeSqueezer may not perform much better than other approaches, even if the code coverage of dynamic analysis is high. A typical example is 458.sjeng, which only contains one icall where most AT functions are its actual targets. In such situations, type-based matching nearly reaches its extreme limit. Fortunately, in such programs, the AICT is small enough to provide good security guarantees.

It is interesting to see that for 444.namd and 456.hmmer, TypeSqueezer's AICT almost remains the same as in Lin et al [23]. With manual inspection, we found that in 444.namd, the icalls covered by dynamic analysis all share the same function signature, i.e., *void (*)(nonbonded*)*. While in 456.hmmer, only one icall has a different function signature from others. As a result, the AICT computed by TypeSqueezer only decreases a little compared to existing type-based approaches (i.e., Lin et al. [23]). Moreover, in 444.namd and 447.dealII, the results for *TypeSqueezer* and *TypeSqueezer** are counterintuitive, as the AICT for icalls that are covered by TypeSqueezer's dynamic analysis is coincidentally larger than the AICT for all icalls computed by TypeSqueezer. We believed that this is due to the non-uniform distribution of AICT.

Interestingly, in 473.astar, it turns out that TypeSqueezer's direct call analysis confines the upper bound of arities for two untriggered AT functions. As a result, these two functions are excluded from the icall's target set as their arities mismatch that of the icall. This example partially demonstrates the effectiveness of the direct call analysis in TypeSqueezer.

4.1.2 AICT reduction breakdowns. To further analyze the individual contribution of TypeSqueezer's components towards reducing AICT, we conducted separate evaluations on the evaluation dataset (including the SPEC CPU2006 benchmarks and real-world applications in Table 2) and presented the geometric mean of AICT results in Table 4. In the Table, "Static Arity" stands for static arity analysis, "+Static Width" further introduces static width analysis when computing AICT, "+Hybrid Arity/Width" adds TypeSqueezer's hybrid arity/width analysis but excludes the direct call analysis, "+Direct Call" supplements the direct call analysis. Finally, "+Value-Based Type Inference" corresponds to the AICT when introducing all components of TypeSqueezer.

As shown in Table 4, the most significant contribution to AICT reduction is the hybrid arity/width analysis, while direct call analysis and value-based type inference further reduce AICT by 4.25 and 2.82, respectively. In summary, each component of TypeSqueezer demonstrated positive effects regarding AICT reduction.

4.1.3 Prevalence of address references. We observed that the most common type of argument is the pointer. To understand the prevalence of this type, we conducted a study on SPEC CPU2006, and found that about 72.81% arguments among all functions are in the

pointer type. The only exception goes to 445.gobmk, in which 32-bit integers are heavily used. A large number of pointers among arguments suggest that the effectiveness of width-based indirect call targets resolving techniques (e.g., τCFI) may not be as effective as expected. The reason is that a 64-bit argument at the callsite is compatible with any argument type at the calltarget. This is an important indicator that motivates TypeSqueezer, which makes a further step to distinguish between values and address references.

4.1.4 Thwarting advanced code-reuse attacks. To see TypeSqueezer in action of a CFI implementation to mitigate code-reuse attacks, we analyzed the feasibility of advanced code-reuse attacks when the CFG recovered by TypeSqueezer is enforced on target binaries. Results are presented in Table 5, where "✓" indicates successful defense for an exploit and vice versa for "✗". According to the results, by enforcing the AT policy [45] and the arity policy [37], all function-reuse attacks were thwarted except the improved Apache exploit (denoted as *Apache**) from Control Jujutsu [13] that bypasses all previous binary-level type-based CFI solutions, including TypeArmor [37], τCFI [28], and Lin et al. [23]. In contrast, such an attack is defeated by TypeSqueezer. Further details about how TypeSqueezer thwarts such an attack are discussed in §4.3.1.

4.2 Soundness

4.2.1 Collecting ground truth. To collect the source-level function signatures for icalls and AT functions, we implemented an LLVM pass that operates at the IR level. Note that at the beginning we only had the function prototypes at the IR level. To propagate them to the binary level, we compiled the benchmark programs with debug information (i.e., with *-g* compiling options). By inspecting the line number to binary address mapping from the debug section in ELF files, the ground truth for a given binary address can be obtained. Note that multiple indirect calls residing at the same line are not supported in our current implementation, nor do they appear in our dataset.

4.2.2 Soundness verification. We leverage a methodology similar to previous CFI solutions [20, 45] to compare the CFG recovered by TypeSqueezer with that recorded by dynamically profiling the binary (again, we use Intel Pin here). As our approach relies on hybrid analysis, the input used for CFG inference cannot be identical to that used for dynamic-profiling-based verification. We hence performed this particular evaluation using SPEC CPU2006 benchmarks only — all SPEC benchmark programs have three input sets (i.e., train, test, and reference), which directly satisfies the requirement of this evaluation. Specifically, for benchmark programs with multiple *reference* inputs (the most complex ones), we used a random one of them for verification and the others as inputs to TypeSqueezer for CFG generation. For benchmark programs that come with only one reference input, we used the reference input for TypeSqueezer and the test/train inputs for verification. Detailed per-benchmark statistics regarding the number of arguments covered by value-based type inference, the number of collected dynamic samples, as well as the proportion of inferred types, are given in Table 6. Overall, our experiments investigated a total of 4825 arguments. On average, each argument has 13840607.47 samples collected, with 20.15 distinct values among those collected values. The distribution of inferred types is as follows: *Addr2Addr* accounts for 44.39%, *Addr2Val* for 17.76%, *Addr2Unknown* for 2.46%, *Value* for 25.75%, and *Unknown* for 9.64%.

In this experiment, TypeSqueezer achieved a recall rate of 100%, i.e., no valid edges were found rejected when comparing the CFG inferred by TypeSqueezer to the CFG collected from dynamic profiling. By further comparing the recovered function signatures with those collected from LLVM IR, we found no violation of TypeSqueezer's hybrid arity/width analysis, strengthening the soundness of this component.

4.2.3 Manual review of potential value collision. As suggested by prior research [38, 39], value collisions (where a value coincidentally resides within an accessible memory region) may potentially lead to incorrect result of our dynamic value-based type inference, consequently compromising the soundness of the generated CFG. To ascertain the proper handling of value-collision scenarios by our approach, we conducted a thorough manual review on arguments of the evaluated benchmark programs, specifically on cases where the inferred type given by TypeSqueezer is different from their source-level ground truths. The goal of the manual review is to check whether the causes of such identified type inaccuracies are indeed value collisions. We encountered discrepancies in 54 arguments' recovered signatures compared to those obtained from LLVM IR. Our manual scrutiny revealed that the majority of these discrepancies were attributed to unreliable debug information (e.g., line numbers), a well-known issue in highly optimized binaries [43]. Upon closer examination, we determined that only 4 of the 54 instances diverged from their actual source-level types.

We found that *all the wrongly inferred arguments were in fact due to explicit type casting*. For instance, function `case_compare()` of the 403.gcc benchmark declares two arguments of type `unsigned long`, but then manipulates them as pointers. In other words, these "incorrect" cases caused by TypeSqueezer were capturing the actual runtime semantics of analyzed programs. We further discuss this potential limitation of our approach in § 5.

4.3 Case Studies

4.3.1 Thwarting the improved Apache exploit. To further demonstrate the refined CFI policy that TypeSqueezer constructs and its superior capability in defending against control-flow hijacking attacks, we leverage the improved Apache exploit introduced in Control Jujutsu [13] as a case study. In the original Apache exploit, control flow is hijacked from the icall residing in function `ap_run_dirwalk_stat()`, which then redirects to a non-AT function `piped_log_spawn()`. It can be prevented by enforcing the address-taken policy proposed by bin-CFI [46]. The authors of Control Jujutsu argue that the target function can be replaced by an alternative, e.g., function `ap_open_piped_log_ex()` which eventually invokes the target function by two direct calls.

After manually checking the source code of the icall and the alternative function, we confirm that TypeArmor, τCFI, and Lin et al.[23] cannot prevent the improved attack from happening. In particular, the new function and the callsite both have 3 arguments

Table 5: Thwarting advanced code-reuse attacks.

Exploit	Target	Defense			
		TypeArmor[37]	τCFI[28]	Lin et al.[23]	TypeSqueezer
COOP ML-G [34]	IE-64bit	✓	✓	✓	✓
	IE-64bit(2)	✓	✓	✓	✓
	Firefox	✓	✓	✓	✓
COOP REC-G [12]	Chromium	✓	✓	✓	✓
Control Jujutsu [13]	Apache	✓	✓	✓	✓
	Apache*	✗	✗	✗	✓
	Nginx	✓	✓	✓	✓

Table 6: Per-benchmark statistics of value-based type inference.

Benchmark	Argument Count	Samples (Per Argument)		Proportion of Inferred Types				
		Total Amount	Distinct Values	Addr2Addr	Addr2Val	Addr2Unknown	Value	Unknown
401.bzip2	30	3	1.47	6.67%	13.33%	0	40%	40%
458.sjeng	2	5069955451	64	0	0	0	72.33%	27.67%
433.milc	18	76800.55	12.39	0	16.67%	0	47.50%	35.83%
482.shpinx3	0	0	0	0	0	0	0	0
456.hmmer	16	108494.50	43.81	87.50%	12.50%	0	0	0
464.h264ref	192	100657293.72	20.28	21.88%	8.27%	0	68.16%	1.69%
445.gobmk	118	4758750.46	79.59	3.39%	12.07%	6.90%	63.53%	14.10%
400.perlbench	169	11866985.90	27.97	52.87%	22.94%	9.89%	6.76%	7.54%
403.gcc	333	267524.87	57.28	22.07%	42.35%	11.44%	19.18%	4.95%
483.xalancbmk	3285	3782172.86	13.56	67.94%	9.95%	1.91%	15.79%	4.41%
471.omnetpp	210	5834602.99	43.72	78.09%	4.78%	0.93%	13.31%	2.89%
447.dealII	175	1420575.10	17.22	68.44%	11.27%	3.15%	15.15%	1.99%
473.astar	3	2780359179	87	33.33%	66.67%	0	0	0
450.soplex	185	215149.43	3.06	68.23%	16.90%	1.07%	12.18%	1.63%
453.povray	77	160727052.95	19.25	55.52%	28.64%	1.53%	12.31%	2.00%
444.namd	12	32832.14	1	100%	0	0	0	0
overall	4825	13840607.47	20.15	44.39%	17.76%	2.46%	25.75%	9.64%

and return values, which by passes TypeArmor. The first two arguments of the callsite and the AT function both being 64-bit pointers. The third argument of the callsite is of type `uint32_t` while the third argument in the AT function is an enum which would be translated to a 32-bit value in assembly. Thus, the width of all three arguments of the callsite and the AT function are the same. While for the return value, the callsite expects an integer (32-bit), and the calltarget sets a pointer (64-bit). However, since τCFI always overestimates the width at the calltarget and underestimates it at the callsite, it concludes that the width of the return value is at least 32-bit at the callsite and at most 64-bit at the calltarget. Consequently, it also fails to filter out the illegal control flow by checking the width of the return value. For the similar reason, Lin et al. [23] also cannot defend against this attack.

In contrast, with its runtime review mechanism, TypeSqueezer is capable of thwarting the aforementioned exploit variation. We ran this alternative attack against an Apache server fortified by TypeSqueezer. Specifically, a well known test suite, Apache HTTP Test Project[3], was used as test cases for the hybrid inference of Apache's CFG. In the attack simulation, TypeSqueezer concluded that the second argument of the icall is of type *address reference that indexes another address* and the expected return value is of type *value*, while the second argument of the target function is of type *address reference that indexes value* and its actual return value is of type *address reference*. Consequently, the compromised control flow failed to pass TypeSqueezer's runtime review (i.e., signature mismatch due to value-based type inference only, see §3.4), at which point our approach could effectively notice that the attack intended to abuse the system call `exec()`.

[3]https://httpd.apache.org/test/

```
1  static int
2  autohelperbarrierspat66(int trans, int move, int color, int action)
3  {
4    int a, b;
5    UNUSED(color);
6    UNUSED(action);
7
8    a = AFFINE_TRANSFORM(682, trans, move);   // uses trans, move
9    b = AFFINE_TRANSFORM(719, trans, move);   // uses trans, move
10
11   return play_attack_defend_n(color, 1, 2, a, b, b);    // uses color
12 }
```

Figure 6: Source code of `autohelperbarrierspat66()` from 445.gobmk in SPEC CPU2006.

4.3.2 Limitations of static analysis. Figure 6 shows the C source code of function `autohelperbarrierspat66()` in 445.gobmk of SPEC CPU2006, whose assembly code is earlier shown in Figure 2b. Although the source function defines four arguments in total, only

Table 7: Normalized runtime overhead for SPEC CPU2006 benchmarks.

Benchmark	401.bzip2	458.sjeng	433.milc	482.shpinx3	456.hmmer	464.h264ref	445.gobmk	400.perlbench
Overhead (%)	5.99	5.56	8.20	0.89	6.08	3.30	3.84	0.70
Benchmark	403.gcc	483.xalancbmk	471.omnetpp	447.dealII	473.astar	450.soplex	453.povray	444.namd
Overhead (%)	8.80	4.23	5.96	4.72	4.21	9.02	1.85	5.06

the first three arguments are used. By manually analyzing the source code of 445.gobmk, we found that such behavior is intended. In particular, the function pointer of the indirect callsite that invokes this function would also invoke another function which indeed consumes four arguments. To support flexible programming, i.e., to allow the function pointer to invoke both functions, this function is intentinoally defined to have the same signature as the other. Under such an intended design, static analysis of assembly code can only figure out that the first three arguments are consumed by the function. However, for the remaining arguments, static analysis cannot ensure their usage and it has to conservatively assume that the function has at least three arguments. Consequently, static analysis cannot recover the original function prototype exactly without additional information.

On the other hand, TypeSqueezer is able to address this problem. As described in §2.2, by sharing the static analysis result with other calltargets and callsites that have the same function signature (identified by dynamic analysis), TypeSqueezer is able to reconstruct its source-level arity collectively.

4.4 Performance overhead

To assess potential runtime overhead while testing the compatibility of TypeSqueezer, we used our approach to fortify the SPEC CPU2006 benchmarks used in the effectiveness evaluation, and tested their hardened versions using all available inputs (test, train, and reference). The normalized runtime overhead for each tested benchmark (compared to vanilla binary) is presented in Table 7. The results show that the hardened binary introduces an average runtime overhead of approximately 4.90%. All the fortified benchmark programs finished execution successfully, indicating that the CFI instrumentation of TypeSqueezer upholds compatibility.

Note that static binary rewriting may suffer from a performance trade-off due to the requirement of maintaining original program semantics (e.g., Dyninst utilizes trampolines to insert code snippets). Further performance optimization of our approach could be achieved by adopting reassembly-based instrumentation technique (like Uroboros [39] and Ramblr [38]) which allows direct insertion of CFI checks. Also note that the runtime review mechanism of TypeSqueezer is supposed to be rarely (if not never) triggered during normal executions, thus the overhead caused by that component is negligible.

5 DISCUSSION

First, it is worth pointing out that TypeSqueezer is not intended to defend against all types of low-level attacks. To the best of our knowledge, even the finest-grained source-level CFI systems cannot promise to stop all potential attacks. For example, Farkhani et al. [14] demonstrate the feasibility of bypassing type-based CFI solutions in a large code base. Further, data-only attacks [10, 17] are covered under TypeSqueezer since such attacks will not divert a program's control flow. Nevertheless, by effectively refining indirect call targets, TypeSqueezer significantly raises the bar for adversaries to launch control-flow hijacking attacks for binary executables.

Second, as a hybrid analysis approach, TypeSqueezer's effectiveness relies on the completeness of test cases. Intuitively, and as also demonstrated in our evaluation, higher binary code coverage allows its dynamic analysis to include more signatures into the clusters, and consequently leads to better CFG accuracy. However, note that TypeSqueezer is orthogonal to approaches aiming to improve code coverage of software analysis. Therefore, techniques such as symbolic execution [7, 21], fuzzing [15, 33, 42], and hybrid fuzzing [19, 41] that can enhance code coverage may further augment the effectiveness of TypeSqueezer.

Third, to be conservative, TypeSqueezer only examines the first 8 bytes starting from the memory location pointed to by an address reference because locations farther away may belong to different data structures. Also, TypeSqueezer may not be able to distinguish different pointer-to-pointer types (e.g., int ** and char **), as reconstructing source-level types from binary-level information is undecidable. Nevertheless, we stress that the key motivation of TypeSqueezer is to distinguish function signatures to the best extent possible given the limitations of the binary-level analysis.

Fourth, as discussed in §4.2.3, TypeSqueezer could potentially make type inferences that are inconsistent with the corresponding source-level declarations but in fact captures correct runtime semantics regarding the arguments' usage. We argue that CFI enforcement is in fact dynamic in nature; therefore, when there's a discrepancy between static and dynamic signatures (both being correct from their own perspectives), it makes more sense to adopt the dynamic version to avoid false positives/negatives in detection.

Last but not least, unlike BPA [20], TypeSqueezer does not consider unreachable icalls in its analysis. Excluding the unreachable icalls may provide stronger guarantees against control-flow hijacking attacks. For example, some programs like the 401.bzip2 benchmark of SPEC intentionally modify their code to use only a portion of their functionality for performance measurement purposes. As a result, certain code snippets in these programs may be unreachable. BPA's points-to analysis identifies dead code if it is unreachable from the program entry point, and the dead icall targets are assigned empty sets. TypeSqueezer could potentially adopt similar techniques, such as identifying dead code after computing the targets of each indirect call. Subsequently, the indirect calls residing there can be marked as unreachable and their targets will be set to empty, which further refines the precision of the CFG. It remains an open question to efficiently and effectively identify unreachable icalls for programs with non-trivial amounts of dead code, and we leave it as our future work.

6 RELATED WORK

CFG Generation from Source Code. Intuitively, source-level solutions construct CFGs with the best possible accuracy due to the rich supportive information available in the program source code. Existing approaches of this flavor [22, 25, 29, 36] typically work by matching prototypes of icalls with those of AT functions, so that any icall would only be allowed to transfer control to AT functions that have the same prototype signature. Specifically, IFCC [36] and MCFI [29] resolve icall targets by source-level arity-matching and signature-matching separately. On top of this, advanced source-level CFI solutions [22, 25] also attempt to refine the generated CFGs by further enriching the function prototype signatures, which is achieved by retrieving multi-layer types of icalls and AT functions.

Although type-based matching at the source level can provide the theoretically best results for indirect CFG inference, the applicability of such approaches is severely limited since they cannot process COTS binaries. Compared to source-level solutions, TypeSqueezer recovers fine-grained function signatures for binary executables and still yields comparable accuracy in CFGs recovered.

CFG Generation from Binary Executables. On the other hand, binary-level solutions construct CFGs from executables. While this is hard in practice due to the information loss during the compiling (e.g., type information), it is still crucial for protecting COTS and legacy binaries whose source code is unavailable. Specifically, the coarse-grained solutions [45, 46] allow icalls to target any AT functions. However, they leave too much wiggle room for attackers and thus are vulnerable [16]. Fine-grained approaches [20, 23, 28, 37] enforce stricter policies. Among them, TypeArmor [37] and τCFI [28] try to approximate coarse-grained function signatures and then compute icall targets via type-based matching at different gratitude. In particular, TypeArmor determines the upper or lower bound of arity for a given icall/AT function. τCFI uses a similar strategy to approximate argument widths to represent the actual argument type. Further, Lin et al. [23] present detailed discussions on the impact of compiler optimizations for binary-level function signature recovery techniques and propose countermeasures to deal with them. From another perspective, on top of VSA [2], which is a binary-level points-to analysis framework, BPA [20] assumes a novel block-memory model that extends the scalability of VSA, which enables it to analyze large and complex 32-bit binaries. Lockdown [31] restricts control flows between different code models (e.g., executables and shared objects), and a cross-module indirect call is only allowed to target functions that are present in imported symbols, which can be found in the dynamic symbol table of the corresponding model. Zeng et al. [43] propose to infer function signatures based on debug information for binary executables. In contrast to all of the above approaches that rely on static analysis, TypeSqueezer leverages hybrid analysis to compensate for the inherent imprecision of static analysis while embracing the incompleteness of dynamic analysis.

In addition, several binary-level approaches [30, 32, 44] are proposed to protect icalls that are derived from C++ virtual calls. Specifically, VTint [44] enforces virtual table integrity to defeat virtual table injection attacks. vfGuard [32] only allows virtual icalls to invoke virtual functions that have the same offset. Besides, Marx [30] reconstructs C++ class hierarchy from binaries and enforces stricter protection for virtual calls by considering class hierarchy. However, only protecting virtual calls is incomplete for binary executables. In contrast, TypeSqueezer can be used for a complete CFI protection for all indirect callsites. Moreover, the advancement in the CFG construction of C++ virtual calls can be integrated into TypeSqueezer.

Machine learning techniques have also been applied in the field of binary analysis. For example, EKLAVYA [11] employs recurrent neural networks (RNNs) to deduce function signatures autonomously, without any prerequisite information. On top of it, ReSIL [24] integrates compiler-optimization-specific domain knowledge to enhance function signature recovery. Callee [47] further leverages deep neural networks (DNNs) to reconstruct call graphs within a given binary. Notably, it utilizes contrastive learning to address queries about callsite and calltarget pairs and facilitate indirect call target resolving. It further leverages transfer learning to draw training data from direct calls. Nonetheless, owing to the ongoing challenge of interpreting neural networks, the applicability of the aforementioned approaches is constrained. Specifically, they may not be suited for scenarios such as CFI and binary rewriting, where an unsound CFG could jeopardize program functionality.

7 CONCLUSION

In this paper, we present TypeSqueezer, a retrofitted binary-level indirect call target resolving technique that leverages a hybrid style type-based matching. By introducing dynamic analysis to compensate for static analysis, TypeSqueezer overcomes the inherent limitation of static type-based matching approaches and reconstructs fine-grained function signatures for binary executables. We evaluate TypeSqueezer with the SPEC CPU2006 benchmarks and a number of real-world applications. The results demonstrate that TypeSqueezer produces the most accurate CFG that supersedes the state-of-the-art type-based CFG construction techniques at the binary level.

ACKNOWLEDGEMENT

We would like to thank all the anonymous reviewers sincerely for their valuable comments. Those comments helped us improve our paper. This work is partially supported by the National Natural Science Foundation of China (Key Program Grant No. 62232013).

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, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, Vijay Atluri, Catherine A. Meadows, and Ari Juels (Eds.). ACM, 340–353. https://doi.org/10.1145/1102120.1102165

[2] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *Compiler Construction: 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 13*. Springer, 5–23.

[3] Andrew R Bernat and Barton P Miller. 2011. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. 9–16.

[4] 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*. 30–40.

[5] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 15th ACM conference on Computer and communications security*. 27–38.

[6] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 1–33.

[7] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[8] Nicholas Carlini, Antonio Barresi, Mathias Payer, David A. Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, Jaeyeon Jung and Thorsten Holz (Eds.). USENIX Association, 161–176. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[9] 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*. 559–572.

[10] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-control-data attacks are realistic threats.. In *USENIX security symposium*, Vol. 5. 146.

[11] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium (USENIX Security 17)*. 99–116.

[12] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. 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*. 243–255.

[13] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard E. Shrobe, Martin C. Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 901–913. https://doi.org/10.1145/2810103.2813646

[14] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William K. Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the Effectiveness of Type-based Control Flow Integrity. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 28–39. https://doi.org/10.1145/3274694.3274739

[15] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

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

[17] 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 *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 969–986.

[18] Ralf Hund, Thorsten Holz, and Felix C Freiling. 2009. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *USENIX security symposium*. 383–398.

[19] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel.. In *NDSS*.

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

[21] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[22] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. 2018. Fine-cfi: fine-grained control-flow integrity for operating system kernels. *IEEE Transactions on Information Forensics and Security* 13, 6 (2018), 1535–1550.

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

[24] Yan Lin, Debin Gao, and David Lo. 2022. Resil: Revivifying function signature inference using deep learning with domain-specific knowledge. In *Proceedings of the Twelfth ACM Conference on Data and Application Security and Privacy*. 107–118.

[25] Kangjie Lu and Hong Hu. 2019. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1867–1881.

[26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[27] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2012. The System V ABI. https://refspecs.linuxbase.org/elf/x86_64-abi-0.99.pdf. Accessed: 2023-03-29.

[28] Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. 2018. τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings*. Springer, 423–444.

[29] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 577–587.

[30] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. MARX: Uncovering Class Hierarchies in C++ Programs.. In *NDSS*.

[31] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9148)*, Magnus Almgren, Vincenzo Gulisano, and Federico Maggi (Eds.). Springer, 144–164. https://doi.org/10.1007/978-3-319-20550-2_8

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

[33] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing.. In *NDSS*, Vol. 17. 1–14.

[34] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. IEEE Computer Society, 745–762. https://doi.org/10.1109/SP.2015.51

[35] 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*. 552–561.

[36] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC and LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*. 941–955.

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

[38] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again.. In *NDSS*.

[39] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*. 627–642.

[40] Rafal Wojtczuk. 2001. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e* 70 (2001), 227–242.

[41] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

[42] Michał Zalewski. 2016. https://lcamtuf.coredump.cx/afl/technical_details.txt. Accessed: 2023-03-29.

[43] Dongrui Zeng and Gang Tan. 2018. From Debugging-Information Based Binary-Level Type Inference to CFG Generation. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*, Ziming Zhao, Gail-Joon Ahn, Ram Krishnan, and Gabriel Ghinita (Eds.). ACM, 366–376. https://doi.org/10.1145/3176258.3176309

[44] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting Virtual Function Tables' Integrity.. In *NDSS*.

[45] 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 *2013 IEEE Symposium on Security and Privacy*. IEEE, 559–573.

[46] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, Samuel T. King (Ed.). USENIX Association, 337–352. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang

[47] Wenyu Zhu, Zhiyao Feng, Zihan Zhang, Jianjun Chen, Zhijian Ou, Min Yang, and Chao Zhang. 2023. Callee: Recovering call graphs for binaries with transfer and contrastive learning. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2357–2374.