OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary

Zhuo Zhang[§], Yapeng Ye[§], Wei You[†*], Guanhong Tao[§], Wen-chuan Lee[§],
Yonghwi Kwon[‡], Yousra Aafer[‖], Xiangyu Zhang[§]
[§]Purdue University, [†]Renmin University of China, [‡]University of Virginia, [‖]University of Waterloo
{zhan3299, ye203, taog, lee1938, xyzhang}@purdue.edu,
youwei@ruc.edu.cn, yongkwon@virginia.edu, yousra.aafer@uwaterloo.ca

***Abstract*—Recovering variables and data structure information from stripped binary is a prominent challenge in binary program analysis. While various state-of-the-art techniques are effective in specific settings, such effectiveness may not generalize. This is mainly because the problem is inherently uncertain due to the information loss in compilation. Most existing techniques are deterministic and lack a systematic way of handling such uncertainty. We propose a novel probabilistic technique for variable and structure recovery. Random variables are introduced to denote the likelihood of an abstract memory location having various types and structural properties such as being a field of some data structure. These random variables are connected through probabilistic constraints derived through program analysis. Solving these constraints produces the posterior probabilities of the random variables, which essentially denote the recovery results. Our experiments show that our technique substantially outperforms a number of state-of-the-art systems, including IDA, Ghidra, Angr, and Howard. Our case studies demonstrate the recovered information improves binary code hardening and binary decompilation.**

I. INTRODUCTION

A prominent challenge in binary program analysis is to recognize variables, derive their types, and identify complex array and data structure definitions. Such information is lost during compilation, that is, variables and data structure fields are translated to plain registers and memory locations without any structural or type information. Variable accesses, including those for both simple global scalar variables and complex stack/heap data structure fields with a long reference path (e.g., `a.b.c.d`), are often uniformly compiled to dereferences of some registers that hold a computed address. Recovering the missing variable and structure information is of importance for software security. Such information can be used to guide vulnerability detection [1], legacy code hardening (e.g., adding bound checks) [2], [3], [4], [5], executable code patching (i.e., applying an existing security patch to an executable) [6], and decompilation (to understand hidden program behaviors) [7], [8], [9]. It is also a key step in any non-trivial binary rewriting, such as binary debloating to reduce attack surface [10].

Most binary analysis platforms have the functionalities of variable recovery and some support of structure recovery, i.e., array, struct, and class recovery. Many of them, including the most widely used IDA platform [7], hard-code a set of reverse engineering rules that are effective in certain scenarios (e.g., for binaries generated by some compilers). However, they are usually not general enough because modern compilers are diverse and feature aggressive optimizations, which may violate many instruction patterns that these rules rely on. A number of systems, including Ghidra [8], Angr [11], and TIE [12], make use of static program analysis, such as data-flow analysis and abstract interpretation, to identify variables and infer types. However, their underlying static analysis is often not sufficiently accurate. For example, many rely on *Value Set Analysis* (VSA) [13] to derive the points-to relations at the binary level. However, VSA is known to produce a lot of bogus information, reporting many memory accesses potentially aliased with almost the entire address space. Some techniques such as REWARDS [1] and Howard [14] rely on dynamic analysis to achieve better accuracy. They need high quality inputs to reach good coverage. Such inputs may not be feasible in security applications. In addition, as compilation is lossy, variable and structure recovery is inevitably uncertain. Such uncertainty often yields contradicting results. For instance, many techniques rely on specific instruction patterns of loading base address to recognize a data structure. However, such patterns may appear in code snippets that do not access data structure at all (just by chance). Existing techniques lack a systematic way of dealing with such uncertainty.

We observe that there are a large number of hints of various kinds that can be collected to guide variable and structure recovery, many of them have not been fully leveraged by existing techniques, due to both the difficulty of precluding bogus hints and the lack of a systematic way of integrating them in the presence of uncertainty. For example, some of such hints include: two objects of the same class often go through similar data-flow; two objects of the same class may have direct data-flow between their corresponding fields (due to object copying). However, leveraging such hints requires identifying precise data-flow, which is difficult, and aggregating them when there is uncertainty.

In this paper, we propose a probabilistic variable and data structure recovery technique. It extends a recent binary abstract interpretation infrastructure BDA [15] that has better scalability and accuracy, to collect a large set of basic behavioral properties of the subject binary, such as its memory access patterns, data-flow, and points-to relations. For each (abstract)

*Corresponding author

memory location, i.e., a potential variable/data-structure-field, a set of random variables are introduced to denote its possible *primitive types* (e.g., int, long, and pointer) and its *structural properties* (e.g., being a field of some data structure or an element of some array). These random variables are correlated through the hints collected by program analysis. For example, two memory locations may be two elements of a same array if they are accessed by the same instruction. This hint can be encoded as a probabilistic constraint involving the random variables for the two memory locations. Note that although such hints are uncertain, the introduction of random variables and probabilistic constraints naturally models the uncertainty. Intuitively, a random variable may be involved in multiple hints and hence its probability is constrained by all those hints. All these probabilistic constraints are resolved together to derive the posterior distribution. We develop a customized iterative probabilistic constraint solving algorithm. It features the capabilities of handling a large number of random variables, constraints, and the need of updating the constraints on-the-fly (e.g., when disclosing a new array). It also features optimizations that leverage the domain specific modular characteristics of programs.

Our contributions are summarized as follows.

- We propose a novel probabilistic variable and data structure recovery technique that is capable of handling the inherent uncertainty of the problem.
- We develop a set of probabilistic inference rules that are capable of aggregating in-depth program behavioral properties to achieve precision and good coverage in recovery results.
- We develop an iterative and optimized probabilistic constraint solving technique that handles the challenges for probabilistic inference in program analysis context.
- We develop a prototype OSPREY (*recOvery of variable and data Structure by PRobabilistic analysis for strippEd binarY*). We compare its performance with a number of state-of-the-art techniques, including Ghidra, IDA, Angr, and Howard, on two sets of benchmarks collected from the literature [14], [12]. Our results show that OSPREY outperforms them by 20.41%-56.78% in terms of precision and 11.89%-50.62% in terms of recall. For complex variables (arrays and data structures), our improvement is 6.96%-89.05% (precision) and 46.45%-74.02% (recall). We also conduct two case studies: using our recovered information to (1) improve decompilation of IDA and (2) harden stripped binaries.

II. MOTIVATION

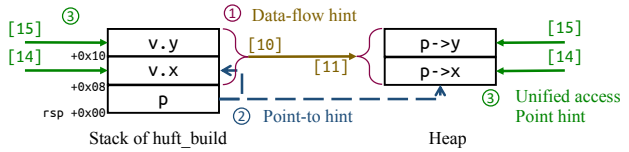
In this section, we use an example to illustrate the limitations of existing techniques and motivate our technique. Figure 1a presents the source code of a function `huft_build` in *gzip* (lines 8-15). It is substantially simplified for the illustration purpose. We also introduce a crafted `main()` function (lines 5-7) which uses a predicate over a random number to represent that the likelihood of reaching the function through random test input generation is low (line 6). Figure 1b presents

```
01. typedef struct {
02.   long x;
03.   long y;
04. } elem_t;
05. int main() {
06.   if (!rand(1000)) huft_build(…);
07. }
08. void  huft_build(…) {
09.   elem_t *p, v = {.x=0, .y=1};
10.   if (…)
11.     p = &v;
12.   else {
13.     p = malloc(sizeof(elem_t));
14.     *p = v;
15.   }
16.   output(p->x, p->y);
17. }
```

(a) Source code of `huft_build`.

```
<huft_build>:
[01] mov [rsp+0x8], 0          line 09
[02] mov [rsp+0x10], 1
[03] ...
[04] lea rbx, [rsp+0x8]        line 11
[05] mov [rsp], rbx
[06] ...
[07] mov rdi, 0x10             line 13
[08] call malloc
[09] mov [rsp], rax
[10] movdqa xmm0, [rsp+0x8]    line 14
[11] movups [rax], xmm0
[12] ...
[13] mov rcx, [rsp]            line 16
[14] mov rdi, [rcx]
[15] mov rsi, [rcx+0x8]
```

(b) Assembly of `huft_build`.



(c) Memory layout of `huft_build`'s variables.

Fig. 1: Motivation example.

the corresponding assemble code, and Figure 1c shows part of the memory layout of the variables. In the source code, lines 1-4 define a structure `elem_t` consisting of two fields `x` and `y`; inside the function, line 9 declares `p` as a pointer to `elem_t`, and `v` as a stack-inlined `elem_t`; the conditional at line 10 has two branches, with the true branch setting `p` to the address of `v` and the false branch allocating a piece of heap memory to `p` (line 13), and storing `v` to the allocated space (line 14); and finally, line 16 outputs `p->x` and `p->y`.

After compilation, global variables are denoted by constant addresses and local variables are translated to offsets on stack frames. For example, the definitions of `v.x` and `v.y` at line 9 are translated to memory writes to stack offsets `rsp+0x8` and `rsp+0x10` (instructions `[01]`-`[02]` in Figure 1b, respectively. The assignment to `p` at line 11 is translated to a write to offset `rsp+0x0` at instruction `[5]` in Figure 1b. This is due to the stack memory layout shown on the left of Figure 1c. Observe that from the assembly code the types of these stack offsets are unknown. It is also unclear `rsp+0x8` and `rsp+0x10` belong to a data structure while `rsp` denotes an 8-byte scalar variable. It is almost impossible to know that the heap variable stored in register `rax` at instruction `[09]` is of the same type as the data structure denoted by `rsp+0x8` and `rsp+0x10`. This example only represents some simple situations. In practice, there are much more difficult challenges such as nesting structures, array of structures, and arrays inside structures. In the following, we discuss how the-state-of-the-art techniques and our technique perform on this example. Note that the ideal recovery result is to identify `p` as a pointer to `elem_t` while `v` is an instance of the same structure on stack, as shown in the "ground truth" column in Figure 2.

IDA [7] is one of the most widely-used commercial decompilation toolkits. It has the functionality of recovering variables and their types. Its recovery algorithm, which is called *semi-naive* algorithm in [16], is based on a local (intra-procedural) static analysis. It identifies absolute addresses, `rsp`-based

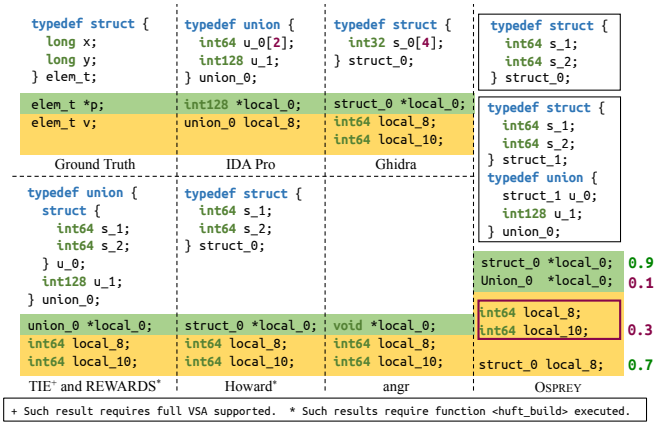


Fig. 2: Results of different techniques for `huft_build`.

offsets, and `rbp`-based offsets as variables or data structure fields. For example, it recognizes `rsp+0x8` (at instruction `[01]`) as a variable/field. In order to distinguish data structure fields from scalar variables, IDA developers hard-coded a number of code pattern matching rules. For example, they consider field accesses are performed by first loading the base address of the data structure to a register, and then adding the field offset to the register. As such, they consider all the accessed addresses that share the same base belong to a data structure. Another sample rule is that an instruction pair like the `movdqa` instruction at `[10]` and the `movups` instruction at `[11]` denotes a 128-bit packed floating-point value movement. Unfortunately, modern compilers aggressively utilize these instruction patterns to optimize code generation. In our case, the two instructions are not related to floating-point value copy but rather general data movement. As shown in Figure 2, IDA misidentifies `elem_t` as a union (denoted as `union_0`) of a 64-bit value array of size two, and a monolithic field of 128-bit. The data structure is recognized through the `lea` instruction at `[04]`, which loads the base address `rsp+0x8`. However, since `rsp+0x8` and `rsp+0x10` are accessed in two manners, one accessing individual addresses as instructions `[01]` and `[02]`, and the other accessing the region as a whole like instructions `[10]` and `[11]`, IDA determines that it is a union. Also observe that IDA fails to recognize that variable `local_0` (i.e., the local variable at stack offset 0 corresponding to `p` in the source code) is a pointer to the data structure. In our experiment over 101 programs (Section VI), IDA achieves 66.88% precision and 76.29% recall.

Ghidra [8] is a state-of-the-art decompiler developed by NSA. Its algorithm is similar to IDA's. The improvement is that Ghidra leverages a register-based data-flow analysis [17] to analyze potential base addresses that are beyond `rsp` and `rbp` registers. In our example, it identifies `rax` at instruction `[09]` denotes the base address of the allocated heap structure at `[08]` as the return value of `malloc` at `[08]` is implicitly stored in `rax`. This allows Ghidra to identify `local_0` (i.e., `rsp`) as a pointer to the heap data structure as shown in the "Ghidra" column in Figure 2. However, the data-flow analysis is limited. It does not reason about data flow through memory. Observe that the base address in `rax` is stored to `[rsp]` at instruction `[09]` and then loaded to `rcx` at `[13]`. Ghidra cannot recognize `rcx` at `[13]` denotes the same base address as `rax` at `[09]`. As a result, it cannot recognize `local_0` is pointing to the same data structure of the two stack offsets `rsp+0x8` and `rsp+0x10`. Instead, it identifies `local_0` a 32-bit value heap array of size 4 and the two stack offsets as separate scalar variables. Inspection of Ghidra's source code indicates that Ghidra developers do not consider stack offsets as reliable base addresses (potentially due to that compiler optimizations may lead to arbitrary stack addressing) such that it does not even group the two stack offsets to a structure. This demonstrates that the intrinsic uncertainty in variable recovery leads to inevitably ad-hoc solutions. In our experiment, Ghidra achieves 69.77% precision and 76.73% recall.

TIE [12] is a static type inference technique for binary programs. It leverages a heavy-weight abstract interpretation technique called *Value Set Analysis* (VSA) [13] to reason about data-flow through memory. VSA over-approximates the set of values that may be held in registers and memory locations such that a memory read may read the value(s) written by a memory write as long as their address registers' value sets have overlap, meaning that the read and the write may reference the same address. Facilitated by VSA, TIE is able to determine that the access of `[rsp]` at instruction `[13]` may receive its value from the write at instruction `[09]` that represents the allocated heap region. As such, the accesses in instructions `[14]` and `[15]` allow TIE to determine that the heap structure consists of two `int64` fields, as shown in Figure 2. However, VSA is conservative and hence leads to a large amount of bogus data-flow. As such, existing public VSA implementations do not scale to large programs [15], including *gzip*. Besides, the inherent uncertainty in variable recovery and type inference often leads to contradicting results. TIE cannot rule out the bogus results and resorts to a conservative solution of retaining all of them. Assume the underlying VSA scaled to *gzip* and hence TIE could produce results for our sample function `huft_build`. TIE would observe that instructions `[14]` and `[15]` access two `int64` fields inside the heap structure. Meanwhile, it would observe that instruction `[10]` directly accesses a 128 bits value in the same structure. It would consider the structure may contain just a monolithic field. To cope with the contradiction, TIE simply declares a `union` to aggregate the results, as shown in Figure 2. Note that since TIE is not available, in order to produce the presented results, we strictly followed their algorithm in the paper. Finally, as commented by some of the TIE authors in [9], TIE does not support recursive types, although they are widely used (e.g., in linked lists and binary trees). For example, "`struct s {int a; struct s *next}`" would be recovered as "`struct s {int a; void *next}`" at best.

REWARDS [1] is a binary variable recovery and type inference technique based on dynamic analysis. Through dynamic tainting, it precisely tracks data-flow through registers and memory such that base-addresses and field accesses can be rec-

ognized with high accuracy. However, its effectiveness hinges on the availability of high quality inputs, which may not be true in many security applications. Theoretical, one could use fuzzing [18], [19], [20], [21], [22] or symbolic execution [23], [24], [25], [26], [27] to generate such inputs. However, most these techniques are driven by a more-or-less random path exploration algorithm whose goal is to achieve new code coverage. In our example, we use a random function (line 6) to denote the small likelihood of function `huft_build()` being covered by path exploration. If functions, code blocks, and program paths are not covered, the related data-flow and hence the corresponding variable/field accesses cannot be recovered by REWARDS. Similar to TIE, REWARDS cannot deal with uncertainty. In Figure 2, if we assume the function has all its paths covered, REWARDS would generate the same undesirable result as TIE.

Howard [14] improves REWARDS using heuristics to resolve conflicts. As shown in the "Howard" column in Figure 2. It prioritizes complex structures over monolithic fields. However, it cannot recognize structures on stack. More detailed discussion can be found in Appendix A. Howard can achieve 81.5% accuracy, with 59% function coverage.

Angr [11] is a state-of-the-art open source binary analysis infrastructure. Its variable recovery leverages an advanced concolic execution engine. Despite the more precise data-flow analysis, Angr's variable recovery is not as aggressive as the others. Hence, in Figure 2, the current implementation of Angr cannot recognize the structures. More discussion can be found in Appendix A. In our experiment, Angr achieves 33.40% precision and 59.27% recall.

A. Our Technique

Observations. From the above discussion, we observe that compilation and code generation is a lossy procedure, whose reverse function is inherently uncertain. It is hence very difficult to define generally applicable rules to recover variables. In addition, the underlying analysis plays a critical role. These analysis have different trade-offs in accuracy, scalability, and the demand of high quality inputs.

Insights. The first insight is that ***while existing techniques mostly focus on memory access patterns (i.e., base addresses and offset values) to identify structures, there are many other program behaviors that can serve as hints to recover data structures***. For example, they include the following. The first is called *data-flow hint*. In Figure 1c, there is direct data-flow from `v` to `*p`, denoted by the brown arrow ①, due to the copy at instructions `[10]` and `[11]`. It implies that the two memory regions may be of the same complex type. The second kind of hints originates from points-to relations, called *points-to hint*. As blue arrows ② in Figure 1c indicate, variable `p` may point to both `v` and `*p`, suggesting that they are of the same type. The third kind of hint is called *unified access point*. The green arrows ③ mean that instruction `[14]` accesses both `v.x` and `p->x`, while instruction `[15]` accesses both `v.y` and `p->y`. Instructions `[14]` and `[15]` are likely unified access points to fields of the same data structure.

The second insight is that ***the various kinds of hints in variable/structure recovery can be integrated in a more organic manner using probabilistic inference [28]*** . Instead of making a deterministic call of the type of a memory region, depending on the number of hints collected, we compute the probabilities for the memory region having various possible types. This requires developing a set of probabilistic inference rules specific to variable recovery. In our example, the float-point instructions at instructions `[10]` and `[11]` cause a conflict, which is suppressed by the large number of other hints (e.g., ①, ②, and ③ in Figure 1c) in probabilistic analysis.

To realize the above two insights, a critical challenge is to precisely identify data-flow and points-to relations. The recent advance made by BDA [15] makes this feasible.

Our Technique. For each memory location, we introduce multiple random variables to denote the probabilities of possible types of the memory location. We construct the set of possible types and compute the probabilities for these random variables as follows. Specifically, OSPREY extends BDA [15] to compute valuable program properties (introduction to BDA and our extension can be found in Sections III and IV), including memory access patterns, data-flow through register and memory, points-to, heap usage, and so on. These program properties are regarded as basic facts, each of which has a prior probability representing its implication of typing and structural properties. For example in Figure 1c, the *points-to hint* ② that `p` may point to both `v` and `*p` indicates a large prior probability that `v` and `*p` are of the same type. After collecting all the hints with their probabilities, OSPREY performs probabilistic inference to propagate and aggregate these hints, and derive the posterior marginal probabilities that indicate the probable variables, types, and data structure declarations. For instance, in Figure 2, the likelihood of `v` (or `local_8`) being a stack based structure is much higher than that of two separated `int64s` (0.7 v/s 0.3). The likelihood of `p` (or `local_0`) being a pointer to a structure is much higher than being a pointer to a union (0.9 v/s 0.1). This aligns perfectly with the ideal result. Our experiments show that if we only report the most probable ones, our technique can achieve 90.18% precision and 88.62% recall, and 89.05% precision and 74.02% recall for complex variables (e.g., `struct`), substantially outperforming other existing techniques.

III. DESIGN OVERVIEW

Figure 3 shows the workflow of OSPREY. Given a stripped binary, BDA is first used to collect basic analysis facts of the binary (e.g., data-flow and points-to). These basic facts are then first processed by a deterministic reasoning step ②. For example, access/data-flow patterns can be extracted and compared to form hints. The resulted abstract relations/hints then go through the probabilistic constraint construction step ③, where predicates describing structural and type properties of individual memory chunks are introduced (e.g., whether a memory chunk denotes a field starting at some memory

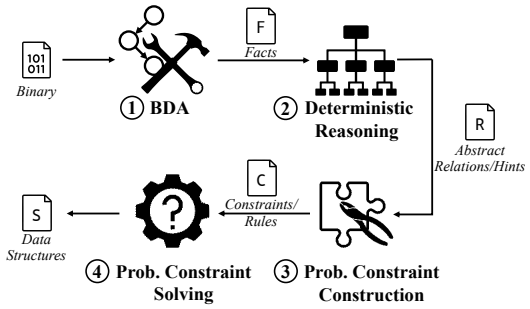


Fig. 3: System design.

address), each denoted by a random variable. Here a memory chunk is a smallest memory unit accessed by some instruction. A set of inference rules are introduced to describe the correlations across these random variables. As such, a random variable is constrained in multiple ways (by various hints). In step ④, these constraints/rules are transformed to a probabilistic graph model. A customized inference algorithm (developed from scratch) is then used to resolve these probabilistic constraints to produce the posterior probabilities. Different from most existing probabilistic inference algorithms, our algorithm is iterative to deal with on-the-fly changes of the constraints, which are inevitable due to the nature of our problem. For example, finding a new likely array leads to introduction of new predicates denoting its properties and requires re-inference. Our algorithm is also optimized as most existing inference engines cannot deal with the large number of random variables in our context. Our optimization leverages the modular characteristics commonly seen in programs and program analysis. Finally, the most probable type and structural predicates are reported and further processed to generate the final variable, type, and structure declarations.

Background: BDA – Path Sampling Driven Per-path Abstract Interpretation. The first step of our technique is to collect basic facts of the subject binary's behaviors. Traditional static analysis such as VSA [13] is not accurate (for larger programs) [15]. Although dynamic analysis are accurate, they need to have good quality input to achieve good coverage. BDA is an advanced static analysis technique that aims to achieve the benefits of both. It uses a sophisticated path sampling algorithm so that the different paths of a program can be sampled uniformly. Note that simply tossing a fair coin at each predicate leads to a distribution that is substantially biased towards short paths. Uniform sampling allows exploring a lot more long paths. For each sampled path, it performs accurate abstract interpretation. As shown in [15], it produces binary points-to results that are substantially better than VSA, leading to much higher accuracy in downstream analysis.

IV. Deterministic Reasoning

Before probabilistic inference, our technique performs deterministic reasoning, through which analysis facts are collected and processed to derive a set of relations and hints. Such information provides the needed abstraction so that the later probabilistic inference, which is sensitive to problem scale, does not have to be performed on the low-level facts.

$f \in \langle \text{Function} \rangle ::= \text{Int}_{64}$ $\quad o \in \langle \text{Offset} \rangle ::= \text{Int}_{64}$
$i \in \langle \text{Instruction} \rangle ::= \text{Int}_{64}$ $\quad k \in \langle \text{Constant} \rangle ::= \text{Int}_{64}$
$s \in \langle \text{Size} \rangle ::= \text{Uint}_{64}$ $\quad r \in \langle \text{MemRegion: MR} \rangle ::= \mathcal{G} | \mathcal{H}_i | \mathcal{S}_f$
$a \in \langle \text{MemAddress: MA} \rangle ::= \langle r, o \rangle$ $\quad v \in \langle \text{MemChunk: MC} \rangle ::= \langle a, s \rangle$

Fig. 4: Definitions.

Definitions. As shown in Figure 4, we use f to denote a function, which is essentially a 64-bit integer denoting the function's entry point, o to denote an offset, i to denote an instruction, which is essentially a 64-bit integer representing the starting address of the instruction, and s to denote a size.

The memory space is partitioned to three distinct regions: *global*, *stack*, and *heap*. The global region, denoted as $\mathcal{G}$, stands for the space holding all the initialized and uninitialized global data. A stack frame or a heap-allocated block constitutes a region as well.

Here, we assume that a binary is correctly disassembled and function entries are properly identified such that the correctness of memory partition can be guaranteed. Although these are very challenging tasks, addressing them is beyond the scope of this paper. As discussed in Section VIII, there are existing techniques [29], [6], [30], [31], [32], [33] that particularly focus on these problems. A stack region for a function f, denoted as $\mathcal{S}_f$, models the stack frame that holds local variables/structures for f. A heap region allocated at an instruction i is denoted as $\mathcal{H}_i$. A memory region r could be any of the three kinds. A memory address a is represented as $\langle r, o \rangle$, in which r stands for the region a belongs to and o for a's offset relative to the base of the region. A *memory chunk*, which is a term we inherit from VSA [13], denotes a variable-like smallest memory unit that is ever visited by some instruction. It is represented as $\langle a, s \rangle$ where a models the starting address of the unit and s its size. It may correspond to a scalar variable, a data structure field, or an array element of some primitive type.

Consider the assemble code at instruction `[11]`, "`movups [rax], xmm0`", in Figure 1b. As register `rax` acquires its value $a = \langle r = \mathcal{H}_{08}, o = 0 \rangle$ from instruction `[08]`, the `movups` instruction accesses a 16-byte variable-like memory chunk $v = \langle a = \langle \mathcal{H}_{08}, 0 \rangle, s = 16 \rangle$.

Primitive Analysis Facts Collected by BDA. As the first step, we extend BDA to collect a set of basic facts. Recall that BDA is a per-path abstract interpretation technique driven by path sampling. It uses precise symbolic values (i.e., without approximation) and interprets individual paths separately. One can consider that BDA is analogous to executing the subject binary on an abstract domain. It does not need to merges values across paths like other abstract interpretation techniques (e.g., VSA), so the abstract domain is precise instead of approximate. We collect six types of facts such as memory access behaviors and points-to relations, as presented in the top of Figure 5. Specifically, $Access(i, v, k)$ [F_{01}] states that instruction i accessed a memory chunk v for k times during the sample runs. By precisely tracking data-flow through both registers and memory, BDA can determine the base address of all offsetting operations. In particular, it looks for data-flow paths that starts by loading an address to a register,

Primitive Analysis Facts

F_{01}	$Access(i, v, k)$	:	Memory chunk v was accessed by instruction i for $k > 0$ times during sampling.
F_{02}	$BaseAddr(i, v, a)$	:	Instruction i has accessed memory chunk v with base address a during sampling.
F_{03}	$MemCopy(v_s, v_d)$	:	The value loaded from v_s was stored to v_d directly, or indirectly via register copying in the middle during sampling.
F_{04}	$PointsTo(v, a)$	:	Memory chunk v stored an address a during sampling.
F_{05}	$MallocedSize(i, s)$	:	The `malloc` function call at instruction i requested s bytes.
F_{06}	$MayArrray(a, k, s)$	:	There may be an array with k elements, each s bytes, starting from address a.

Helper Functions

H_{01}	$SameRegion(a_1, a_2)$: Bool	::=	$a_1.r = a_2.r$	e.g., $SameRegion(\langle \mathcal{S}, 0\rangle, \langle \mathcal{S}, 8\rangle)$: `true`
H_{02}	$Offset(a_1, a_2)$: Size $\cup \{\infty\}$	::=	$SameRegion(a_1, a_2)\ ?\ a_1.o - a_2.o : \infty$	e.g., $Offset(\langle \mathcal{S}, 8\rangle, \langle \mathcal{S}, 0\rangle) = 8$
H_{03}	$AdjacentChunk(v_1, v_2)^*$: Bool	::=	$Offset(v_2.a, v_1.a) = v_1.s$	e.g., $AdjacentChunk(\langle\langle \mathcal{S}, 0\rangle, 8\rangle, \langle\langle \mathcal{S}, 8\rangle, 1\rangle) =$ `true`
H_{04}	$OverlappingChunk(v_1, v_2)^*$: Bool	::=	$Offset(v_2.a, v_1.a) < v_1.s$	e.g., $OverlappingChunk(\langle\langle \mathcal{S}, 0\rangle, 8\rangle, \langle\langle \mathcal{S}, 4\rangle, 1\rangle) =$ `true`
H_{05}	$AddrDifferenceGCD(a_1, a_2, ..., a_n)^+$: Size	::=	$\gcd(\{Offset(a_{k+1}, a_k) \mid 0 \le k < n\})$	e.g., $AddrDifferenceGCD(\langle \mathcal{S}, 0\rangle, \langle \mathcal{S}, 8\rangle, \langle \mathcal{S}, 32\rangle) = 8$
	$SizeDifferenceGCD(s_1, s_2, ..., s_n)^+$: Size	::=	$\gcd(\{s_{k+1} - s_k \mid 0 \le k < n\})$	e.g., $SizeDifferenceGCD(12, 20, 36, 72) = 8$
H_{06}	$MallocedSizes(i)$: $\mathcal{P}$(Size)	::=	$\{s_k \in \text{Size} \mid MallocedSize(i, s_k)\}$ in ascendant order	
H_{07}	$AccessedAddrsInRegion(i, r)$: $\mathcal{P}$(MC)	::=	$\{v.a \in \text{MA} \mid (v.a.r = r) \wedge Access(i, v)\}$	

Deterministic Inference Rules

R_{01}	$Accessed(i, v)$	:-	$Access(i, v, \hat{k})$
R_{02}	$Accessed(v)$	:-	$Accessed(\hat{i}, v)$
R_{03}	$AccessSingleChunk(i, r)$	:-	$\lvert AccessedAddrsInRegion(i, r)\rvert = 1$
R_{04}	$AccessMultiChunks(i, r)$	:-	$\lvert AccessAddrsInRegion(i, r)\rvert > 1$
R_{05}	$HiAddrAccessed(i, r, a_h)$	:-	$a_h = \max(AccessAddrsInRegion(i, r))$
R_{06}	$LoAddrAccessed(i, r, a_l)$	:-	$a_l = \min(AccessedAddrsInRegion(i, r))$
R_{07}	$MostFreqAddrAccessed(i, r, a_f, k)$	:-	$k = \max(\{k_t \mid Access(i, v, k_t)\}) \wedge Access(i, v, k) \wedge v.a = a_f$
R_{08}	$ConstantAllocSize(i, s)$	:-	$(\lvert MallocedSizes(i)\rvert = 1) \wedge (s \in MallocedSizes(i))$
R_{09}	$AllocUnit(i, s)$	:-	$(\lvert MallocedSizes(i)\rvert > 1) \wedge (SizeDifferenceGCD(MallocedSizes(i)) = s)$
R_{10}	$DataFlowHint(a_s, a_d, s)$	:-	$a_s = v_s.a \wedge a_d = v_d.a \wedge (Offset(v'_s.a, a_s) = Offset(v'_d.a, a_d) = s) \wedge$ $SameRegion(a_s, v'_s.a) \wedge SameRegion(a_d, v'_d.a) \wedge MemCopy(v_s, v_d) \wedge MemCopy(v'_s, v'_d)$
R_{11}	$UnifiedAccessPntHint(a_s, a_d, s)$	:-	$a_s = v_s.a \wedge a_d = v_d.a \wedge (Offset(v'_s.a, a_s) = Offset(v'_d.a, a_d) = s) \wedge$ $SameRegion(a_s, v'_s.a) \wedge SameRegion(a_d, v'_d.a) \wedge Accessed(i_1, v_s) \wedge Accessed(i_1, v_d) \wedge$ $Accessed(i_2, v'_s) \wedge Accessed(i_2, v'_d)$
R_{12}	$PointsToHint(a_s, a_d, s)$	:-	$a_s = v_s.a \wedge a_d = v_d.a \wedge (Offset(v'_s.a, a_s) = Offset(v'_d.a, a_d) = s) \wedge$ $SameRegion(a_s, v'_s.a) \wedge SameRegion(a_d, v'_d.a) \wedge BaseAddr(-, v'_s, a_s) \wedge BaseAddr(-, v'_d, a_d)$ $PointsTo(v_x, a_s) \wedge PointsTo(v_x, a_d)$

$^+$Assuming $\forall k \in [0, n)$, $Offset(a_{k+1}, a_k) \ge 0$ and $s_{k+1} - s \ge 0$ without losing generality. *Assuming $Offset(v_2.a, v_1.a) \ge 0$ without losing generality

Fig. 5: Deterministic Reasoning Rules.

which is further copied to other registers or memory chunks, incremented by constant offsets, and eventually dereferenced. $BaseAddr(i, v, a)$ [F_{02}] denotes that i accessed a memory chunk v whose base address is a. $MemCopy(v_s, v_d)$ [F_{03}] states that chunk v_s was copied to v_d. It is abstracted from a data-flow path from a memory read to a memory write, with possible register copies in the middle. $PointsTo(v, a)$ [F_{04}] states that an address value a was ever stored to v. Intuitively, one can consider v a pointer pointing to a. $MallocedSize(i, s)$ [F_{05}] records that a memory allocation function invocation i ever requested size s. $MayArray(a, k, s)$[F_{06}] denotes that a may start an array of k elements, each with size s. Similar to Ghidra and IDA, these array-related hints are collected via heuristics, e.g., by looking at the arguments of `calloc` library call. We will show later that we have more advanced inference rules for arrays. *MayArray* only denotes the direct hints. Examples can be found in Appendix B.

Helper Functions. In the middle of Figure 5, we define a number of helper functions that are derived from the six kinds of basic analysis facts. These helper functions essentially derive aggregated information across a set of primitive analysis facts. They will be used in the inference rules discussed later. Specifically, $SameRegion(a_1, a_2)$ [H_{01}] determines whether two memory addresses belong to the same memory region. Note that in Figure 5, the explanation and example for each helper function are to its right. $Offset(a_1, a_2)$[H_{02}] returns the offset between two memory addresses, which equals to the difference between their offset values if the two addresses belong to the same region, ∞ otherwise. $AdjacentChunk(v_1, v_2)$ [H_{03}] determines if two memory chunks are next to each other. $AddrDifferenceGCD\ (a_1, ..., a_n)$ [H_{05}] returns the *greatest common divisor* (GCD) of the differences of a list of sorted addresses. $SizeDifferenceGCD\ (s_1, ..., s_n)$ [H_{05}] returns the GCD of the differences between a list of sorted sizes. $MallocedSizes(i)$ [H_{06}] returns the list of requested sizes from a malloc-site i. $AccessedAddrsInRegion(i, r)$ [H_{07}] returns all the addresses accessed by i in region r.

Deterministic Inference Rules. The goal of deterministic inference is to derive additional relations that were not explicit. In the lower half of Figure 5, we present the inference rules in the following format.

$$T\ :-\ P_1 \wedge P_2 \wedge \cdots \wedge P_n$$

T is the target relation and P_i is a predicate. It means that the satisfaction of predicates $P_1, P_2, \ldots, P_n$ leads to the introduction of T. Observe that no probabilities are involved.

Specifically, *Accessed*(i, v) [R_{01}] denotes if instruction i has accessed memory chunk v and *Accessed*(v) [R_{02}] denotes if v has been accessed. They are derived from the primitive fact *Access(...)* [F_{01}]. The next two relations model the access pattern of instruction i in memory region r. *AccessSingleChunk*(i, r) [R_{03}] denotes that instruction i is always accessing only one memory chunk in region r. A typical example is an instruction writing to a constant address, e.g., instruction "`mov [0xdeadbeef],0`". *AccessMultiChunks*(i, r) [R_{04}], in contrast, denotes i accessed multiple chunks in r, such as an instruction in some for-loop that accesses individual elements in a memory buffer. *HiAddrAccessed*(i, r, a_h)[R_{05}] dictates that a_h is the highest address in r accessed by i. *LoAddrAccessed*(i, r, a_l)[R_{06}] is the inverse. *MostFreqAddrAccessed*(i, r, a_f, k) [R_{07}] denotes a_f is the most frequently accessed address in r by i.

The next two rules describe the allocation patterns. *ConstantAllocSize*(i, s) [R_{08}] denotes that i has only requested one size s. *AllocUnit*(i, s) [R_{09}] determines if i allocated memory of different sizes and the differences are all multiples of s.

The next three rules describe the three kinds of hints (Section II). *DataFlowHint*(a_s, a_d, s) [R_{10}] suggests the presence of structure if there are copies from two addresses separated by an offset (e.g., two fields from a structure) to two other respective addresses separated by the same offset. Formally, it renders true if given two addresses a_s and a_d, there are two other addresses (denoted by $v'_s.a$ and $v'_d.a$) that have the same offset from a_s and a_d, respectively, such that there are memory copies from a_s to a_d and $v'_s.a$ to $v'_d.a$. Here a_s and a_d denote two instances of the same structure. *UnifiedAccessPntHint*(a_s, a_d, s) [R_{11}] suggests the presence of structure if two addresses (i.e., denoting the same field from two instances of the same structure) are accessed by a same instruction i_1 and their offsets are also accessed by another same instruction i_2. Formally, it renders true given two addresses a_s and a_d, there are two other addresses (denoted by $v'_s.a$ and $v'_d.a$) that have the same offset from a_s and a_d, respectively, such that a_s and a_d are accessed by an instruction i_1 and $v'_s.a$ and $v'_d.a$ accessed by another instruction i_2. *PointsToHint*(a_s, a_d, s) [R_{12}] determines a_s and a_d may denote two instances of the same structure if a_s and a_d are two base addresses for two other addresses that have the same s offset from the base, and both a_s and a_d are stored to the same pointer variable. Appendix C presents an example for deterministic inference.

V. Probabilistic Reasoning

As discussed in Section II, variable and structure recovery is a process with inherent uncertainty such that the collected hints may have contradictions due to: (1) the behavior patterns defining hints may happen by chance, instead of reflecting the internal structure; (2) BDA's per-path interpretation may not respect path feasibility such that infeasible behaviors may be included in the deterministic reasoning step. For example, violations of path feasibility may lead to out-of-bound buffer accesses and then bogus data-flow hints. We resort to probabilistic inference to resolve such contradictions. Intuitively, the effects of incorrect hints will be suppressed by the correct ones which are dominant. In particular, for each memory chunk v, we introduce a number of random variables to describe the type and structural properties of v. The random variables of multiple memory chunks are hence connected through the relations derived from the previous deterministic reasoning step and represented as a set of probabilistic inference rules. Each rule can be considered a probability function. They are transformed to a probabilistic graph model [34] and an inference algorithm is used to compute the posterior marginal probabilities. The most probable results are reported. Different from many existing probabilistic inference applications, where the set of inference rules are static, we have *dynamic inference rules*, meaning that rules will be updated, removed, and added on the fly based on the inference results. We hence develop an iterative and optimized inference algorithm (Section V-B).

A. Probabilistic Inference Rules

Predicates and Random Variables. Figure 6 presents the set of predicates we introduce. They denote the typing and structural properties. Random variables are introduced to denote their instantiations on individual instructions and memory chunks, each describing the likelihood of the predicate being true. For instance, The random variable for *Scalar*$(\langle\langle G, 0x8043abf0\rangle, 8\rangle)$ denotes the likelihood that the 8-byte global memory chunk starting at $0x8043abf0$ is a scalar variable. In the remainder of the paper, we will use the two terms predicate and random variable interchangeably. Specifically, *PrimitiveVar*(v)[P_{01}] asserts that memory chunk v denotes a primitive variable, which is a variable without further inner structure. It could be a scalar variable, a structure field, or a primitive array element. Similarly, *PrimitiveAccess*(i, v)[P_{02}] asserts that instruction i exclusively accesses a primitive variable v. The meanings of *UnfoldableHeap* and *FoldableHeap* will be explained in the later discussion of heap structure recovery. *HomoSegment*(a_1, a_2, s) [P_{05}] asserts that the memory region $a_1 \sim (a_1 + s)$ and $a_2 \sim (a_2 + s)$ are homomorphic, hence likely two instances of the same structure. They are likely homomorphic when their access patterns and data-flow are similar. *ArrayStart*(a) [P_{06}] represents a is the starting address of an array.

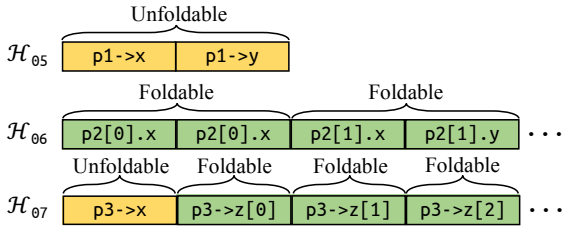
While the above predicates are auxiliary, the remaining ones (underlined in Figure 6) denote our final outcomes. Variables, structures and types can be directly derived from the inferred values of these predicates. In particular, *Scalar*(v)[P_{07}] indicates v is a scalar variable (not an array or a structure). *Array*(a_1, a_2, s)[P_{08}] represents that the memory region from a_1 to a_2 form an array of size s. *FieldOf*(v, a) [P_{09}] asserts that v is a field of a structure starting at a. *Pointer*(v, a) [P_{10}] asserts v is a pointer to a structure starting at a. The last few predicates assert the primitive types of variables. Note that they allow us to express the most commonly seen structural properties, including nesting structures, array of structures,

P_{01}	*PrimitiveVar*(v)	:	v is of primitive type, e.g., `char`, `int`, and `void *`
P_{02}	*PrimitiveAccess*(i, v)	:	Instruction i accessed a primitive variable v
P_{03}	*UnfoldableHeap*(i, s)	:	The size of the unfoldable part of heap structure allocated at i is s
P_{04}	*FoldableHeap*(i, s)	:	The unit size of the foldable part of heap structure allocated at i is s
P_{05}	*HomoSegment*(a_1, a_2, s)	:	The two s-byte segments starting at a_1 and a_2, respectively, are homomorphic
P_{06}	*ArrayStart*(a)	:	Address a is the starting address of an array
P_{07}	*Scalar*(v)	:	Variable v is a scalar
P_{08}	*Array*(a_1, a_2, s)	:	Memory from a_1 to a_2 belongs to an array whose element size is s-byte
P_{09}	*FieldOf*(v, a)	:	Variable v is a field of a structure with starting address a
P_{10}	*Pointer*(v, a)	:	Variable v is a pointer pointing to a structure denoted by a
P_{11}	*IntVar*(v) / *LongVar*(v) / ...	:	Variable v is of the int / long /... type

Fig. 6: Predicate definitions.

```
01. typedef struct { long x; long y; } A;
02. typedef struct { long x; long z[]; } B;
03.
04. void heap_example(size_t n, size_t m) {
05.   A *p1 = malloc(sizeof(A));
06.   A *p2 = malloc(sizeof(A) * n);
07.   B *p3 = malloc(sizeof(B) + sizeof(long [m]));
08.   ...
```

(a) Source code.



(b) Memory layout of code in Figure 7a.

Fig. 7: Example to demonstrate our heap model.

and structure with array field(s). We have other predicates for unions. They are elided for discussion simplicity.

Example. Consider an example in Figure 7, with the source code in Figure 7a. Three types of structures are allocated on the heap. Line 5 allocates a singleton structure (`*p1`); line 6 allocates an array of the same structure (`*p2`); and line 7 allocates a structure (`*p3`) with an array. Note that the size of `*p3` is not fixed. These structures can be easily represented by our predicates. Particularly, the structure of `*p1` is represented as *FieldOf*(`&(p1->x)`, `p1`), *FieldOf*(`&(p1->y)`, `p1`), *Long*(`&(p1->x)`), and *Long*(`&(p1->y)`) (note that the syntax of these predicates is simplified for illustration); `*p2` is represented as *Array*(`p2`, `p2+16*n`, 16), *FieldOf*(`&(p2->x)`, `p2`) and so on (similar to `*p1`); `*p3` is represented as *FieldOf*(`&(p3->x)`, `p3`), *Array*(`&(p3->z)`, `&(p3->z)+16*m`, 16), *Long*(`&(p3->x)`), and *Long*(`&(p3->z)`). □

Figure 8 presents the probabilistic inference rules. These rules read as follows: the first column is the rule id for easy reference; the second column is the condition that needs to be satisfied in order to introduce the inference rule in the third column. Each inference rule is a first-order logic formula annotated with prior probability. Each predicate instantiation is associated with a random variable whose posterior probability will be computed by inference. For example, rule C_{A01} means that v has $p(k)$ probability of being a primitive variable if an instruction i has accessed it k times (over all the sample runs). Note that the probability is a function of k. The up-arrow denotes that if *Access*(i,v,k) is likely, then *PrimitiveVar*(v) is likely. A down-arrow denotes the opposite.

Primitive Variable and Scalar Variable Recovery. Rules C_{A01}-C_{A05} are to identify primitive variables. Rule C_{A02} means that a variable is likely primitive if its adjacent one is likely primitive; C_{A03} means that if two variables have overlapping address, one likely being primitive renders the other one unlikely (note the down-arrow); C_{A04} and C_{A05} state that if a variable v is primitive, the instruction that accesses it is a primitive access such that another variable v' accessed by it is primitive too. Rules C_{A06}-C_{A08} are for scalar variable recovery. A primitive variable may not be a scalar variable as it could be a field or an array element. C_{A06} says v is scalar if it is primitive and there is an instruction i that exclusively accesses it. Intuitively, if i accesses (non-scalar) array elements or structure fields, it likely accesses multiple memory chunks. C_{A07} says a scalar's neighbor may be a scalar too, depending on their access frequencies (e.g., when the frequencies are similar). C_{A08} says a scalar variable cannot be a field.

Array Recovery. Rules C_{B01}-C_{B09} are for array recovery. A common observation is that, the vast majority of arrays are visited in loops. If multiple elements on a continuous region are accessed by an instruction, intuitively, it's likely that this is access to an array. In particular, rules C_{B01}-C_{B02} receive the basic array hints from the previous analysis steps; C_{B03}-C_{B06} aggregate hints to enhance confidence and/or derive new arrays; and C_{B07}-C_{B09} derive array heads. Intuitively, C_{B01} states that there is likely an array if our deterministic reasoning says so (e.g., by observing `calloc`). C_{B02} says if addresses are accessed by the same instruction, there is likely an array and the lowest and highest addresses accessed by the instruction form the lower and upper bounds of an array, respectively. C_{B03} says that when two arrays overlap, have the same element size s and the distance of the two arrays is divisible by s, the two arrays can enhance each other's confidence (the first formula) and they can be merged to a larger array (the second formula). C_{B04} says that when two

ID	Condition	Probabilistic Constraint
C_{A01}		$\mathit{Access}(i, v, k) \xrightarrow{p(k)\uparrow} \mathit{PrimitiveVar}(v)$
C_{A02}	$\mathit{AdjacentChunk}(v_1, v_2) \wedge \mathit{Accessed}(v_1) \wedge \mathit{Accessed}(v_2)$,	$\mathit{PrimitiveVar}(v_1) \xleftrightarrow{p\uparrow} \mathit{PrimitiveVar}(v_2)$
C_{A03}	$\mathit{OverlappingChunk}(v_1, v_2) \wedge \mathit{Accessed}(v_1) \wedge \mathit{Accessed}(v_2)$,	$\mathit{PrimitiveVar}(v_1) \xleftrightarrow{p\downarrow} \mathit{PrimitiveVar}(v_2)$
C_{A04}	$\mathit{Accessed}(i, v)$,	$\mathit{PrimitiveVar}(v) \xrightarrow{p\uparrow} \mathit{PrimitiveAccess}(i, v)$
C_{A05}	$\mathit{Accessed}(i, v')$,	$\mathit{PrimitiveAccess}(i, v) \xrightarrow{p\uparrow} \mathit{PrimitiveVar}(v')$
C_{A06}	$\mathit{AccessSingleChunk}(i, v.a.r) \wedge \mathit{Access}(i, v, k)$	$\mathit{PrimitiveAccess}(i, v) \xrightarrow{p(k)\uparrow} \mathit{Scalar}(v)$
C_{A07}	$\mathit{AdjacentChunk}(v_1, v_2) \wedge \mathit{Access}(i_1, v_1, k_1) \wedge \mathit{Access}(i_2, v_2, k_2) \wedge$ $\mathit{AccessSingleChunk}(i_1, v_1.a.r) \wedge \mathit{AccessSingleChunk}(i_2, v_2.a.r)$	$\mathit{Scalar}(v_1) \xleftrightarrow{p(k_1,k_2)\uparrow} \mathit{Scalar}(v_2)$
C_{A08}		$\mathit{Scalar}(v) \xleftrightarrow{p(k)\downarrow} \mathit{FieldOf}(v, a)$
C_{B01}		$\mathit{MayArray}(a, k, s) \xrightarrow{p\uparrow} \mathit{Array}(a, a + s \times k, s) \wedge \mathit{ArrayStart}(a)$
C_{B02}	$\mathit{AccessMultiChunks}(i, r) \wedge \mathit{LoAddrAccessed}(i, r, v_1.a) \wedge$ $\mathit{HiAddrAccessed}(i, r, v_2.a)$	$\mathit{PrimitiveAccess}(i, v_1) \wedge \mathit{PrimitiveAccess}(i, v_2) \xrightarrow{p\uparrow}$ $\mathit{Array}(v_1.a, v_2.a+v_2.s, v_1.s)$
C_{B03}	$(a_{1l} \leq a_{2l} \leq a_{1h} \leq a_{2h}) \wedge (s_1 = s_2 = s) \wedge (s \mid a_{2l} - a_{1l})$	$\mathit{Array}(a_{1l}, a_{1h}, s_1) \xleftrightarrow{p\uparrow} \mathit{Array}(a_{2l}, a_{2h}, s_2)$
		$\mathit{Array}(a_{1l}, a_{1h}, s_1) \wedge \mathit{Array}(a_{2l}, a_{2h}, s_2) \xrightarrow{p\uparrow} \mathit{Array}(a_{1l}, a_{2h}, s)$
C_{B04}	$(a_{1l} \leq a_{2l} \leq a_{1h} \leq a_{2h}) \wedge$	$\mathit{Array}(a_{1l}, a_{1h}, s_1) \xleftrightarrow{p\downarrow} \mathit{Array}(a_{2l}, a_{2h}, s_2)$
	$((s_1 \neq s_2) \vee ((s_1 = s_2 = s) \wedge (s \nmid a_{2l} - a_{1l})))$	$\mathit{Array}(a_{1l}, a_{1h}, s_1) \xrightarrow{p\uparrow} \mathit{Array}(a_{1l}, a_{2l}, s_1)$
		$\mathit{Array}(a_{2l}, a_{2h}, s_2) \xrightarrow{p\uparrow} \mathit{Array}(a_{1h}, a_{2h}, s_2)$
C_{B05}	$a_1 \leq v.a \leq a_2$	$\mathit{Scalar}(v) \xleftrightarrow{p\downarrow} \mathit{Array}(a_1, a_2, s)$
		$\mathit{Array}(a_1, a_2, s) \wedge \mathit{Scalar}(v) \xrightarrow{p\uparrow} \mathit{Array}(a_1, v.a, s)$
		$\mathit{Array}(a_1, a_2, s) \wedge \mathit{Scalar}(v) \xrightarrow{p\uparrow} \mathit{Array}(v.a+v.s, a_2, s)$
C_{B06}	$a_2 - a_1 < s$	$\mathit{Array}(a_1, a_2, s) = \mathit{false}$
C_{B07}	$\mathit{BaseAddr}(i, v, a) \wedge \mathit{AccessMultiChunks}(i, v.a.r)$	$\mathit{PrimitiveAccess}(i, v) \xrightarrow{p\uparrow} \mathit{ArrayStart}(a)$
C_{B08}	$\mathit{MostFreqAddrAccessed}(i, r, v, k) \wedge \mathit{AccessMultiChunks}(i, r)$	$\mathit{PrimitiveAccess}(i, v) \xrightarrow{p(k)\uparrow} \mathit{ArrayStart}(v.a)$
C_{B09}	$\mathit{Accessed}(i,v_1) \wedge \mathit{Accessed}(i,v_2) \wedge \mathit{SameRegion}(v_1.a,v_2.a) \wedge (v_1.a < v_2.a)$	$\mathit{ArrayStart}(v_1.a) \xrightarrow{p\downarrow} \mathit{ArrayStart}(v_2.a)$
C_{C01}		$\mathit{ConstantAllocSize}(i, s) \xrightarrow{p\uparrow} \mathit{UnfoldableHeap}(i, s) \wedge \mathit{FoldableHeap}(i, 0)$
C_{C02}		$\mathit{AllocUnit}(i, s) \xrightarrow{p\uparrow} \mathit{FoldableHeap}(i, s)$
C_{C03}	$v.a.r = \mathcal{H}_i$	$\mathit{PrimitiveVar}(v) \xrightarrow{p\uparrow} \mathit{UnfoldableHeap}(i, v.a.o + v.s)$
C_{C04}	$s_1 \neq s_2$	$\mathit{UnfoldableHeap}(i, s_1) \xleftrightarrow{p\downarrow} \mathit{UnfoldableHeap}(i, s_2)$
C_{C05}	$s_1 \leq s_2$	$\mathit{UnfoldableHeap}(i, s_1) \xrightarrow{p\uparrow} \mathit{UnfoldableHeap}(i, s_2)$
C_{C06}	$(a_1.r = a_2.r = \mathcal{H}_i) \wedge (s_1 = s_2)$	$\mathit{Array}(a_1, a_2, s_1) \xrightarrow{p\uparrow} \mathit{FoldableHeap}(i, s_2)$
C_{C07}	$\mathit{Accessed}(v) \wedge (v.a.r = \mathcal{H}_i) \wedge (v.a.o \geq s_h + s_t)$	$\mathit{PrimitiveVar}(v) \wedge \mathit{UnfoldableHeap}(i, s_h) \wedge \mathit{FoldableHeap}(i, s_t) \xrightarrow{p\uparrow}$ $\mathit{PrimitiveVar}(\langle \langle v.a.r, (v.a.o - s_h)\%s_t + s_h \rangle, v.s \rangle)$
		$\mathit{UnfoldableHeap}(i, s_h) \wedge \mathit{FoldableHeap}(i, s_t) \xrightarrow{p\downarrow} \mathit{PrimitiveVar}(v)$
C_{D01}		$\mathit{DataFlowHint}(a_1, a_2, s) \xrightarrow{p(s)\uparrow} \mathit{HomoSegment}(a_1, a_2, s)$
C_{D02}		$\mathit{PointsToHint}(a_1, a_2, s) \xrightarrow{p(s)\uparrow} \mathit{HomoSegment}(a_1, a_2, s)$
C_{D03}		$\mathit{UnifiedAccessPntHint}(a_1, a_2, s) \xrightarrow{p(s)\uparrow} \mathit{HomoSegment}(a_1, a_2, s)$
C_{D04}	$(0 < a_2 - a_1 < s_1)$	$\mathit{HomoSegment}(a_1, a_{1'}, s_1) \xleftrightarrow{p\uparrow} \mathit{HomoSegment}(a_2, a_{2'}, s_2)$
		$\mathit{HomoSegment}(a_1, a_{1'}, s_1) \wedge \mathit{HomoSegment}(a_2, a_{2'}, s_2) \xrightarrow{p\uparrow}$ $\mathit{HomoSegment}(a_1, a_{1'}, a_2 - a_l + s_2)$
C_{D05}	$(0 < v_1.a - a_1 = v_2.a - a_2 < s) \wedge (v_1.s \neq v_2.s)$	$\mathit{PrimitiveVar}(v_1) \wedge \mathit{PrimitiveVar}(v_2) \xleftrightarrow{p\downarrow} \mathit{HomoSegment}(a_1, a_2, s)$
C_{D06}	$\mathit{BaseAddr}(v_1, i, v_2.a) \wedge \mathit{Accessed}(v_1) \wedge \mathit{Accessed}(v_2)$	$\mathit{PrimitiveVar}(v_1) \wedge \mathit{PrimitiveVar}(v_2) \xrightarrow{p\uparrow} \mathit{FieldOf}(v_1, v_2.a)$
C_{D07}	$v.a.r = \mathcal{H}_i$	$\mathit{PrimitiveVar}(v) \xrightarrow{p\uparrow} \mathit{FieldOf}(v, \langle \mathcal{H}_i, 0 \rangle)$
C_{D08}	$(n \leq s) \wedge (v_1.a = a_1 + n) \wedge (v_2.a = a_2 + n)$	$\mathit{FieldOf}(v_1, a_1) \wedge \mathit{HomoSegment}(a_1, a_2, s) \xrightarrow{p\uparrow} \mathit{FieldOf}(v_2, a_2)$
C_{D10}	$a_1 \neq a_2$	$\mathit{FieldOf}(v, a_1) \xleftrightarrow{p\downarrow} \mathit{FieldOf}(v, a_2)$
C_{D11}	$\mathit{PointsTo}(v_1, v_2.a) \wedge \mathit{Accessed}(v_1) \wedge \mathit{Accessed}(v_2)$	$\mathit{PrimitiveVar}(v_1) \wedge \mathit{PrimitiveVar}(v_2) \xrightarrow{p\uparrow} \mathit{Pointer}(v_1, v_2.a)$

Fig. 8: Probabilistic Inference

arrays overlap, but they are not homomorphic (e.g. having different element sizes or misalign), one likely being true array renders the other unlikely (the first formula) and the non-overlapping parts can still be considered possible arrays (the second and third formulas). C_{B05} says that a scalar appearing within the range of an array breaks it to two smaller arrays.

Heap Folding. Rules C_{C01}-C_{C07} are auxiliary rules for analysing heap structures. While BDA can achieve alias analysis accuracy similar to dynamic analysis (with better coverage), it leads to sparse heap behaviors. For example, assume a

large heap array of structures is allocated. Different paths may access different heap array elements (at distinct addresses), each disclosing part of the behavior of the structure. Since our goal is to recover the complete structural properties, we need to aggregate these sparse behaviors.

We observe any heap region allocated can be partitioned into two *consecutive* parts: *unfoldable* and *foldable*, while such a region may be a singleton structure with fixed size, an array of structures of a fixed size, or a singleton structure with varying size. The three allocations at lines 5-7 in Figure 7a denote such different cases. *The unfoldable part includes all the fields whose accesses always occur at the same addresses*, whereas *the foldable part includes the fields whose accesses may occur at different (sparse) addresses*. We propose to fold the behaviors of all the instances in the foldable part to the first instance, which will hence possess all the structural properties of all the instances. For example, as shown in Figure 7b, the heap region of $\mathcal{H}_{05}$ has only unfoldable fields as `p1->x` and `p1->y` always have the addresses of $\langle \mathcal{H}_{05}, 0 \rangle$ and $\langle \mathcal{H}_{05}, 16 \rangle$, respectively. In contrast, all fields in the region $\mathcal{H}_{06}$ are foldable as the $p2[*].x$'s have various addresses. We hence want to fold the behaviors of $p2[1]$, $p2[2]$, and so on to $p2[0]$. The region $\mathcal{H}_{07}$ has an unfoldable field followed by a foldable field which is an array of varying size. *Observe that foldable fields can only occur after unfoldable fields in a region*. In Figure 6, we introduce *UnfoldableHeap*(i, s) to denote the first s bytes of the heap region allocated at i are unfoldable and *FoldableHeap*(i, s) to denote the region allocated at i has a foldable part with an element size of s. For example, we have *FoldableHeap*$(7, 16)$ for region $\mathcal{H}_7$ in Figure 7b.

C_{C01} states that if i only allocates a constant size region, the entire region is unfoldable. C_{C02} says that if through deterministic analysis, we know that the allocation size of i is a multiple of s, the foldable part has an element size of s. C_{C03} says that if a primitive field v is found inside a heap region, all the part up to v is unfoldable. This is because unfoldable fields must precede foldable fields. C_{C04} states that a heap region cannot have different unfoldable parts. However, the presence of a smaller unfoldable part can enhance the confidence of a larger unfoldable part (C_{C05}). C_{C06} says that an array found inside a heap region must belong to the foldable part. Rule C_{C07} is the folding rule. The first formula says that a primitive field v found inside a later structure instance inside the foldable region indicates the presence of a primitive field at the corresponding offset inside the first instance. For example in $\mathcal{H}_{06}$, the identification of `y` field in `p2[1]` indicates the presence of `y` field in `p2[0]`, although `p2[0]->y` is never seen during sample runs. The second formula eliminates the primitive field v after it is folded.

Structure Recovery. Like existing work, we leverage the instruction patterns of loading base address to recognize a data structure. However, we model its uncertainty using probabilities. In addition, we consider the data flow among different variables of the same type. Specifically, rules C_{D01}-C_{D10} are for structure recovery, including global/stack/heap structures. Intuitively, we first identify memory segments (i.e., part of a structure) that are homomorphic, meaning that they have highly similar access patterns, data flow, and points-to relations. These segments are then intersected, unioned, or separated to form the final structures. Individual fields can be then identified from their access pattern within the structure. Specifically, rules C_{D01}-C_{D03} receive deterministic hints. C_{D04} states that if a pair of homomorphic segments overlap with another pair of homomorphic segments, they enhance each other's confidence (the first formula) and may form a pair of new homomohpic segments that are the union of the original two pairs (the second formula). Intuitively, it corresponds to that the sub-parts of a same complex structure are being exposed differently (e.g., through different data flow), and we leverage the overlap of these parts to join them. C_{D05} says that if the corresponding primitive fields in a pair of homomorhpic segments have different access patterns (4-byte access versus 8-byte access), either the primitive field predicates are likely false or the homomorphic predicate. Rules C_{D06} and C_{D07} identify fields of structure from the deterministic reasoning results (e.g., *BaseAddr*) and if the accesses are primitive. C_{D08} transfers field information across a pair of homomorphic segments. Rule C_{D09} asserts a field cannot have two different base addresses. C_{D09} determines a pointer variable v_1 if a valid address $v_2.a$ is stored to v_1 and v_2 has been accessed as a primitive variable.

OSPREY also has a set of typing rules that associate primitive types (e.g., int, long, and string) to variables, based on their data-flow to program points that disclose types such as invocations to string library functions. These rules are similar to existing works [1], [12], [14] and hence elided.

B. Probabilistic Constraint Solving

Each of the probabilistic constraints in Figure 8 (the formulas in the last column) essentially denotes a probability function over the random variables involved. The functions can be further transformed to a probabilistic graph model called *factor graph* [34], which is a bi-partite graph with two kinds of nodes, *function node* denoting a probability function, and *variable node* denoting a random variable. Edges are introduced between a function node and all the variable nodes related to the function. The whole factor graph denotes the joint distribution of all the random variables. An example can be found in Appendix E.

Given a set of observations (e.g., $x_1 = 1$) from the deterministic reasoning step, and the prior probabilities (p values), posterior marginal probabilities are computed by propagating and updating probabilities along the edges. Some of the rules, such as C_{B02}, generate new predicate nodes during inference. After each round of inference (i.e., probabilities converge after continuous updates), it checks all the (new) predicate nodes to coalesce those denoting the same meaning to one node. The node inherits all the edges of all the other nodes that are coalesced. Then another round of inference starts. Note that while some probabilistic inference applications are stochastic, our application (variable recovery and typing) has the

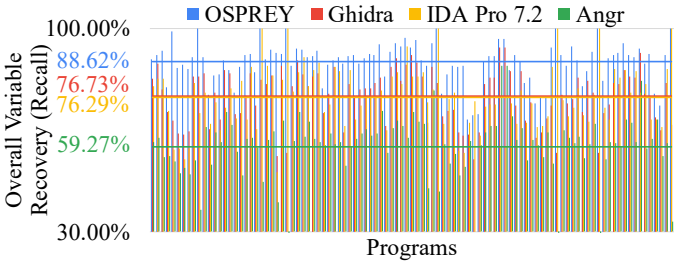


Fig. 9: Recall for all variables (primitive and complex)

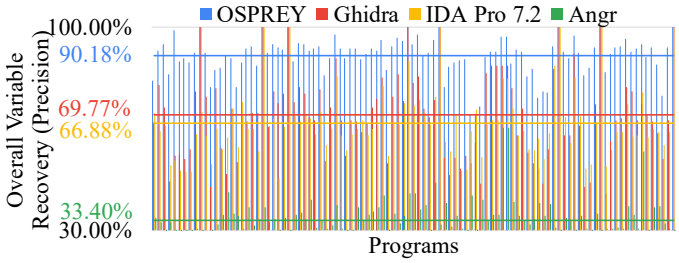


Fig. 10: Precision for all variables (primitive and complex)

uncertainty originating from loss of debugging information. In other words, there is deterministic ground truth (or, the ground-truth variables and their types are deterministic). In this context, the number of hints that we can aggregate plays a more important role than the prior probabilities. Graph models provide a systematic way of aggregating these hints, while respecting the inherent structural properties (e.g., control-flow and data-flow constraints). We hence adopt simple prior probabilities, $p\uparrow = 0.8$, $p\downarrow = 0.2$, and $p(k)$ is computed from the ratio between k and the total number of sampled paths in BDA. In fact, there are a number of existing work [28], [35], [36], [37] leveraging probabilistic inference for similar applications with (mostly) deterministic ground truth (e.g., specification inference for explicit information flow). They use preset prior probabilities and their results are not sensitive to prior probability configurations. We follow a similar setting.

Posterior Probability Computation On Factor Graph. There are standard off-the-shelf algorithms that can compute posterior probabilities for factor graphs. Most of them are message passing based [38], [39], [40], i.e., a function node aggregates probabilities (or *beliefs*) from its neighboring variable nodes, deriving an outgoing belief based on the probability function. Such algorithms become very expensive and have low precision when the graph is large and loopy (as messages are being passed in a circle and computation can hardly converge). There are optimized algorithms such as *junction tree algorithm* [41] that removes cycles by coalescing them to single nodes. However, they do not work well in our context due to the particularly large number of nodes and the extensive presence of loops in our factor graphs. We hence develop an optimized algorithm from scratch, leveraging the modular characteristics of program behaviors. Specifically, we observe that *PrimitiveVar* is the most common kind of node and involved in most constraints. These nodes have very few loops with the other kinds of nodes, although there are loops within themselves. Thus, we first construct a *base graph* only considering *PrimitiveVar*-related rules (i.e., $C_{A01}-C_{A08}$). The *base graph* is still very large (typically around 3000 nodes for even a small program) and cannot be directly solved. We also observe that the memory chunks of the *PrimitiveVar* nodes are distributed in various memory regions that are relatively autonomous (e.g., different stack frames). We hence partition the base graph to many sub-graphs based on memory regions. Empirically, a sub-graph contains 40 nodes on average. Each sub-graph is solved by a junction tree algorithm. With the solved values of all sub-graphs as the initial values, we dynamically construct a secondary graph considering the remaining random variables. Specifically, for any rule, if its pre-condition is satisfied, we include it in the graph. It naturally handles dynamic rule updates. We then resolve the secondary graph using loopy belief propagation [40] which, in general, starts with arbitrary initial values and iteratively updates messages till convergence. Note that we adapt loopy belief propagation by pre-calculating suitable initial values, which does not compromise the original algorithm's correctness. Figure 19 in Appendix presents the statistics of the subgraphs in evaluation.

VI. Evaluation

To assess the effectiveness of OSPREY, we perform two sets of experiments, using the benchmarks from TIE [12] and Howard [14]. The first set is performed on Coreutils [42], a standard benchmark widely used in binary analysis projects [14], [12], [43], [11], [29], consisting of 101 programs. We compare OSPREY with other state-of-the-art binary analysis tools, including Ghidra (version 9.2), Angr (version 8.20) and IDA Pro (version 7.2). We cannot compare with TIE as the system is not available. And we confirmed with the BAP [43] team that BAP does not have TIE as part of it. Another set is performed on the benchmark provided by the Howard project [14], consisting of 5 programs. All experiments were conducted on a server equipped with 48-cores CPU (Intel® Xeon™ Silver 4214 CPU @ 2.20GHz) and 256G main memory. To follow a similar setup in TIE and Howard, we use GCC 4.4 to compile the programs into two versions: a version with debugging information used as the ground truth and a stripped version used for evaluation. Our assumption of proper disassembly is guaranteed because GCC does not interleave code and data on Linux [44].

A. Evaluation on Coreutils

Similar to the standard in the literature [12], [14], we inspect individual variables on the stacks and heaps (including structure types). If it is a pointer type, we inspect the structure that is being pointed to. For example, if a `(Socket *)` variable is recovered as `(void*)`, we consider it incorrect. We say it is correct only if the variable is recovered as a pointer pointing to a structure homomorphic to `Socket`. We only consider the functions covered by BDA . The overall recall and precision are shown in Figure 9 and Figure 10, respectively. As we can see, OSPREY achieves more than 88% recall, and more than 90% precision, outperforming the best of other tools (i.e., Ghidra with around 77% recall and 70% precision). Figures

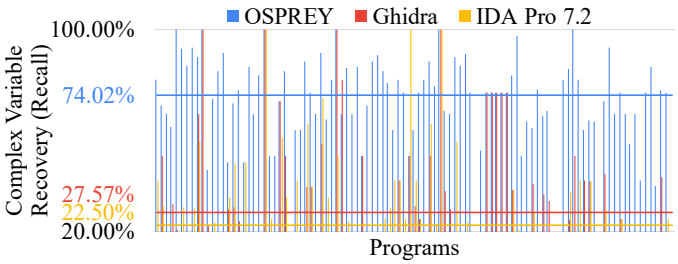


Fig. 11: Recall for complex variables

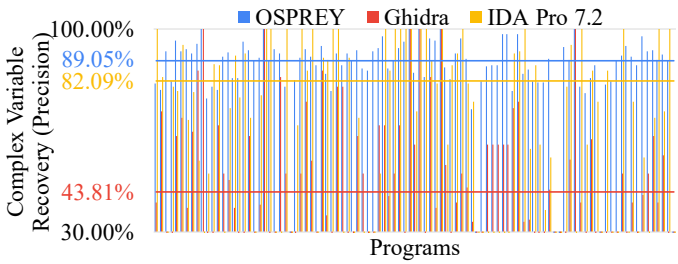


Fig. 12: Precision for complex variables

11 and 12 present the recall and precision of complex types recovery. Complex types include structures, unions, arrays and pointers to structures and unions. Note that Angr could not recover complex data types, hence we do not list its results on the figures. Observe that the recall of OSPREY is around 74%, more than 2 times higher than Ghidra and IDA Pro. The precision of OSPREY also outperforms Ghidra and IDA Pro. One may mention that IDA Pro has a comparable precision rate with OSPREY. The reason is that IDA Pro performs a very conservative type analysis to ensure high precision, leading to a low recall. In Appendix D, we provide insights about why in some cases the deterministic approaches perform better.

To better quantify our results on complex variables, we construct a syntax tree for each complex type (with fields being the child nodes). Nesting structures and unions are precisely modeled, and any inner nesting structure or union type without outer references are ignored. Cycles are removed using a leaf node with a special type tag. We then compare the edit distance of the recovered trees and the ground-truth trees. We compute *tree difference* that is defined as the ratio of the tree edit distance (i.e., the minimum number of tree edits that transform a tree into another) and the whole tree size. The smaller the tree difference, the better the recovery result. Figure 20 in Appendix shows the results. Overall, OSPREY has the minimal tree difference, which is 2.50 and 2.18 times smaller than Ghidra and IDA Pro. Details can be found in our supplementary material [45].

B. Evaluation on Howard Benchmark

Table II in Appendix shows the results for the Howard benchmark. Overall, OSPREY substantially outperforms Ghidra, IDA Pro and Angr, especially for complex variables, in all metrics (recall, precision and tree difference) For all variables, the precision improvement over Ghidra, IDA Pro, and Angr is 28.38%, 38.85%, and 65.51%, respectively, and the recall improvement is 22.98%, 34.78%, and 48.49%, respectively. For complex variables, the precision improvement over Ghidra and IDA Pro is 40.73% and 25.18%, respectively, and the recall improvement is 50.64% and 62.22%, respectively. Our tree differences are 5.21 and 2.64 times smaller than Ghidra and IDA Pro. Compared to Coreutil programs, these programs are more complex, providing more hints to OSPREY. Especially in the complex variable recovery for lighttpd, OSPREY has 84% recall and 86% precision, while Ghidra has 5.5% recall and 27% precision, IDA Pro 6.8% and 50%. Manual inspection discloses that lighthttp has a large number of structures on heap, providing ample hints for OSPREY. We also perform side-by-side comparison with Howard. Details can be found in our supplementary material [45].

C. Sensitivity Analysis

We analyze the sensitivity of OSPREY's accuracy on the prior probabilities $p\uparrow$ and $p\downarrow$. Table I shows the average F_1 scores [46] for the programs in the Howard benchmark set, with $p\uparrow$ varying from 0.7 to 0.9 and $p\downarrow$ from 0.1 to 0.3. We elide other metrics as they reveal similar trendings. Note that the F_1 scores vary within a limited range, less than 2%, with different prior probabilities. It supports that OSPREY is robust against the prior probability changes.

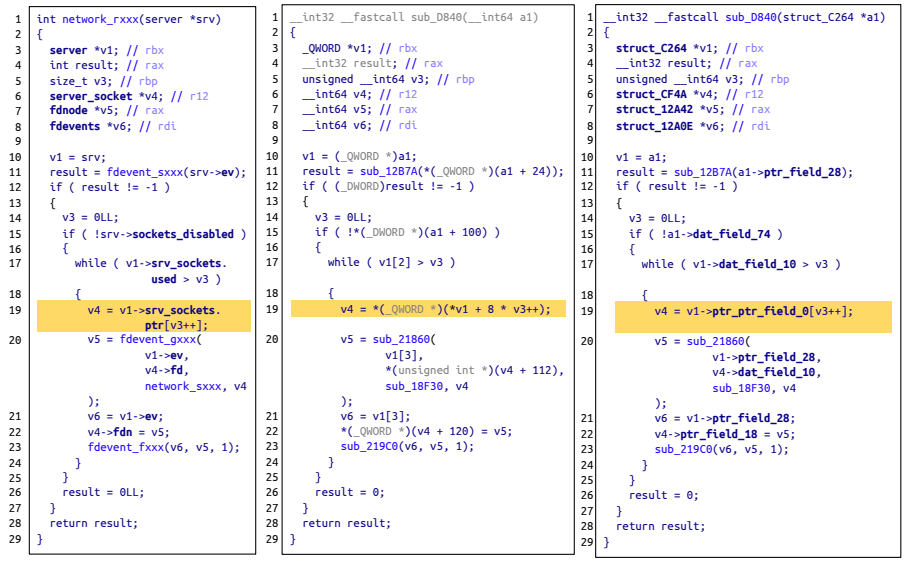
D. Performance Evaluation and Ablation Study

We evaluate the execution time (Appendix G) and scalability (Appendix H). We also study the impact of aggressive optimization (Appendix I) and compilers (Appendix J), as well as the contribution breakdown of different components, by replacing BDA with a dynamic execution based behavior profiler and replacing probabilistic inference with deterministic inference (Appendix K). The results show that OSPREY substantial outperforms other techniques in terms of precision and recall with various compilers and optimization settings, and with complex programs such as Nginx and Apache, although it is the most heavy-weight. The contribution breakdown of BDA and probabilistic inference shows that both are critical.

VII. APPLICATIONS

A. Improving IDA Decompilation

Decompilation transforms low level binary code to human-readable high-level program. The readability of decompiled code hinges on the recovery of variables and data structures. To investigate how OSPREY improves decompilation in IDA, we implement an IDA plugin to feed the decompiler of IDA with the recovered information provided by OSPREY. In Figure 13 and 14, we show a case study on the decompilation of lighttpd's function `network_register_fdevents`. The ground truth, the decompilation results of the vanilla IDA, and of the enhanced IDA are presented in the three columns, respectively. IDA can precisely recover some primitive variables (e.g., `result` at line 4 and `v3` at line 5), but fails to recover the complex data structures (e.g., `v4` at line 6, which is a pointer to a `server_socket` structure). OSPREY can successfully recover the `server_socket` structure. In fact as shown in Figure 13d and 13d, OSPREY can precisely

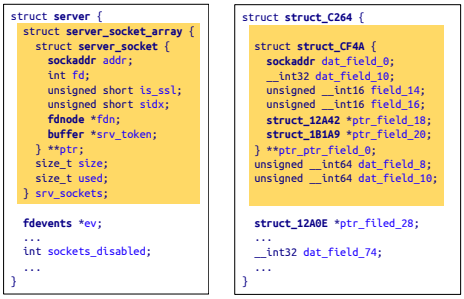


(a) Ground truth (b) Vanilla IDA Pro 7.2 (c) IDA Pro 7.2 w/ OSPREY

Fig. 13: Decompiled results for lighttpd's function *network_register_fdevents*

TABLE I: Average F_1 scores for OSPREY with different prior probabilities

	$p\uparrow=0.7$	$p\uparrow=0.8$	$p\uparrow=0.9$
$p\downarrow=0.1$	0.894	0.907	0.901
$p\downarrow=0.2$	0.909	0.912	0.902
$p\downarrow=0.3$	0.898	0.908	0.903



(d) Ground truth (e) By OSPREY

Fig. 14: Reconstructed Symbols

recover the multiple layers of structure nesting and all the pointer fields. Note that `server_socket_array` is an inner structure type without any outer reference. The recovery of the structure can substantially improve the readability of the decompiled code. See lines 19-20 in Figure 13a. Without the recovered information, we can only learn there are a memory access with complex addressing. With the recovered field and array accesses, we have much more semantic information.

B. Harden Stripped Binary

In the second application, we enhance a recent binary address sanitizer (ASAN) [47] tool RetroWrite [48]) that cannot detect out-of-bound accesses within stack frames or data structures (e.g., overflow of an array field inside a structure). The extended tool can take our recovered structure information to provide protection within data structures. It successfully detects CVE-2019-12802 [49] which cannot be detected by the vanilla RetroWrite. Details can be found in Appendix F.

VIII. Related Work

Binary Analysis. Binary analysis could be static [50], [51], [52], dynamic [1], [53], [54] or hybrid [55], [56]. It has a wide range of applications, such as IoT firmware security [57], [58], [59], [60], [61], [62], memory forensics [63], [64], malware analysis [65], and auto-exploit [66], [67]. A large body of works focus on function entry identification [68], which is the fundamental but challenging tasks of binary analysis. Most related to OSPREY are the studies that focus on binary variable recovery and type inference [12], [1], [53], [11]. Specifically, TIE [12] and REWARD [1] perform static and dynamic analysis to recover type information, respectively. Howard [53] improves REWARDS using heuristics to resolve conflicts. Angr [11] leverages symbolic execution to recover variables. Our work is also related to decompilation [9]. Since it focuses on control-flow recovery, OSPREY is complementary.

Probabilistic Program Analysis. In recent years, probabilistic techniques have been increasingly used in program analysis applications, including symbolic execution [69], [70], model checking [71], [72], [73], binary disassembling [74], and Python type inference [75]. To the best of our knowledge, OSPREY is the first approach that enforces probabilistic variable recovery on stripped binaries.

IX. Conclusion

We develop a novel probabilistic variable and data structure recovery technique for stripped binaries. It features using random variables to denote the likelihood of recovery results such that a large number of various kinds of hints can be organically integrated with the inherent uncertainty considered. A customized and optimized probabilistic constraint solving technique is developed to resolve these constraints. Our experiments show that our technique substantially outperforms the state-of-the-art and improves two downstream analysis.

Acknowledgment

The authors would like to thank the anonymous reviewers and Anders Fogh (the PC contact) for their constructive comments.

The Purdue authors were supported in part by NSF 1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947, and IARPA TrojAI W911NF-19-S-0012. The RUC author was supported in part by National Natural Science Foundation of China (NSFC) under grants 62002361 and U1836209, and the Fundamental Research Funds for the Central Universities and the Research Funds of Renmin University of China under grant 20XNLG03. The UVA author was supported in part by NSF 1850392 and 1916499. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, 2010, pp. 1–1.

[2] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 161–176.

[3] J.-P. Martin, M. Hicks, M. Costa, P. Akritidis, and M. Castro, "Dynamically checking ownership policies in concurrent c/c++ programs," *ACM Sigplan Notices*, vol. 45, no. 1, pp. 457–470, 2010.

[4] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 555–565.

[5] K. Fawaz, H. Feng, and K. G. Shin, "Anatomization and protection of mobile apps' location privacy threats," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 753–768.

[6] E. Bauman, Z. Lin, K. W. Hamlen *et al.*, "Superset disassembly: Statically rewriting x86 binaries without heuristics." in *NDSS*, 2018.

[7] "About ida," https://www.hex-rays.com/products/ida/, 2019.

[8] "Ghidra," https://ghidra-sre.org/, 2019.

[9] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 353–368.

[10] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "{RAZOR}: A framework for post-deployment software debloating," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1733–1750.

[11] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.

[12] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.

[13] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *International conference on compiler construction*. Springer, 2004, pp. 5–23.

[14] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures." in *NDSS*, 2011.

[15] Z. Zhang, W. You, G. Tao, G. Wei, Y. Kwon, and X. Zhang, "Bda: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–31, 2019.

[16] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 32, no. 6, pp. 1–84, 2010.

[17] "Nationalsecurityagency/ghidra," https://github.com/NationalSecurityAgency/ghidra/blob/master/Ghidra/Features/Decompiler/ghidra_scripts/CreateStructure.java#L25, 2019.

[18] E. De Cristofaro, J.-M. Bohli, and D. Westhoff, "Fair: fuzzy-based aggregation providing in-network resilience for real-time wireless sensor networks," in *Proceedings of the second ACM conference on Wireless network security*, 2009, pp. 253–260.

[19] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[20] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[21] "American fuzzy lop (afl)," http://lcamtuf.coredump.cx/afl, 2020.

[22] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, 2019, pp. 803–817.

[23] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, 2008, pp. 209–224.

[24] X. Ge, K. Taneja, T. Xie, and N. Tillmann, "Dyta: dynamic symbolic execution guided with static verification results," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 992–994.

[25] J. Chen, W. Hu, L. Zhang, D. Hao, S. Khurshid, and L. Zhang, "Learning to accelerate symbolic execution via code transformation," in *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, ser. LIPIcs, T. D. Millstein, Ed., vol. 109. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018, pp. 6:1–6:27.

[26] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *Acm Sigplan Notices*, vol. 46, no. 6. ACM, 2011, pp. 504–515.

[27] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, 2011, pp. 265–278.

[28] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: specification inference for explicit information flow problems," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 75–86, 2009.

[29] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1187–1198.

[30] M. A. B. Khadra, D. Stoffel, and W. Kunz, "Speculative disassembly of binary code," in *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. IEEE, 2016, pp. 1–10.

[31] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries," in *USENIX security Symposium*, vol. 13, 2004, pp. 18–18.

[32] M. Popa, "Binary code disassembly for reverse engineering," *Journal of Mobile, Embedded and Distributed Systems*, vol. 4, no. 4, pp. 233–248, 2012.

[33] A. Flores-Montoya and E. Schulte, "Datalog disassembly," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[34] H.-A. Loeliger, J. Dauwels, J. Hu, S. Korl, L. Ping, and F. R. Kschischang, "The factor graph approach to model-based signal processing," *Proceedings of the IEEE*, vol. 95, no. 6, pp. 1295–1322, 2007.

[35] N. E. Beckman and A. V. Nori, "Probabilistic, modular and scalable inference of typestate specifications," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 211–221.

[36] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 607–618.

[37] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 161–176.

[38] J. S. Yedidia, W. T. Freeman, and Y. Weiss, "Generalized belief propagation," in *Advances in neural information processing systems*, 2001, pp. 689–695.

[39] P. F. Felzenszwalb and D. P. Huttenlocher, "Efficient belief propagation for early vision," *International journal of computer vision*, vol. 70, no. 1, pp. 41–54, 2006.

[40] K. Murphy, Y. Weiss, and M. I. Jordan, "Loopy belief propagation for approximate inference: An empirical study," *arXiv preprint arXiv:1301.6725*, 2013.

[41] D. Kahle, T. Savitsky, S. Schnelle, and V. Cevher, "Junction tree algorithm," *Stat*, vol. 631, 2008.

[42] "Coreutils," https://www.gnu.org/software/coreutils/, 2019.

[43] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[44] D. Andriesse, X. Chen, V. Van Der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 583–600.

[45] "Osprey," https://www.cs.purdue.edu/homes/zhan3299/proj/osprey, 2020.

[46] "F1 score," https://en.wikipedia.org/wiki/F1_score, 2020.

[47] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[48] S. Dinesh, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," Ph.D. dissertation, Purdue University Graduate School, 2019.

[49] "Nvd - cve-2019-12802," https://nvd.nist.gov/vuln/detail/CVE-2019-12802, 2019.

[50] J. Lee, T. Avgerinos, and D. Brumley, "TIE: principled reverse engineering of types in binary programs," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[51] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen, "On the static analysis of indirect control transfers in binaries," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, June 24-29, 2000, Las Vegas, Nevada, USA*, 2000.

[52] H. Theiling, "Extracting safe and precise control flow from binaries," in *7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA 2000), 12-14 December 2000, Cheju Island, South Korea*, 2000, pp. 23–30.

[53] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[54] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda, "Inspector gadget: Automated extraction of proprietary gadgets from malware binaries," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 2010, pp. 29–44.

[55] M. H. Nguyen, T. B. Nguyen, T. T. Quan, and M. Ogawa, "A hybrid approach for control flow graph construction from binary code," in *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 2*, 2013, pp. 159–164.

[56] W. He, M. Golla, R. Padhi, J. Ofek, M. Dürmuth, E. Fernandes, and B. Ur, "Rethinking access control and authentication for the home internet of things (iot)," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 255–272.

[57] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel *et al.*, "Toward the analysis of embedded firmware through automated re-hosting," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*, 2019, pp. 135–150.

[58] G. Hernandez, F. Fowze, D. Tian, T. Yavuz, and K. R. Butler, "Firmusb: Vetting usb device firmware using domain informed symbolic execution," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2245–2262.

[59] K. Jansen, M. Schäfer, D. Moser, V. Lenders, C. Pöpper, and J. Schmitt, "Crowd-gps-sec: Leveraging crowdsourcing to detect and localize gps spoofing attacks," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 1018–1031.

[60] D. Freed, S. Havron, E. Tseng, A. Gallardo, R. Chatterjee, T. Ristenpart, and N. Dell, "" is my phone hacked?" analyzing clinical computer security interventions with survivors of intimate partner violence," *Proceedings of the ACM on Human-Computer Interaction*, vol. 3, no. CSCW, pp. 1–24, 2019.

[61] H. Cho, P. Zhang, D. Kim, J. Park, C.-H. Lee, Z. Zhao, A. Doupé, and G.-J. Ahn, "Prime+Count: Novel Cross-world Covert Channels on ARM TrustZone," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Dec. 2018.

[62] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[63] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, and D. Bruschi, "When hardware meets software: A bulletproof solution to forensic memory acquisition," in *Proceedings of the 28th annual computer security applications conference*, 2012, pp. 79–88.

[64] M. Schwarz and A. Fogh, "Drama: How your dram becomes a security problem," 2016.

[65] L. Demetrio, B. Biggio, G. Lagorio, F. Roli, and A. Armando, "Explaining vulnerabilities of deep learning to adversarial malware binaries," *arXiv preprint arXiv:1901.03583*, 2019.

[66] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, "Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 587–600.

[67] H. Lee, C. Song, and B. B. Kang, "Lord of the x86 rings: A portable user mode privilege separation architecture on x86," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1441–1454.

[68] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, 2016, pp. 583–600.

[69] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, 2012, pp. 166–176.

[70] M. Borges, A. Filieri, M. d'Amorim, and C. S. Pasareanu, "Iterative distribution-aware sampling for probabilistic symbolic execution," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, 2015, pp. 866–877.

[71] M. Z. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011, pp. 585–591.

[72] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 341–350.

[73] A. F. Donaldson, A. Miller, and D. Parker, "Language-level symmetry reduction for probabilistic model checking," in *QEST 2009, Sixth International Conference on the Quantitative Evaluation of Systems, Budapest, Hungary, 13-16 September 2009*, 2009, pp. 289–298.

[74] K. A. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin, "Probabilistic disassembly," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, 2019, pp. 1187–1198.

[75] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 607–618.

[76] T. Xie, N. Tillmann, J. De Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE, 2009, pp. 359–368.

[77] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, 2014, pp. 829–844.

[78] W. You, Z. Zhang, Y. Kwon, Y. Aafer, F. Peng, Y. Shi, C. Harmon, and X. Zhang, "Pmp: Cost-effective forced execution with probabilistic memory pre-planning," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 381–398.

[79] A. V. Aho and J. D. Ullman, *Principles of compiler design*. Addison-Wesley, 1977.

[80] P. Zhao and J. N. Amaral, "Function outlining and partial inlining," in *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05)*, 2005, pp. 101–108.

[81] "fortune (unix) - wikipedia," https://en.wikipedia.org/wiki/Fortune_(Unix), 2020.

[82] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[83] "Pin - a dynamic binary instrumentation tool," https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html, 2012.

APPENDIX

A. Limitations of Existing Techniques

Howard [14] is also dynamic analysis based. It improves REWARDS using heuristics to resolve conflicting results. For example, it favors data structures with fields over monolithic scalar variables. Thus, the 128-bit floating-point value copy at instruction `[10]` is ignored by Howard in light of the field accesses at instructions `[14]` and `[15]`, leading to the correctly recovered type for the heap structure. However, Howard employs a number of heuristics to tolerate the various code patterns induced by compiler optimizations. For example, it does not consider `rsp+0x8` as a valid base address. As such, *Howard* mis-classifies offsets `rsp+0x8` and `rsp+0x10` as two separate variables `local_8` and `local_10` as in Figure 2. This illustrates the difficulty of devising generally applicable deterministic heuristics due to the complex behaviors of modern compilers. A heuristic rule being general in one case may become too strict in another case.

Angr [11] is a state-of-the-art open-sourced binary analysis infrastructure, which is widely used in academia and industry. Its variable recovery does not rely on either static or dynamic analysis. Instead, it leverages its built-in concolic execution engine which combines symbolic execution [76] and forced execution [77], [78] to recover variables and their data-flow. Despite the more precise basic information (e.g., data-flow), Angr's variable recovery and type inference are not as aggressive as a few other techniques, especially in the presence of conflicting results. Hence, in Figure 2, the current implementation of Angr cannot recognize the structure on the heap or on the stack. In our experiment (Section VI), Angr achieves 33.04% precision and 59.27% recall.

B. Example for Primitive Analysis Facts Collected by BDA

Consider the motivation example in Figure 1b and assume the function `huft_build` was sampled 10 times. Thus, instruction `[01]` was executed 10 times. As `rsp` stores the base address of region $\mathcal{S}_{\texttt{huft\_build}}$, we have $\mathit{Access}(\texttt{01}, \langle\langle \mathcal{S}_{\texttt{huft\_build}}, 8\rangle, 8\rangle, 10)$ for the first instruction. At instruction `[08]`, `malloc` is called to request 16 bytes of memory, represented by $\mathit{MallocedSize}(\texttt{08}, 16)$. After that, `malloc` returns the base address of heap region $\mathcal{H}_{\texttt{08}}$ and stores it to `rax`. Instruction `[09]` further stores this address to `[rsp]`. Hence we get $\mathit{PointsTo}(\langle\langle \mathcal{S}_{\texttt{huft\_build}}, 0\rangle, 8\rangle, \langle \mathcal{H}_{\texttt{08}}, 0\rangle)$. Instructions `[10]` and `[11]` copy value from `rsp+0x8` to `rax`, generating $\mathit{MemCopy}(\langle\langle \mathcal{S}_{\texttt{huft\_build}}, 8\rangle, 16\rangle, \langle\langle \mathcal{H}_{\texttt{08}}, 0\rangle, 16\rangle)$. Instruction `[15]` accesses `rcx+0x8` where `rcx` is the base register holding the value of $\langle \mathcal{H}_{\texttt{08}}, 0\rangle$, we have $\mathit{BaseAddr}(\texttt{15}, \langle\langle \mathcal{H}_{\texttt{08}}, 8\rangle, 8\rangle, \langle \mathcal{H}_{\texttt{08}}, 0\rangle)$.

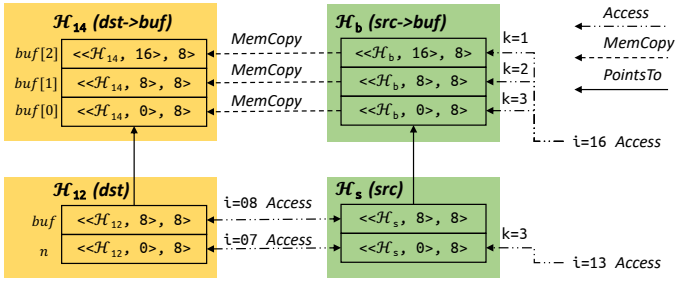
C. Example for Deterministic Inference

In Figure 15, we use a customized string copy function to demonstrate the deterministic reasoning procedure, with the source code in Figure 15a. Lines 1-4 define a `struct`

```
01. typedef struct {
02.   int n;
03.   char *buf;
04. } str_t;
05.
06. void my_print(str_t *s) {
07.   size_t n = s->n;
08.   char *buf = s->buf;
09.   write(1, buf, n);
10. }
```

```
11. str_t *my_strcpy(str_t *src) {
12.   str_t *dst = malloc(sizeof(str_t));
13.   int n = dst->n = src->n;
14.   dst->buf = malloc(sizeof(char) * n);
15.   for (int i = 0; i < n, i++)
16.     dst->buf[i] = src->buf[i];
17.   my_print(src);
18.   my_print(dst);
19.   return dst;
20. }
```

(a) Source code



(b) Memory regions (boxes), chunks (entries in box), and relations (arrows)

Fig. 15: Example for deterministic reasoning

`str_t` that consists of an `int` field `n` and a `char *` field `buf`, indicating the string's length and memory location, respectively. Lines 6-10 define a `my_print()` function that prints a `str_t` structure to `stdout`. Function `my_strcpy()` copies `src` to a heap-allocated `dst` (lines 12-16), and then prints the two strings (lines 17-18). Note that we use source code to illustrate for easy understanding, while OSPREY works on stripped binaries.

Assume BDA samples `my_strcpy()` 3 times, and `src->n` equals to 1, 2, and 3, in the respective sample runs. Assume *sizeof*(`char`)=8. Figure 15b illustrates the regions (denoted by the colored boxes), the memory chunks in regions from all three runs (denoted by the entries inside the colored boxes), and the derived relations (denoted by the arrows). For example, the arrow at the lower-right corner indicates a relation $\mathit{Access}(13, \langle\langle \mathcal{H}_s, 0\rangle, 8\rangle, 3)$. Observe that all the accessed fields of `src` locate in region $\mathcal{H}_s$, the lower green box, and all the accessed elements in `src->buf` locate in region $\mathcal{H}_b$, the upper green box. At line 12, function `malloc`'s parameter is always 16, leading to relation *ConstantAllocSize(12, 16)*. Expression `src->n` at line 13 only accesses a memory chunk $\langle \mathcal{H}_s, 0\rangle$, leading to $\mathit{AccessSingleChunk}(13, \mathcal{H}_s)$. In contrast, from the accessed addresses `src->buf[i]` at line 16, we have $\mathit{AccessMultiChunks}(16, \mathcal{H}_b)$, $\mathit{HiAddrAccessed}(16, \mathcal{H}_b, \langle \mathcal{H}_b, 16\rangle)$, and $\mathit{LoAddrAccessed}(16, \mathcal{H}_b, \langle \mathcal{H}_b, 0\rangle)$, and $\mathit{MostFreqAddAccessed}(16, \mathcal{H}_b, \langle \mathcal{H}_b, 0\rangle, 3)$ denoting the most frequent accessed address is $\langle \mathcal{H}_b, 0\rangle$, i.e., `src->buf[0]` (accessed three times in the three sample runs).

Consider the `my_print()` function, where line 7 accesses both $\langle \mathcal{H}_s, 0\rangle$ and $\langle \mathcal{H}_{12}, 0\rangle$, with H_{12} the heap region allocated at 12, and line 8 accesses both $\langle \mathcal{H}_s, 8\rangle$ and $\langle \mathcal{H}_{12}, 8\rangle$ that have the same offset, indicating $\mathit{UnifiedAccessPntHint}(\langle \mathcal{H}_s, 0\rangle, \langle \mathcal{H}_{12}, 0\rangle, 8)$. Intuitively, the corresponding

```c
unsigned long sha256(char *msg) {
  struct SHA256 ctx; char *c = msg;
  ...
  while (c) {
    ctx.S0 =
      calculate0(ctx.S0, ctx.S1, c);
    ctx.S1 =
      calculate0(ctx.S0, ctx.S1, c);
    c = get_next_chunk(c);
  }
  return fini(ctx.S0, ctx.S1);
}
```

(a) Missing data structures

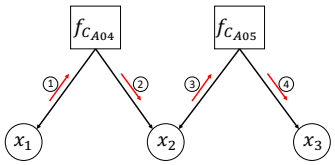
```c
struct my_chunk {
  char buf[0x80];
  struct my_chunk *next;
}

struct my_chunk *xmalloc() {
  struct my_chunk *cur = HEAD;
  while (cur->next & 1)
    cur = (cur->next ^ 1);
  cur->next ^= 1;
  return p;
}
```

(b) Misidentified data structures

Fig. 16: Examples for missing and misidentified data structures

x_1	x_2	$f_{C_{A04}}$
0	0	p
0	1	p
1	0	$1-p$
1	1	p

(a) Probability function for C_{A04}



(b) Factor graph for C_{A04} and C_{A05}

Fig. 17: Factor graph example.

fields of two structures `dst` and `src` are accessed by the same instructions, which implies the presence of structure. Inside function `my_strcpy()`, we acquire a data-flow hint due to the copies from `src` to `dst`. Specifically, we have *DataFlowHint*($\langle \mathcal{H}_b, 0\rangle$, $\langle \mathcal{H}_{14}, 0\rangle$, 16). From the invocation interface between `my_strcpy()` and `my_print()`, we have *PointsToHint*($\langle \mathcal{H}_s, 0\rangle$, $\langle \mathcal{H}_{12}, 0\rangle$, 16) because both the base addresses of `src` and `dst` have been stored to the same function parameter of `my_print()`.

$$① : m_{x_1 \to f_{C_{A04}}}(x_1) = 1$$

$$② : m_{f_{C_{A04}} \to x_2}(x_2 = 0) = \frac{\sum_{x_1} f_{C_{A04}}(x_1, 0) * m_{x_1 \to f_{C_{A04}}}(x_1)}{\sum_{x_1, x_2} f_{C_{A04}}(x_1, x_2) * m_{x_1 \to f_{C_{A04}}}(x_1)}$$

$$= \frac{0.8 + 0.2}{0.8 + 0.8 + 0.2 + 0.8} * 1 = \frac{1}{2.6}$$

$$m_{f_{C_{A04}} \to x_2}(x_2 = 1) = \frac{\sum_{x_1} f_{C_{A04}}(x_1, 1) * m_{x_1 \to f_{C_{A04}}}(x_1)}{\sum_{x_1, x_2} f_{C_{A04}}(x_1, x_2) * m_{x_1 \to f_{C_{A04}}}(x_1)}$$

$$= \frac{0.8 + 0.8}{0.8 + 0.8 + 0.2 + 0.8} * 1 = \frac{1.6}{2.6}$$

$$③ : m_{x_2 \to f_{C_{A05}}}(x_2) = m_{f_{C_{A04}} \to x_2}(x_2)$$

$$④ : m_{f_{C_{A05}} \to x_3}(x_3 = 1) = \frac{\sum_{x_2} f_{C_{A05}}(x_2, 1) * m_{x_2 \to f_{C_{A05}}}(x_2)}{\sum_{x_2, x_3} f_{C_{A05}}(x_2, x_3) * m_{x_2 \to f_{C_{A05}}}(x_2)}$$

$$= 0.65$$

D. Case Studies

Cases Where Ghidra and IDA Pro Do Better. There are few cases where Ghidra and IDA Pro achieve better performance. Further inspection reveals that those are very simple programs without complex structures (e.g., `struct` or in-stack array), where no conflict will occur during deterministic reasoning. Hence, approaches like Ghidra and IDA Pro can handle them well. *OSPREY* also works well, but may misidentify very few variables due to the infeasible paths produced by BDA.

Missing Data Structures. We find that missing data structures are mainly due to stack-nested `structs` that are never used outside their stack frames. Consider the code snippet from *sha256sum* in Figure 16a, where a stack-nested structure `SHA256 ctx` is allocated on stack and used exclusively within the function. As such, OSPREY cannot gather any valuable hints about `ctx`. That is also the major reason that OSPREY has relatively large tree difference for those hashing binaries (e.g., *sha256sum*) in Figure 20 in Appendix.

Misidentified Data Structures. In our benchmarks, custom heap allocators are a major source of misidentified data structures by OSPREY. Consider a simplified `xmalloc` from *grep* in Figure 16b. Its basic allocation unit is called `my_chunk`, consisting of a buffer `buf` and a pointer `next`. Different from common pointers, `my_chuck.next` uses its last bit to indicate whether this chunk is in use (in normal case, the last bit is always zero due to memory alignment). Thus, at line 5, `xmalloc` finds the first chunk whose in-use bit is not set, sets the bit, and returns the chunk. As a result, `my_struct.next` can point to a `struct my_struct` or `char my_struct.buf[1]` (both are common cases). These confusing *PointsTo* hints misled OSPREY to falsely recover unions. Other reasons include insufficient hints.

E. Example for Transforming A Probabilistic Constraint to A Factor Graph

Let boolean variables x_1, x_2, and x_3 denote *PrimitiveVar*(v), *PrimitiveAccess*(i, v), and *PrimitiveVar*(v'), respectively. Rules C_{A04} is transformed to $x_1 \xrightarrow{p\uparrow} x_2$, which denotes the probability function in Figure 17a. The probability function for C_{A05} is similar. The two form a factor graph in Figure 17b, which could be solved by belief propagation algorithms with passing messages on it. For example, assume the prior probabilities of C_{A04} and C_{A05} are both 0.8, and we want to compute the marginal probability $p(x_3 = 1)$, that is, the probability of v being of primitive type. As the factor graph is a tree, we can call x_3 the root node. Then message passing starts from the leaf node x_1. After messages reach the root finally, the marginal probability of x_3 can be computed. The definition and computation of each message is shown as follows.

F. Harden Stripped Binary

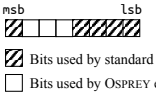
Exposing potential memory bugs is very important for vulnerability detection. Address sanitizer (ASAN) [47], a tripwire-based memory checker, can be used to increase the likelihood of triggering a crash when a memory corruption occurs. The principle of ASAN is to insert redzones at the border of variables. Program crashes whenever an out-of-bound access touches the redzone. The effectiveness of ASAN is determined by the accuracy of identifying the variable borders, which is very challenge if source code or debugging information is not available. The state-of-the-art binary-level ASAN solution (*RetroWrite* [48]) conducts

TABLE II: Analysis results of Howard benchmark

Metric	Program	Osprey		Ghidra		IDA Pro		Angr	
		Reca.	Prec.	Reca.	Prec.	Reca.	Prec.	Reca.	Prec.
Overall Variable	wget	85.32	86.14	66.83	62.94	62.82	60.02	39.94	26.96
	lighttpd	87.67	86.35	52.65	52.15	46.18	41.37	44.35	22.90
	grep	82.10	84.07	67.63	69.34	67.09	63.97	46.64	30.53
	gzip	100.0	100.0	84.78	79.10	78.26	75.00	59.78	37.42
	fortune	100.0	100.0	68.29	51.16	26.83	22.00	21.95	11.25
	Avg.	**91.02**	**91.32**	**68.04**	**62.94**	**56.24**	**52.47**	**42.53**	**25.81**
Complex Variable	wget	73.26	83.14	29.21	47.20	20.29	76.39	N/A	N/A
	lighttpd	84.32	85.87	05.51	27.08	06.78	50.00	N/A	N/A
	grep	57.39	84.52	10.43	35.29	11.30	41.67	N/A	N/A
	gzip	100.0	100.0	66.67	73.68	57.14	81.25	N/A	N/A
	fortune	100.0	100.0	50.00	66.67	00.83	33.33	N/A	N/A
	Avg.	**82.99**	**90.71**	**32.35**	**49.98**	**20.77**	**65.53**	**N/A**	**N/A**
Tree Difference	wget	28.92		70.99		62.84		N/A	
	lighttpd	12.37		80.18		64.87		N/A	
	grep	30.09		78.41		60.93		N/A	
	gzip	00.00		42.50		00.00		N/A	
	fortune	00.00		100.0		00.00		N/A	
	Avg.	**14.28**		**74.42**		**37.73**		**N/A**	

```
01. typedef struct node {
02.   long data[0x20];
03.   struct node *next;
04. } node_t;
05.
06. void gee() {
07.   node_t *p =
          malloc(sizeof(node_t));
08.   for (int i=0; i<=0x20, i++)
09.     p->data[i] = 0;
10. }
```

(a) Simplified example of `CVE-2019-12802`, which has an out-of-boundary memory access for array `data` inside structure `node_t`.



(b) Address sanitizer maps 8 bytes of the application memory into 1 byte of the shadow memory named `shadow byte`. However, only 5 bits of each shadow byte are used in standard libasan.so.4.

```
[A] xor rcx, rcx
[B] cmp ecx, 0x20
[C] jg <ret_gee>
[C1] lea rdi, [rax+rcx*8]
[C2] shr rdi, 3
[C3] mov dil, [rdi+SHADOW_BASE]
[C3a] mov rsi, rdi
[C3b] and sil, 0x30
[C3c] cmp sil, 0x30
[C3d] jz <asan_report_error>
[C3e] and dil, 0x8F
[C4] test dil, dil
[C5] jnz <asan_report_error>
[D] mov [rax+rcx*8], 0
[E] inc ecx
[F] jmp B
```

(c) Assemble code for line 8 and line 9 in Fig 18a. Lines A, B, C, D, E, F are the original assemble code, lines C1, C2, C3, C4, C5 are instrumented by `RetroWrite`, and lines C3a, C3b, C3c, C3d, C3e are instrumented by our enhancement. Only our instrumentation can report `CVE-2019-12802`.

Fig. 18: Field-level binary ASAN instrumentation for CVE-2019-12802.

very coarse-grained border identification. Specifically, for an allocated heap region, redzones are only inserted before and after the region, not between the variables/fields within the region. This may degrade the effectiveness of ASAN. Take CVE-2019-12802 [49] as an example. It is an out-of-bound vulnerability whose simplified code is shown in Figure 18a. The vulnerability occurs at line 9, in which there is an out-of-bound memory access for array *data* inside the *node_t* structure. *RetroWrite* does not insert redzone code within the *node_t* structure, hence cannot detect the vulnerability.

We strengthen *RetroWrite* to take in our reconstructed symbol information such that corruptions internal to a structure can be detected. Specifically, we aim to prevent scalar variables from being accessed by any array instruction. To avoid false warnings and offer a strong (probabilistic) guarantee, we carefully define *scalar*

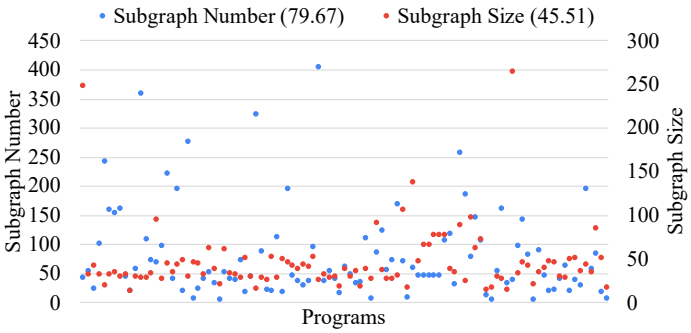


Fig. 19: Subgraph statistics in factor graph inference

variables and *array instruction*. We define v as a scalar variable, if $P(Scalar(v)) > 0.99 \wedge \neg(\exists(a_1, a_2),\ s.t.\ (a_1 \leq v.a \leq a_2) \wedge (P(Array(a_1, a_2)) > 0.01))$ Similarly, we define i as an array instruction, if $\forall v : Accessed(i, v), \exists(a_1, a_2),\ s.t.\ AccessMultiChunks(i, v.a.r) \wedge (a_1 \leq v.a \leq a_2) \wedge (P(Array(a_1, a_2))) > 0.99)$. We leverage *RetroWrite* to instrument the target binary. For any memory access by an array instruction, besides the basic ASAN checks provided by *RetroWrite*, we additionally check it is accessing a scalar variable.

Figures 18c and 18b present the details of our implementation. Lines `[A][B][C][D][E][F]` in Figure 18c are the original assembly code for line 8-9 in Figure 18a, where `rcx` in line `[B]` stores the value of `i` and `rax+rcx*8` in line `[D]` stores the address of `p->data[i]`. Lines `[C1][C2][C3][C4][C5]` are instrumented by *RetroWrite*. They first get the target address of instruction `[D]` (line `[C1]`), read its shadow value (`dil`) from the corresponding shadow memory (lines `[C2][C3]`), and validate the shadow value (lines `[C4][C5]`). *RetroWrite*'s ASAN is based on the standard libasan.so.4. Hence it directly invokes `asan_report_error` to report errors. An interesting observation is that, even though libasan.so.4 uses one byte to store shadow value, only 5 bits of the byte are used, as shown by the shadow value layout in Figure 18b. This allows us to store more meta information using the remaining 3 bits. In our case, we use one bit to record whether the memory stores a scalar variable. After that, we instrument more validation instructions for array instructions. Lines `[C3a][C3b][C3c][C3d][C3e]` are added by OSPREY, for array instruction `[D]`. The instrumentation validates whether the accessed memory stored a scalar variable. As such, the mentioned CVE can be successfully detected. The instrumented code does not cause any false warnings when executed on normal test cases. Note that although probabilistic guarantees may not be strong enough for production systems, they make perfect sense for vulnerability detection, in which rare false warnings are acceptable.

G. Execution Time

In Table III, we measure the execution time of different tools on the two benchmark sets. Due to the space limit, we aggregate CoreUtils' results and show the averaged data. Detailed results [45] are available for interested readers. In

TABLE III: Execution time of different tools. The numbers in the brackets denote how many times OSPREY is slower than the corresponding tool.

	Program	Osprey	Ghidra	IDA	Angr
Howard	wget	3604.80s	94.74s (37.05×)	18.98s (188.88×)	41.47s (85.92×)
	lighttpd	2013.12s	63.89s (30.51×)	16.80s (118.83×)	31.60s (62.70×)
	grep	832.52s	66.75s (11.47×)	32.62s (24.52×)	33.88s (23.57×)
	gzip	483.65s	52.84s (8.15×)	11.84s (39.84×)	18.57s (25.04×)
	fortune	422.92s	37.48s (10.28×)	6.30s (66.13×)	7.11s (58.45×)
CoreUtils		528.24s	35.35s (13.94×)	5.80s (90.08×)	10.55s (49.07×)
Avg.		**1314.21s**	**58.51s (18.57×)**	**15.39s (88.04×)**	**23.87s (50.79×)**

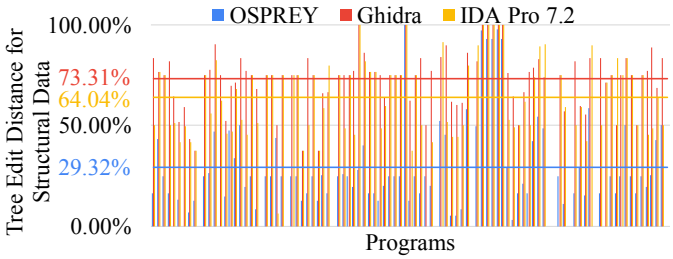


Fig. 20: Tree difference for CoreUtils

general, OSPREY is 18.57, 88.04, and 50.79 times slower than Ghidra, IDA Pro, and Angr, respectively. We argue that reverse engineering is often a one-time effort and OSPREY provides a different trade-off between cost and accuracy. It is also worth noting that Ghidra is the second slowest one due to its register-based data-flow analysis, and IDA Pro is the fastest one as its variable recovery mainly relies on hard-coded code pattern matching rules.

H. Scalability

To assess the scalability of OSPREY, we evaluate OSPREY on Apache and Nginx, two well-known applications with significantly larger code base than the benchmarks we used. On both programs, OSPREY produces the highest F_1 Score for overall and complex variable recovery, and the lowest tree difference. Details can be found in our supplementary material [45].

I. Impact of Aggressive Optimization

To understand the impact of aggressive optimizations, we evaluate OSPREY on the two benchmark sets compiled with -O3, the most aggressive builtin optimization flag of GCC.

The results are shown in Table IV. We calculate the F_1 score [46] for each tool, and summarize CoreUtils' results. Table IVa presents the overall F_1 scores including both scalar and complex variables. The average F_1 scores (with -O3) for OSPREY, Ghidra, IDA Pro, and Angr are 0.70, 0.48, 0.27, and 0.16, respectively; and the degradation from the default optimization (-O0) are 22.48%, 27.18%, 45.47%, and 42.64%, respectively. Although recovering accurate types from aggressively optimized code is very challenging, OSPREY substantially outperforms other state-of-the-art techniques. Besides, OSPREY is the most robust tool among all the evaluated ones. Manual inspection discloses that some aggressive optimizations disrupt OSPREY's hints (e.g., loop unrolling [79] and partial function inlining [80]), resulting in the degraded accuracy. For example, loop unrolling can generate multiple copies of a single memory access instruction such that we lose the hint that detects an array by observing consecutive memory locations being accessed by the same instruction.

TABLE IV: Impact of aggressive optimizations with -O3. **Def.**, **O3**, **Degra.**, and **# CVars** denote the analysis results for binaries compiled under the default optimization (-O0), under -O3, degradation from -O0, and the number of complex variables in memory, respectively.

(a) F_1 scores for overall variable recovery

	Program	Osprey			Ghidra			IDA			Angr		
		Def.	**O3**	**Degra.**	**Def.**	**O3**	**Degra.**	**Def.**	**O3**	**Degra.**	**Def.**	**O3**	**Degra.**
Howards	wget	0.86	0.66	23.11%	0.65	0.51	20.89%	0.48	0.20	58.65%	0.32	0.21	34.52%
	lighttpd	0.87	0.65	25.26%	0.52	0.29	44.59%	0.43	0.15	64.07%	0.30	0.09	71.72%
	grep	0.83	0.74	11.21%	0.68	0.60	11.74%	0.54	0.20	63.03%	0.37	0.16	55.72%
	gzip	1.00	0.74	26.44%	0.82	0.37	55.20%	0.67	0.24	64.30%	0.46	0.16	64.46%
	fortune	1.00	0.82	17.86%	0.58	0.63	-7.16%	0.24	0.33	-38.36%	0.15	0.20	-34.44%
CoreUtils		0.89	0.62	31.01%	0.74	0.49	37.78%	0.71	0.27	61.12%	0.43	0.16	63.84%
Avg.		**0.91**	**0.70**	**22.48%**	**0.67**	**0.48**	**27.18%**	**0.51**	**0.23**	**45.47%**	**0.34**	**0.16**	**42.64%**

(b) F_1 scores for complex variable recovery

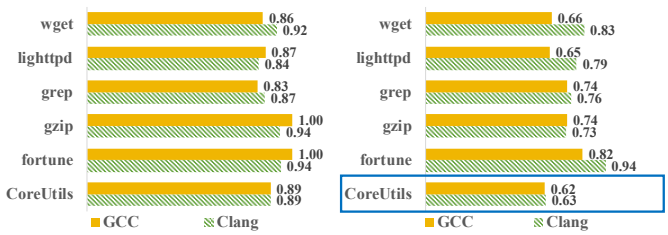
	Program	Osprey		Ghidra		IDA		# CVars	
		Def.	**O3**	**Def.**	**O3**	**Def.**	**O3**	**Def.**	**O3**
Howards	wget	0.78	0.55	0.36	0.45	0.32	0.14	239	127
	lighttpd	0.85	0.44	0.09	0.38	0.12	0.33	318	43
	grep	0.68	0.50	0.16	0.35	0.18	0.20	120	38
	gzip	1.00	0.55	0.70	0.35	0.67	0.33	45	41
	fortune	1.00	0.76	0.57	0.71	0.13	0.52	16	13
CoreUtils		0.80	0.62	0.38	0.43	0.39	0.35	23	11
Avg.		**0.85**	**0.57**	**0.38**	**0.45**	**0.30**	**0.31**	**127**	**45**

(c) Tree difference

	Program	Osprey			Ghidra			IDA		
		Def.	**O3**	**Degra.**	**Def.**	**O3**	**Degra.**	**Def.**	**O3**	**Degra.**
Howards	wget	28.92	57.24	49.47%	70.99	72.88	02.48%	62.84	75.14	16.37%
	lighttpd	12.37	21.42	42.25%	80.18	55.10	-45.53%	64.87	62.81	-03.28%
	grep	30.09	26.96	-11.63%	78.41	72.62	-07.97%	60.93	89.68	32.06%
	gzip	00.00	41.67	100.0%	42.50	62.50	32.00%	00.00	50.00	100.0%
	fortune	00.00	08.00	100.0%	100.0	50.00	-100.0%	00.00	50.00	100.0%
CoreUtils		29.32	63.26	53.65%	73.31	78.69	06.83%	64.04	78.61	18.54%
Avg.		**16.78**	**36.42**	**55.63%**	**74.23**	**65.28**	**-18.70%**	**42.11**	**67.71**	**43.95%**

Table IVb shows the F_1 scores for complex variable recovery. Observe that OSPREY still achieves substantially better F_1 of 0.57 (compared to 0.45 for Ghidra and 0.31 for IDA Pro). One may notice that Ghidra and IDA Pro get better results with the -O3 flag. Although it seems counter-intuitive, further inspection shows that it is not because they are having better performance but rather the number of complex variables in memory becomes smaller. Recall that we consider a structure being pointed to by a pointer in memory a complex variable. With -O3, these pointers are largely allocated to registers. We do not collect results for these cases as Howard does not consider variables in registers. While Ghidra and IDA Pro tend to have trouble with complex variables in memory, the number of such cases are reduced.

We additionally count the number of complex variables, shown in Table IVb. The results show that the number of



(a) Default optimization (b) **O3** optimization

Fig. 21: OSPREY's F_1 scores for overall variable recovery on the two benchmark sets compiled by GCC and Clang. The results of CoreUtils are averaged over all programs.

complex variables decreases a lot from the default setting (127 v/s 45), supporting our hypothesis.

Table IVc presents the tree difference. Although OSPREY has the smallest tree difference of 36.42 (compared to 65.28 for Ghidra and 67.71 for IDA Pro), the aggressive optimizations have larger impact on OSPREY. This is however reasonable because OSPREY's structure recovery mainly depends on hints from program behaviors which can be greatly changed by optimizations, while Ghidra and IDA Pro mainly depend on predefined function prototypes of external library calls which are rarely influenced by optimizations. Ghidra's register-based data-flow analysis also benefits from optimizations. We foresee that a set of rules particularly designed for optimized programs can be developed for OSPREY. We will leave that to our future work.

Finally, we want to point out that *fortune* is an outlier which always achieves better results under aggressive optimizations. This is because *fortune* is a very simple program (randomly outputting predefined sentences [81]) and O3 optimizations put most of its variables in registers, reducing aliasing and greatly benefiting the register-based data-flow analysis.

J. Impact of Different Compilers

To study the robustness over different compilers, we additionally examine OSPREY on benchmarks compiled by Clang [82], another mainstream compiler. We use Clang 6.0 to compile the two benchmark sets with the default and -O3 optimization flags, and summarize the results in Figure 21. The results show that OSPREY has good robustness with different compilers under the default compilation setting (less than 6% difference for each program). Although there is a larger difference between GCC and Clang under the -O3 setting, we speculate that it is because the -O3 optimizations of GCC and Clang behave differently (e.g., they have different thresholds for loop unrolling). The results of complex variable recovery and tree difference reveal similar trends and are hence elided.

K. Contribution Breakdown of Different Components

To better understand the effect of different components, including BDA and probabilistic inference, we further evaluate OSPREY with two variations. Specifically, to study the contributions of BDA, in the first variation, we replace the BDA component with a dynamic-execution component built upon Pintools [83]. Following the same setup as Howard, we use the provided test suite and also KLEE to increase code coverage. To study the effect of probabilistic inference, in the second variation, we turn the probabilistic inference to deterministic inference. The deterministic inference rules are largely derived from the probabilistic rules but have the probabilities removed. As such, when multiple contradictory inference results are encountered (e.g., conflicting types for a variable), which are inevitable due to the inherent uncertainty, the algorithm randomly picks one to proceed.

TABLE V: Effects of BDA and probabilistic inference. **Original**, **w/o BDA**, and **w/o Prob.** stand for the original OSPREY, OSPREY with a dynamic-execution component instead of BDA, and OSPREY with deterministic inference instead of probabilistic inference, respectively. **Cov.** denotes the fraction of functions that the dynamic approach exercised.

Program	Original		w/o BDA			w/o Prob.	
	Reca.	Prec.	Reca.	Prec.	Cov.	Reca.	Prec.
wget	85.32	86.14	29.46	86.31	51%	45.43	47.21
lighttpd	87.67	86.35	73.75	97.16	55%	40.24	40.74
grep	82.10	84.07	44.48	89.78	50%	44.76	46.04
gzip	100.0	100.0	43.48	100.0	74%	64.37	64.37
fortune	100.0	100.0	75.61	100.0	76%	78.57	78.57
Avg.	**91.02**	**91.32**	**53.36**	**94.65**	**61%**	**54.67**	**55.39**

The results are shown in Table V. We report the precision and recall of the first variation for overall variables in the fourth and fifth columns. We also report the dynamic code coverage in the sixth column. Due to page limits, we elide other metrics as they are less interesting. Compared with the original OSPREY, the dynamic-execution-based OSPREY has slightly higher precision but lower recall. As dynamic execution strictly follows feasible paths, there are fewer conflicts, benefiting the precision. However, the conflicts introduced by BDA's incapabilities of determining infeasible paths are decentralized and cumulatively resolved by the large number of hints, making the improvement limited. On the other hand, the dynamic-execution-based OSPREY cannot get hints from the non-executed functions, leading to the low recall. Hence, we argue that BDA is essential to OSPREY.

The results of the second variation are shown in the last two columns of Table V. Note that the deterministic version of OSPREY has nearly 40% decrease in terms of both recall and precision. Such results indicate the probabilistic parts of OSPREY are critical. We also study the reason behind the degradation. On one hand, due to the infeasible paths, BDA may generate many invalid accesses. When these accesses conflict with the valid ones, the deterministic algorithm may choose the wrong one. On the other hand, many inference rules / hints have inherent uncertainty. For example, rule C_{B02} says when an instruction accesses multiple addresses in the same region, likely, there is an array in that region. Note that it is likely but not certain, as the situation could also be that a pointer points to multiple individual objects. Deterministic approaches are by their nature not suitable for handling such inherent uncertainty.