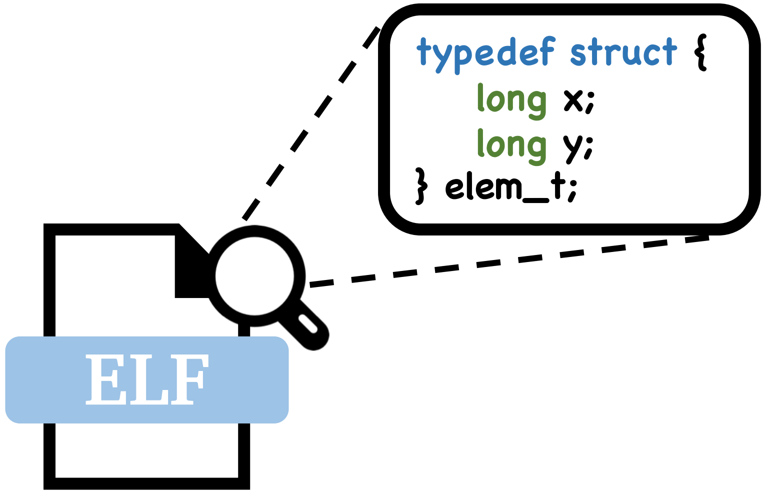
OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary

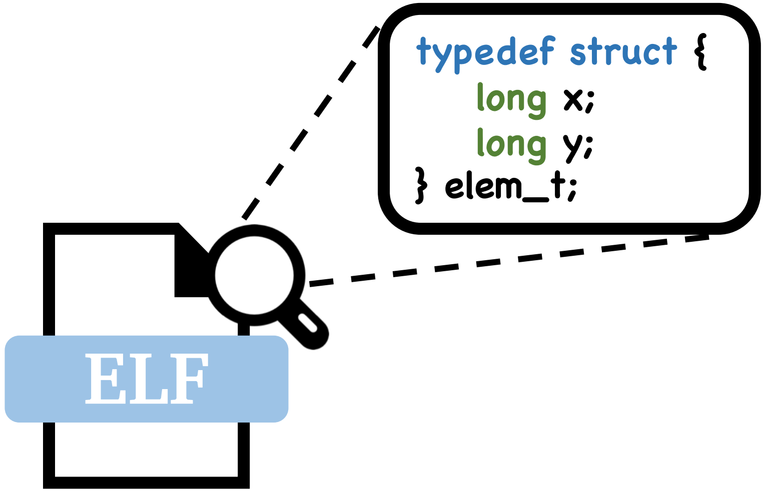
Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, Xiangyu Zhang

PURDUE UNIVERSITY
中国人民大學 RENMIN UNIVERSITY OF CHINA
UNIVERSITY of VIRGINIA
UNIVERSITY OF WATERLOO

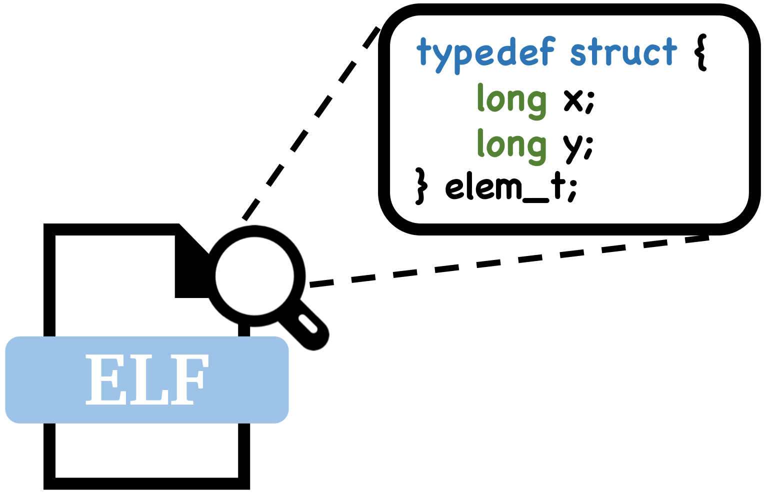


Variable and Data Structure Recovery

Bug Detection



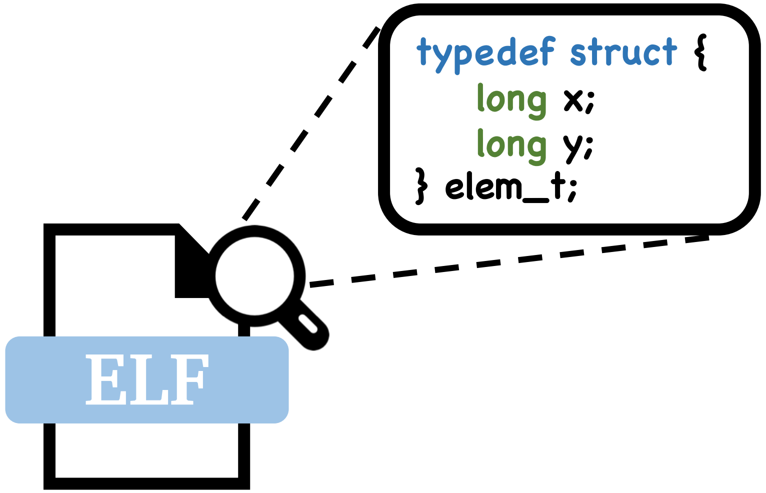
Variable and Data Structure Recovery



Variable and Data Structure Recovery

Bug Detection

Malware Analysis

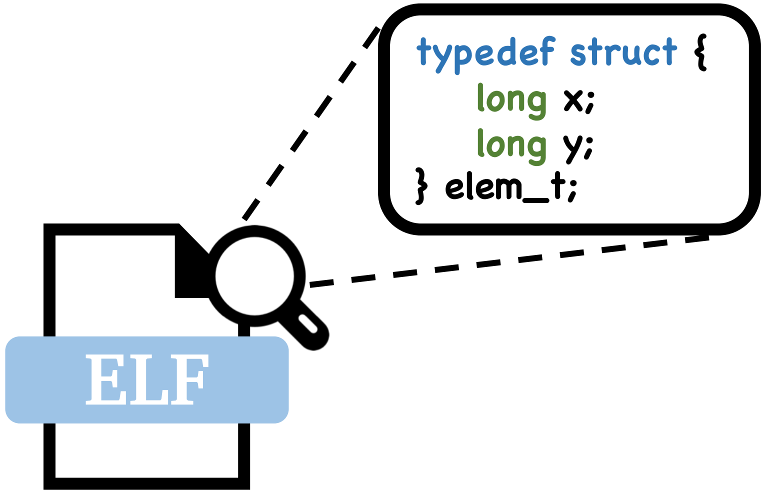


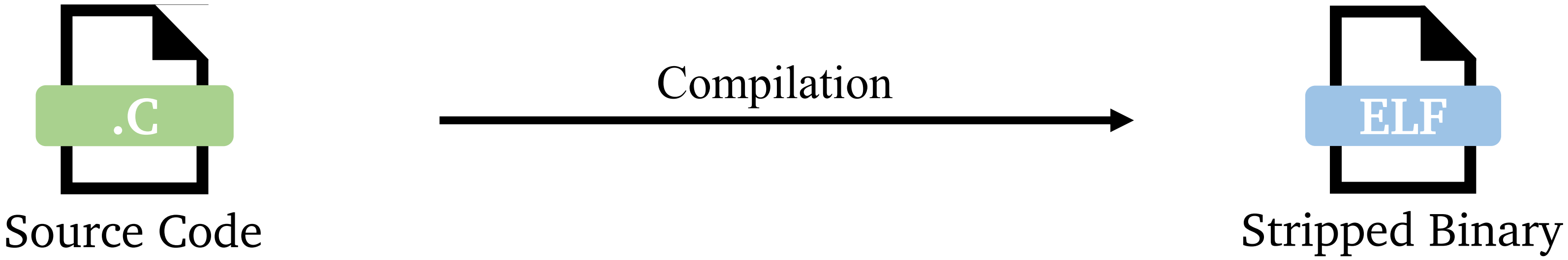
Variable and Data Structure Recovery

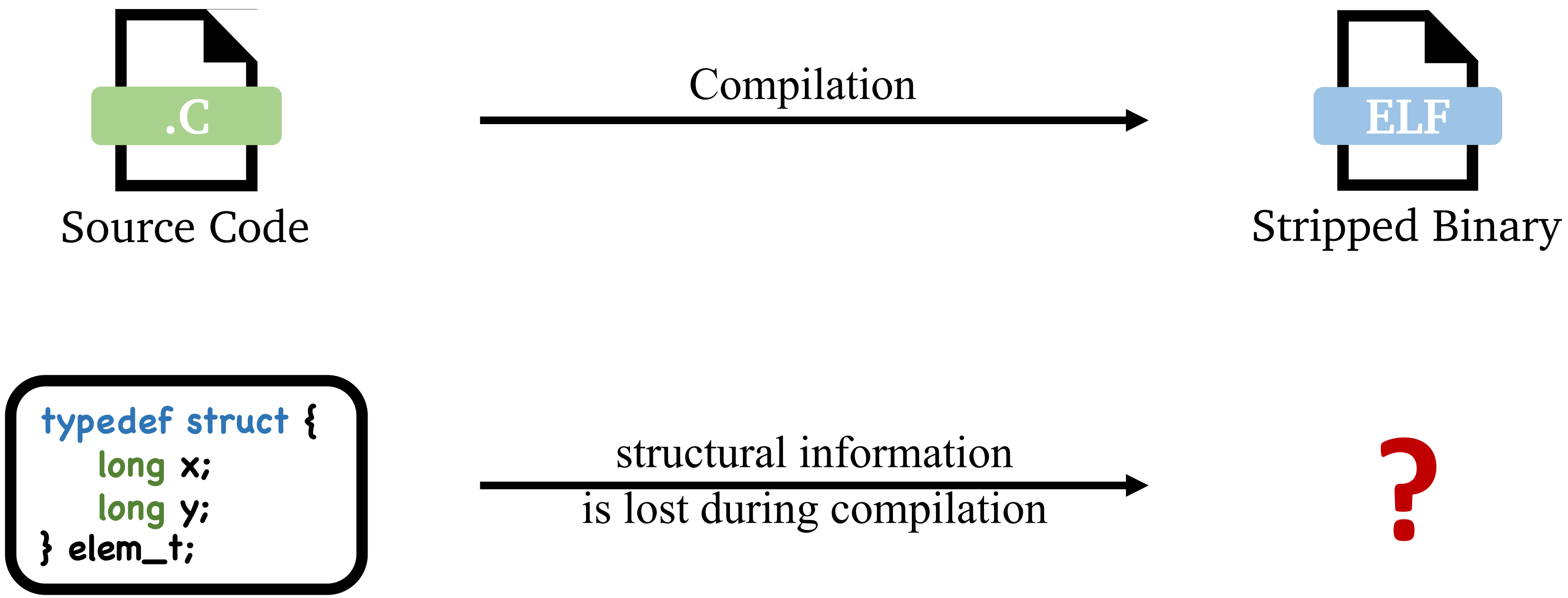
Bug Detection

Malware Analysis

Automatic Exploit Generation


typedef struct {
long x;
long y;
} elem_t;
ELF


.C
Source Code
Compilation
ELF
Stripped Binary


.C
Source Code
Compilation
ELF
Stripped Binary
typedef struct {
long x;
long y;
} elem_t;
structural information
is lost during compilation
?

.C

Source Code

Compilation →

ELF

Stripped Binary

```c
typedef struct {
    long x;
    long y;
} elem_t;
```

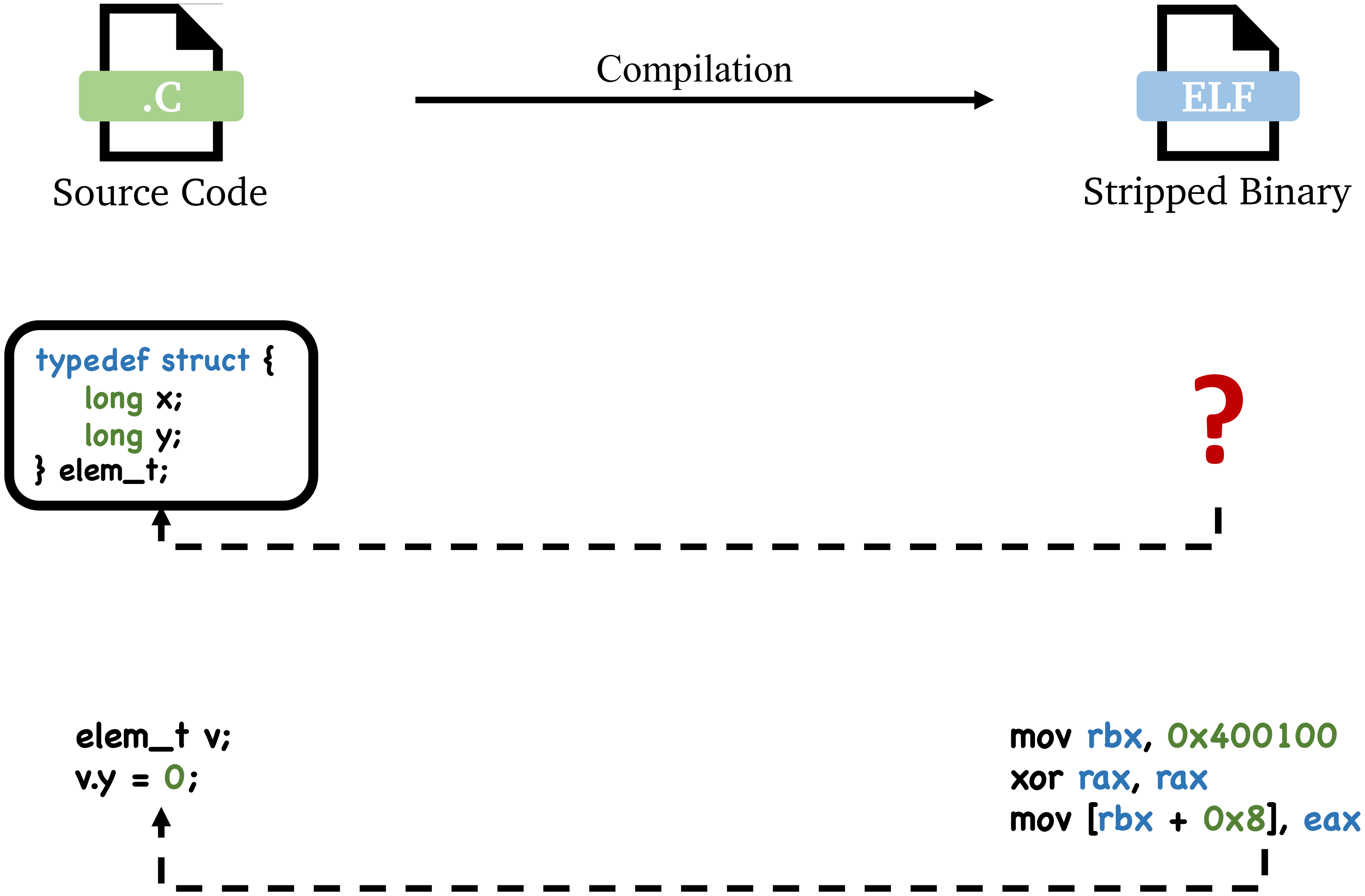
structural information is lost during compilation →

?

```c
elem_t v;
v.y = 0;
```

structural information is lost during compilation →

```
mov rbx, 0x400100
xor rax, rax
mov [rbx + 0x8], eax
```


.C
Source Code
Compilation
ELF
Stripped Binary
typedef struct {
long x;
long y;
} elem_t;
?
elem_t v;
v.y = 0;
mov rbx, 0x400100
xor rax, rax
mov [rbx + 0x8], eax

Motivation Example

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Motivation Example

```c
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Motivation Example

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Motivation Example

```c
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Motivation Example

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Motivation Example

```c
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Motivation Example

Variable and data structure recovery is to recover high-level semantic information from the compiled executables.

- Array
- Structure
- Pointer

In the motivation example, we are trying to recover:

- the structure of `elem_t`
- `p` is a pointer to `elem_t`
- `v` is a an `elem_t` located on the stack

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Motivation Example

Variable and data structure recovery is to recover high-level semantic information from the compiled executables.

- Array
- Structure
- Pointer

In the motivation example, we are trying to recover:

- the structure of **`elem_t`**
- **`p`** is a pointer to **`elem_t`**
- **`v`** is a an **`elem_t`** located on the stack

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Motivation Example

Variable and data structure recovery is to recover high-level semantic information from the compiled executables.

- Array
- Structure
- Pointer

In the motivation example, we are trying to recover:

- the structure of **`elem_t`**
- **`p`** is a pointer to **`elem_t`**
- **`v`** is a an **`elem_t`** located on the stack

```c
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Motivation Example

Variable and data structure recovery is to recover high-level semantic information from the compiled executables.

- Array
- Structure
- Pointer

In the motivation example, we are trying to recover:

- the structure of **`elem_t`**
- **`p`** is a pointer to **`elem_t`**
- **`v`** is a an **`elem_t`** located on the stack

```c
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Ghidra and IDA Pro

```c
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Ghidra and IDA Pro

Ghidra and IDR Pro are based on specific code patterns.

- Unreliable due to uncertainty, leading to contradicting results
- Cannot handle complex structures

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Ghidra and IDA Pro

Ghidra and IDR Pro are based on specific code patterns.

- Unreliable due to uncertainty, leading to contradicting results
- Cannot handle complex structures

```c
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Ghidra and IDA Pro

Ghidra and IDR Pro are based on specific code patterns.

- Unreliable due to uncertainty, leading to contradicting results
- Cannot handle complex structures

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Ghidra and IDA Pro

Ghidra and IDR Pro are based on specific code patterns.

- Unreliable due to uncertainty, leading to contradicting results
- Cannot handle complex structures

For example, the **movdqa** and **movups** instruction pair denotes a 128-bit packed floating-point value movement. Ghidra and IDA Pro recognize this and guess the structure is a float-point variable.

```
call malloc
movdqa xmm0, [rsp + 0x8]
movups [rax], xmm0
```

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Ghidra and IDA Pro

Ghidra and IDR Pro are based on specific code patterns.

- Unreliable due to uncertainty, leading to contradicting results
- Cannot handle complex structures

For example, the **movdqa** and **movups** instruction pair denotes a 128-bit packed floating-point value movement. Ghidra and IDA Pro recognize this and guess the structure is a float-point variable.

```
call malloc
movdqa xmm0, [rsp + 0x8]
movups [rax], xmm0
```

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Ghidra and IDA Pro

Ghidra and IDR Pro are based on specific code patterns.

- Unreliable due to uncertainty, leading to contradicting results
- Cannot handle complex structures

For example, the **movdqa** and **movups** instruction pair denotes a 128-bit packed floating-point value movement. Ghidra and IDA Pro recognize this and guess the structure is a float-point variable.

```
typedef union {
  int64 u_0[2];
  int128 u_1;
} elem_t;
```

IDA Pro

```
typedef struct {
  int32 s_0[4];
} elem_t;
```

Ghidra

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Howard

```c
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Howard

Howard is a state-of-the-art dynamic analysis tool.

- Use dynamic analysis to collect program behaviors
- Leverage heuristics such as field accesses are performed by first loading the base address of the data structure and then offsetting.

However,

- Its effectiveness hinges on the availability of high quality inputs.
- The heuristics do not always hold.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Howard

Howard is a state-of-the-art dynamic analysis tool.

- Use dynamic analysis to collect program behaviors
- Leverage heuristics such as field accesses are performed by first loading the base address of the data structure and then offsetting.

However,

- Its effectiveness hinges on the availability of high quality inputs.
- The heuristics do not always hold.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Howard

Howard is a state-of-the-art dynamic analysis tool.

- Use dynamic analysis to collect program behaviors
- Leverage heuristics such as field accesses are performed by first loading the base address of the data structure and then offsetting.

However,

- Its effectiveness hinges on the availability of high quality inputs.
- The heuristics do not always hold.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Howard

Howard is a state-of-the-art dynamic analysis tool.

- Use dynamic analysis to collect program behaviors
- Leverage heuristics such as field accesses are performed by first loading the base address of the data structure and then offsetting.

However,

- Its effectiveness hinges on the availability of high quality inputs.
- The heuristics do not always hold.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Howard

Howard is a state-of-the-art dynamic analysis tool.

- Use dynamic analysis to collect program behaviors
- Leverage heuristics such as field accesses are performed by first loading the base address of the data structure and then offsetting.

However,

- Its effectiveness hinges on the availability of high quality inputs.
- The heuristics do not always hold.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Howard

Howard is a state-of-the-art dynamic analysis tool.

- Use dynamic analysis to collect program behaviors
- Leverage heuristics such as field accesses are performed by first loading the base address of the data structure and then offsetting.

However,

- Its effectiveness hinges on the availability of high quality inputs.
- The heuristics do not always hold.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Howard

Howard is a state-of-the-art dynamic analysis tool.

- Use dynamic analysis to collect program behaviors
- Leverage heuristics such as field accesses are performed by first loading the base address of the data structure and then offsetting.

However,

- Its effectiveness hinges on the availability of high quality inputs.
- The heuristics do not always hold.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

```
mov [rsp + 0x8], 0x0
mov [rsp + 0x10], 0x1
```

State-of-the-art: Howard

Howard is a state-of-the-art dynamic analysis tool.

- Use dynamic analysis to collect program behaviors
- Leverage heuristics such as field accesses are performed by first loading the base address of the data structure and then offsetting.

However,

- Its effectiveness hinges on the availability of high quality inputs.
- The heuristics do not always hold.

```
typedef struct {
  int64 s_0;
  int64 s_1;
} elem_t;

elem_t *p;
int64 v0;
int64 v1;
```

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

State-of-the-art: Howard

Howard is a state-of-the-art dynamic analysis tool.

- Use dynamic analysis to collect program behaviors
- Leverage heuristics such as field accesses are performed by first loading the base address of the data structure and then offsetting.

However,

- Its effectiveness hinges on the availability of high quality inputs.
- The heuristics do not always hold.

```
typedef struct {
  int64 s_0;
  int64 s_1;
} elem_t;

elem_t *p;
int64 v0;
int64 v1;
```

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

Observation 1: 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.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

***p** is likely to be structure.

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

```
p->x = v.x;
p->y = v.y;
```

***p** and **v** may be of the same data structure.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

Point-to Hint: if a pointer may point to two memory regions, these regions *may* be of the same data structure.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

Point-to Hint: if a pointer may point to two memory regions, these regions *may* be of the same data structure.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

p may point to both stack inlined variable **v** and heap-allocated variable **malloc(...)**, suggesting they may be of the same structure.

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

Point-to Hint: if a pointer may point to two memory regions, these regions *may* be of the same data structure.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

Point-to Hint: if a pointer may point to two memory regions, these regions *may* be of the same data structure.

Unified Access Hint: if an instruction may access two memory regions, they *may* be of the same data structure.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

Point-to Hint: if a pointer may point to two memory regions, these regions *may* be of the same data structure.

Unified Access Hint: if an instruction may access two memory regions, they *may* be of the same data structure.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

both the heap structure **`malloc(...)`**, and the stack structure **`v`** are accessed by the same instruction.

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

Point-to Hint: if a pointer may point to two memory regions, these regions *may* be of the same data structure.

Unified Access Hint: if an instruction may access two memory regions, they *may* be of the same data structure.

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

Point-to Hint: if a pointer may point to two memory regions, these regions *may* be of the same data structure.

Unified Access Hint: if an instruction may access two memory regions, they *may* be of the same data structure.

Other Hints: we also developed other hints (e.g., array hints)

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

Point-to Hint: if a pointer may point to two memory regions, these regions *may* be of the same data structure.

Unified Access Hint: if an instruction may access two memory regions, they *may* be of the same data structure.

Other Hints: we also developed other hints (e.g., array hints)

How to collect hints?

```
typedef struct {
  long x;
  long y;
} elem_t;

int main() {
 if (!rand(1000)) huft_build(...);
}

void huft_build(...) {
  elem_t *p, v = {.x=0, .y=1};

  if (...)
    p = &v;
  else {
    p = malloc(sizeof(elem_t));
    *p = v;
  }

  output(p->x, p->y);
}
```

BDA [OOPSLA’19]: Hint Collection

BDA [OOPSLA'19]: Hint Collection

- BDA is a path sampling driven per-path abstract interpretation technique.
  - For a given sampled path, BDA **ignores the path feasibility** and abstract interprets the program following the given path. <u>Hints can be collected during abstract interpretation</u>.
  - BDA uses **precise symbolic values** as it interprets individual paths separately, which makes it different from other abstract interpretation techniques like VSA that merges values across paths.

- BDA uses a sophisticated path sampling algorithm so that the different paths of a program can be sampled uniformly.
  - 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.

BDA [OOPSLA'19]: Hint Collection

- BDA is a path sampling driven per-path abstract interpretation technique.
  - For a given sampled path, BDA **ignores the path feasibility** and abstract interprets the program following the given path. Hints can be collected during abstract interpretation.
  - BDA uses **precise symbolic values** as it interprets individual paths separately, which makes it different from other abstract interpretation techniques like VSA that merges values across paths.

- BDA uses a sophisticated path sampling algorithm so that the different paths of a program can be sampled uniformly.
  - 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.

BDA [OOPSLA'19]: Hint Collection

- BDA is a path sampling driven per-path abstract interpretation technique.
  - For a given sampled path, BDA **ignores the path feasibility** and abstract interprets the program following the given path. Hints can be collected during abstract interpretation.
  - BDA uses **precise symbolic values** as it interprets individual paths separately, which makes it different from other abstract interpretation techniques like VSA that merges values across paths.

- BDA uses a sophisticated path sampling algorithm so that the different paths of a program can be sampled uniformly.
  - 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.

BDA [OOPSLA'19]: Hint Collection

- BDA is a path sampling driven per-path abstract interpretation technique.
  - For a given sampled path, BDA **ignores the path feasibility** and abstract interprets the program following the given path. <u>Hints can be collected during abstract interpretation</u>.
  - BDA uses **precise symbolic values** as it interprets individual paths separately, which makes it different from other abstract interpretation techniques like VSA that merges values across paths.

- BDA uses a sophisticated path sampling algorithm so that the different paths of a program can be sampled uniformly.
  - 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.

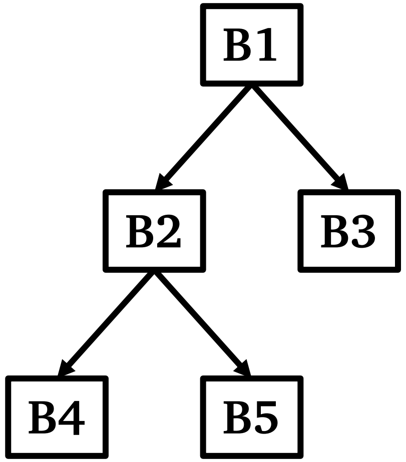
BDA [OOPSLA'19]: Hint Collection

- BDA is a path sampling driven per-path abstract interpretation technique.
  - For a given sampled path, BDA **ignores the path feasibility** and abstract interprets the program following the given path. Hints can be collected during abstract interpretation.
  - BDA uses **precise symbolic values** as it interprets individual paths separately, which makes it different from other abstract interpretation techniques like VSA that merges values across paths.

- BDA uses a sophisticated path sampling algorithm so that the different paths of a program can be sampled uniformly.
  - 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.

BDA [OOPSLA'19]: Hint Collection

- BDA is a path sampling driven per-path abstract interpretation technique.
  - For a given sampled path, BDA **ignores the path feasibility** and abstract interprets the program following the given path. <u>Hints can be collected during abstract interpretation</u>.
  - BDA uses **precise symbolic values** as it interprets individual paths separately, which makes it different from other abstract interpretation techniques like VSA that merges values across paths.

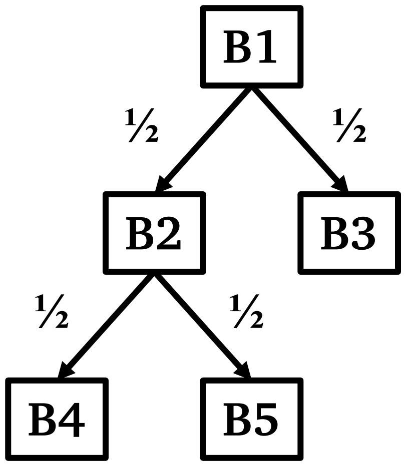


- BDA uses a sophisticated path sampling algorithm so that the different paths of a program can be sampled uniformly.
  - Simply tossing a fair coin at each predicate leads to a distribution that is substantially biased towards short paths.
  - Uniform sampling allows exploring a lot more long paths.

BDA [OOPSLA'19]: Hint Collection

- BDA is a path sampling driven per-path abstract interpretation technique.
  - For a given sampled path, BDA **ignores the path feasibility** and abstract interprets the program following the given path. Hints can be collected during abstract interpretation.
  - BDA uses **precise symbolic values** as it interprets individual paths separately, which makes it different from other abstract interpretation techniques like VSA that merges values across paths.

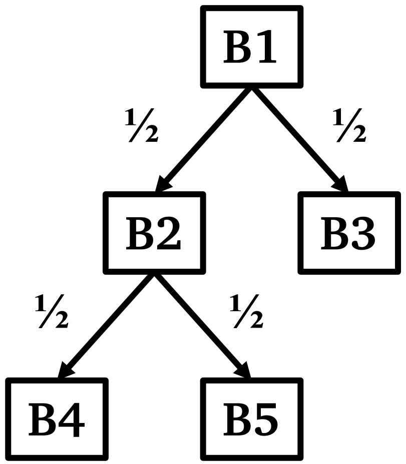
- BDA uses a sophisticated path sampling algorithm so that the different paths of a program can be sampled uniformly.
  - Simply tossing a fair coin at each predicate leads to a distribution that is substantially biased towards short paths.
  - Uniform sampling allows exploring a lot more long paths.



BDA [OOPSLA'19]: Hint Collection

- BDA is a path sampling driven per-path abstract interpretation technique.
  - For a given sampled path, BDA **ignores the path feasibility** and abstract interprets the program following the given path. <u>Hints can be collected during abstract interpretation</u>.
  - BDA uses **precise symbolic values** as it interprets individual paths separately, which makes it different from other abstract interpretation techniques like VSA that merges values across paths.

- BDA uses a sophisticated path sampling algorithm so that the different paths of a program can be sampled uniformly.
  - 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.



Naïve Sampling:

B1 – B3: ½

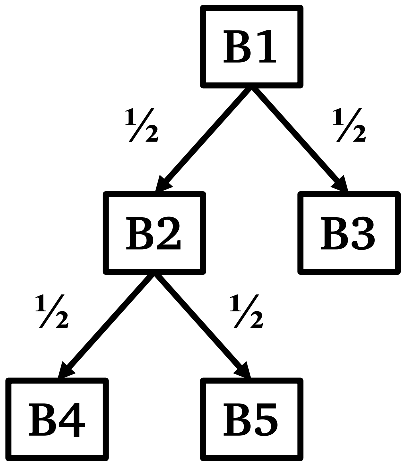
B1 – B2 – B4: ¼

B1 – B2 – B5: ¼

BDA [OOPSLA'19]: Hint Collection

- BDA is a path sampling driven per-path abstract interpretation technique.
  - For a given sampled path, BDA **ignores the path feasibility** and abstract interprets the program following the given path. Hints can be collected during abstract interpretation.
  - BDA uses **precise symbolic values** as it interprets individual paths separately, which makes it different from other abstract interpretation techniques like VSA that merges values across paths.

- BDA uses a sophisticated path sampling algorithm so that the different paths of a program can be sampled uniformly.
  - 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.



Naïve Sampling:

B1 – B3: ½

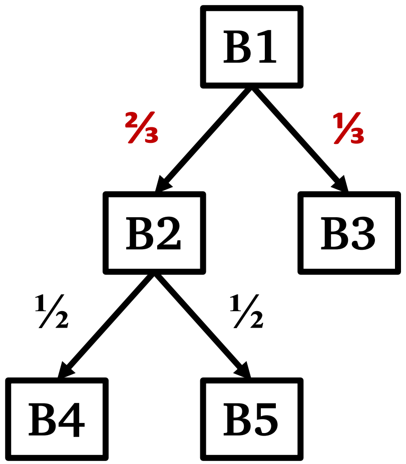
B1 – B2 – B4: ¼

B1 – B2 – B5: ¼

BDA [OOPSLA'19]: Hint Collection

- BDA is a path sampling driven per-path abstract interpretation technique.
  - For a given sampled path, BDA **ignores the path feasibility** and abstract interprets the program following the given path. <u>Hints can be collected during abstract interpretation</u>.
  - BDA uses **precise symbolic values** as it interprets individual paths separately, which makes it different from other abstract interpretation techniques like VSA that merges values across paths.

- BDA uses a sophisticated path sampling algorithm so that the different paths of a program can be sampled uniformly.
  - 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.



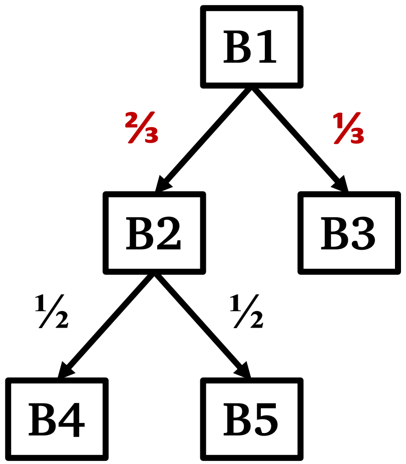
Naïve Sampling:

B1 – B3: ½
B1 – B2 – B4: ¼
B1 – B2 – B5: ¼

BDA [OOPSLA'19]: Hint Collection

- BDA is a path sampling driven per-path abstract interpretation technique.
  - For a given sampled path, BDA **ignores the path feasibility** and abstract interprets the program following the given path. Hints can be collected during abstract interpretation.
  - BDA uses **precise symbolic values** as it interprets individual paths separately, which makes it different from other abstract interpretation techniques like VSA that merges values across paths.

- BDA uses a sophisticated path sampling algorithm so that the different paths of a program can be sampled uniformly.
  - 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.



Naïve Sampling:

B1 – B3: ½
B1 – B2 – B4: ¼
B1 – B2 – B5: ¼

BDA Sampling:

B1 – B3: ⅓
B1 – B2 – B4: ⅓
B1 – B2 – B5: ⅓

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

Point-to Hint: if a pointer may point to two memory regions, these regions *may* be of the same data structure.

Unified Access Hint: if an instruction may access two memory regions, they *may* be of the same data structure.

Other Hints: we also developed other hints (e.g., array hints)

Our Techniques

Observation 1: 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.

Allocation Hint: a heap-allocated variable is *likely* to be a structure or an array of structure.

Dataflow Hint: two memory regions connected by a direct dataflow *may* be of the same data structure.

Point-to Hint: if a pointer may point to two memory regions, these regions *may* be of the same data structure.

Unified Access Hint: if an instruction may access two memory regions, they *may* be of the same data structure.

Other Hints: we also developed other hints (e.g., array hints)

→ ***Uncertainty***

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

AllocationHint

PointToHint

DataFlowHint

UnifiedAccessHint

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

AllocationHint **0.9**

PointToHint **0.7**

DataFlowHint **0.6**

UnifiedAccessHint **0.6**

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

AllocationHint **0.9**

PointToHint **0.7**

DataFlowHint **0.6**

UnifiedAccessHint **0.6**

Probabilistic Inference

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

AllocationHint	0.9
PointToHint	0.7
DataFlowHint	0.6
UnifiedAccessHint	0.6

Probabilistic Inference

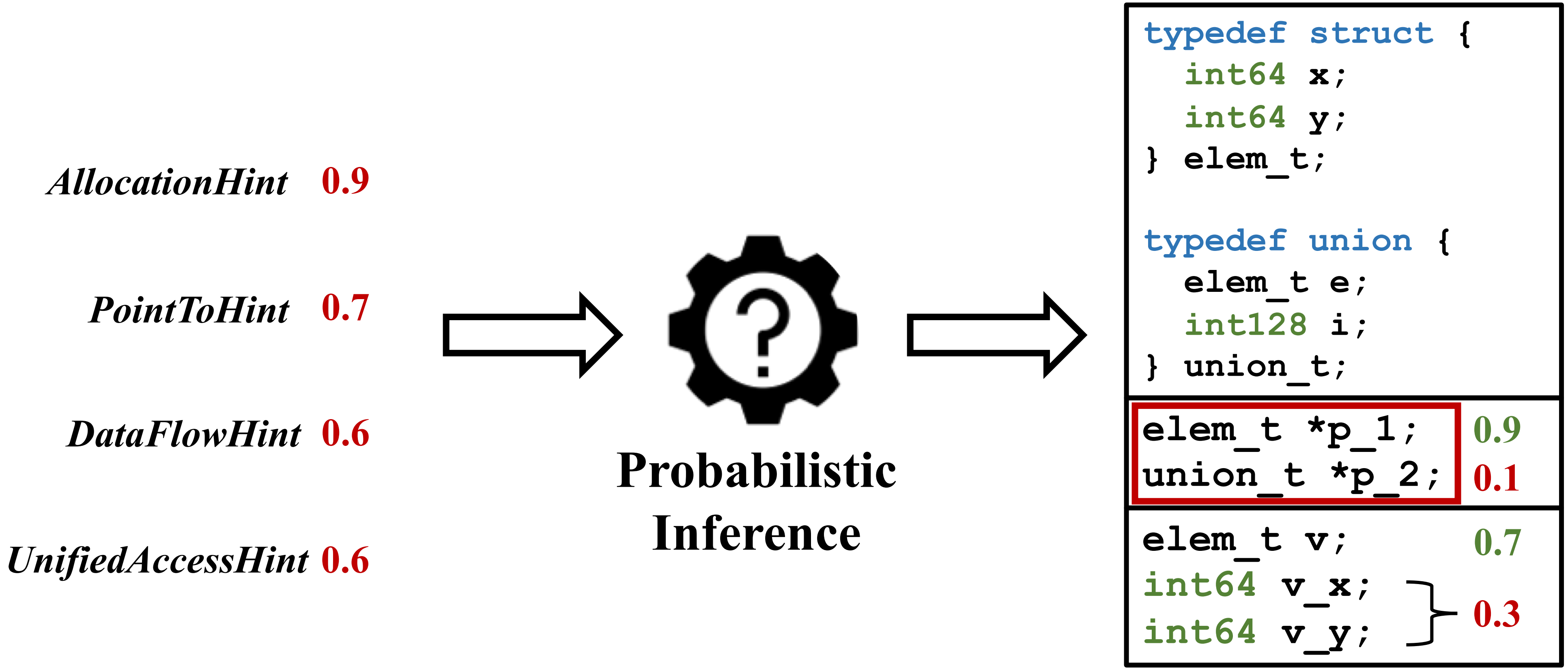
```
typedef struct {
  int64 x;
  int64 y;
} elem_t;

typedef union {
  elem_t e;
  int128 i;
} union_t;
```

```
elem_t *p_1;    0.9
union_t *p_2;   0.1
```

```
elem_t v;       0.7
int64 v_x;  ┐
int64 v_y;  ┘   0.3
```

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.



Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Random Variables: Random variables are introduced to describe variable properties, type properties, and structural properties.

- They are comprehensive and can be used to express the needed symbol information

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Random Variables: Random variables are introduced to describe variable properties, type properties, and structural properties.

- They are comprehensive and can be used to express the needed symbol information

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Random Variables: Random variables are introduced to describe variable properties, type properties, and structural properties.

- They are comprehensive and can be used to express the needed symbol information

PrimitiveVar(**`a`**, **`s`**): a memory region at address **`a`** with size **`s`** is a primitive variable.

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Random Variables: Random variables are introduced to describe variable properties, type properties, and structural properties.

- They are comprehensive and can be used to express the needed symbol information

PrimitiveVar(**`a`**, **`s`**): a memory region at address **`a`** with size **`s`** is a primitive variable.

FieldOf(**`a1`**, **`s`**, **`a2`**): a memory region at address **`a1`** with size **`s`** is a field of a structure at address **`a2`**.

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Random Variables: Random variables are introduced to describe variable properties, type properties, and structural properties.

- They are comprehensive and can be used to express the needed symbol information

PrimitiveVar(`a`, `s`): a memory region at address `a` with size `s` is a primitive variable.

FieldOf(`a1`, `s`, `a2`): a memory region at address `a1` with size `s` is a field of a structure at address `a2`.

HomoRegion(`a1`, `a2`, `s`): the memory regions starting at address `a1` and `a2` (with size `s`) have homogeneous structures.

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Random Variables: Random variables are introduced to describe variable properties, type properties, and structural properties.

PrimitiveVar(`a`, `s`) ***FieldOf***(`a1`, `s`, `a2`) ***HomoRegion***(`a1`, `a2`, `s`)

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Random Variables: Random variables are introduced to describe variable properties, type properties, and structural properties.

PrimitiveVar(**`a`**, **`s`**) ***FieldOf***(**`a1`**, **`s`**, **`a2`**) ***HomoRegion***(**`a1`**, **`a2`**, **`s`**)

Probabilistic Constraints: Probabilistic constraints essentially denote probability functions over the random variables involved.

- Correlate these random variables
- Calculate the posterior marginal probabilities for random variables
- The inference denoted by a constraint is probabilistic

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Random Variables: Random variables are introduced to describe variable properties, type properties, and structural properties.

PrimitiveVar(`a`, `s`) ***FieldOf***(`a1`, `s`, `a2`) ***HomoRegion***(`a1`, `a2`, `s`)

Probabilistic Constraints: Probabilistic constraints essentially denote probability functions over the random variables involved.

- Correlate these random variables
- Calculate the posterior marginal probabilities for random variables
- The inference denoted by a constraint is probabilistic

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Random Variables: Random variables are introduced to describe variable properties, type properties, and structural properties.

PrimitiveVar(**a**, **s**) ***FieldOf***(**a1**, **s**, **a2**) ***HomoRegion***(**a1**, **a2**, **s**)

Probabilistic Constraints: Probabilistic constraints essentially denote probability functions over the random variables involved.

- Correlate these random variables
- Calculate the posterior marginal probabilities for random variables
- The inference denoted by a constraint is probabilistic

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Random Variables: Random variables are introduced to describe variable properties, type properties, and structural properties.

PrimitiveVar(**`a`**, **`s`**) ***FieldOf***(**`a1`**, **`s`**, **`a2`**) ***HomoRegion***(**`a1`**, **`a2`**, **`s`**)

Probabilistic Constraints: Probabilistic constraints essentially denote probability functions over the random variables involved.

- Correlate these random variables
- Calculate the posterior marginal probabilities for random variables
- The inference denoted by a constraint is probabilistic

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Random Variables: Random variables are introduced to describe variable properties, type properties, and structural properties.

PrimitiveVar(`a`, `s`)　　***FieldOf***(`a1`, `s`, `a2`)　　***HomoRegion***(`a1`, `a2`, `s`)

Probabilistic Constraints: Probabilistic constraints essentially denote probability functions over the random variables involved.

- Correlate these random variables
- Calculate the posterior marginal probabilities for random variables
- The inference denoted by a constraint is probabilistic

$$\textbf{\textit{DataFlowHint}}(\texttt{a1}, \texttt{a2}, \texttt{s}) \xrightarrow{\textbf{\textit{0.6}}} \textbf{\textit{HomoRegion}}(\texttt{a1}, \texttt{a2}, \texttt{s})$$

Our Techniques

Observation 2: the various kinds of hints in variable/structure recovery can be integrated in a more sophisticated manner using probabilistic inference.

Random Variables: Random variables are introduced to describe variable properties, type properties, and structural properties.

PrimitiveVar(`a`, `s`) ***FieldOf***(`a1`, `s`, `a2`) ***HomoRegion***(`a1`, `a2`, `s`)

Probabilistic Constraints: Probabilistic constraints essentially denote probability functions over the random variables involved.

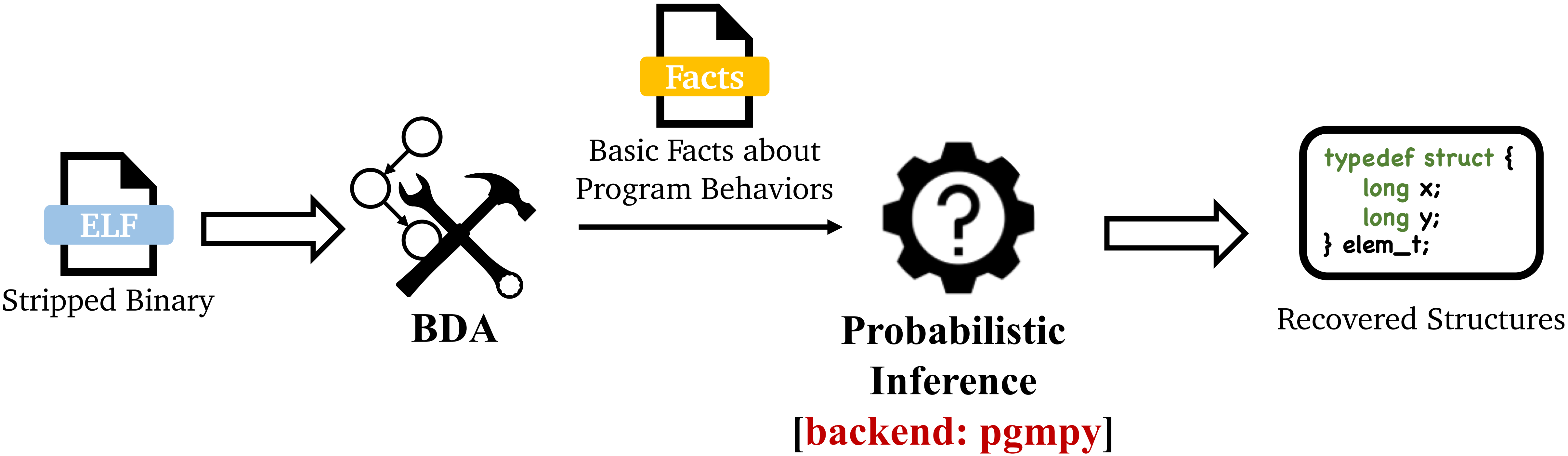
- Correlate these random variables
- Calculate the posterior marginal probabilities for random variables
- The inference denoted by a constraint is probabilistic

$$\textit{DataFlowHint}(\texttt{a1}, \texttt{a2}, \texttt{s}) \xrightarrow{0.6} \textit{HomoRegion}(\texttt{a1}, \texttt{a2}, \texttt{s})$$

$$\textit{FieldOf}(\texttt{a1}, \texttt{s}, \texttt{a2}) \wedge \textit{HomoRegion}(\texttt{a2}, \texttt{a3}, \texttt{\_}) \xrightarrow{0.9} \textit{FieldOf}(\texttt{a3+(a1-a2)}, \texttt{s}, \texttt{a3})$$

Osprey

Osprey 🦅


Facts
Basic Facts about Program Behaviors
ELF
Stripped Binary
BDA
Probabilistic Inference
[backend: pgmpy]
typedef struct {
long x;
long y;
} elem_t;
Recovered Structures

Evaluation

Evaluation

Benchmark:

- CoreUtils (O0 and O3)
- Benchmark used by Howard Project (O0 and O3)

Ground Truth:

- debugging information

Evaluation

Benchmark:

- CoreUtils (O0 and O3)
- Benchmark used by Howard Project (O0 and O3)

Ground Truth:

- debugging information

Evaluation

Benchmark:

- CoreUtils (O0 and O3)
- Benchmark used by Howard Project (O0 and O3)

Ground Truth:

- debugging information

Evaluation

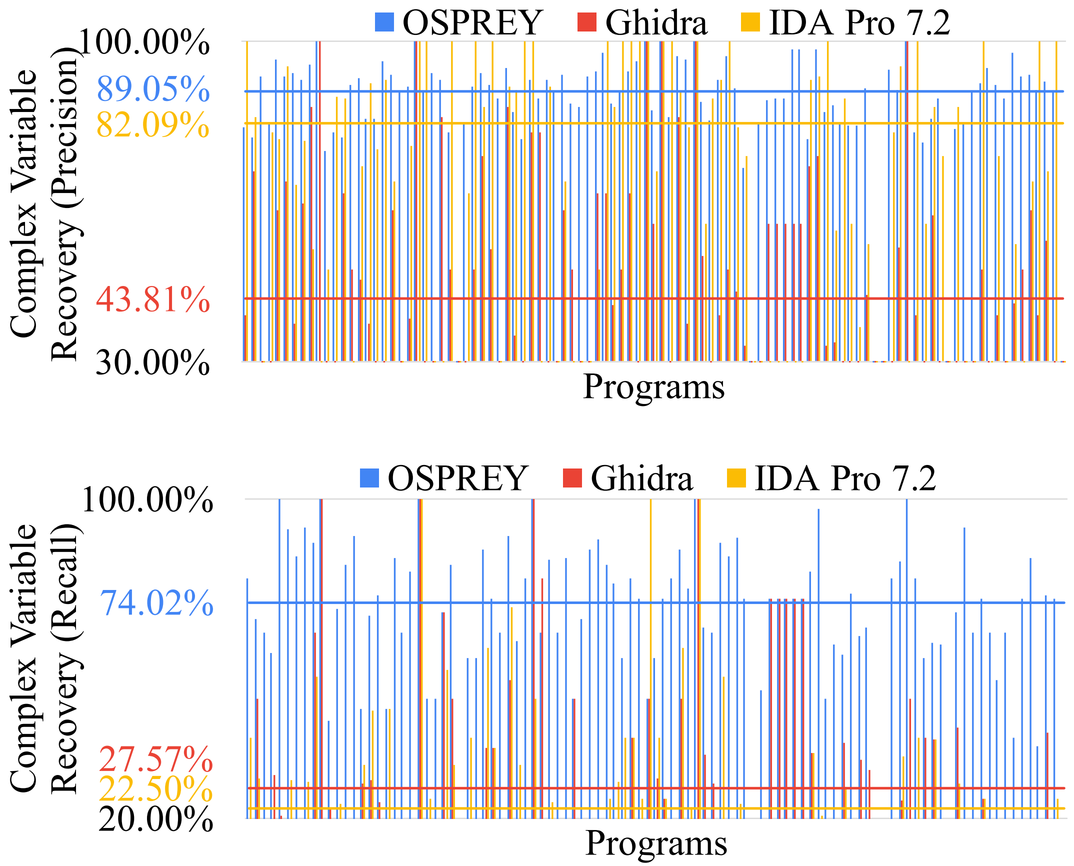
Benchmark:

- CoreUtils (O0 and O3)
- Benchmark used by Howard Project (O0 and O3)

Ground Truth:

- debugging information

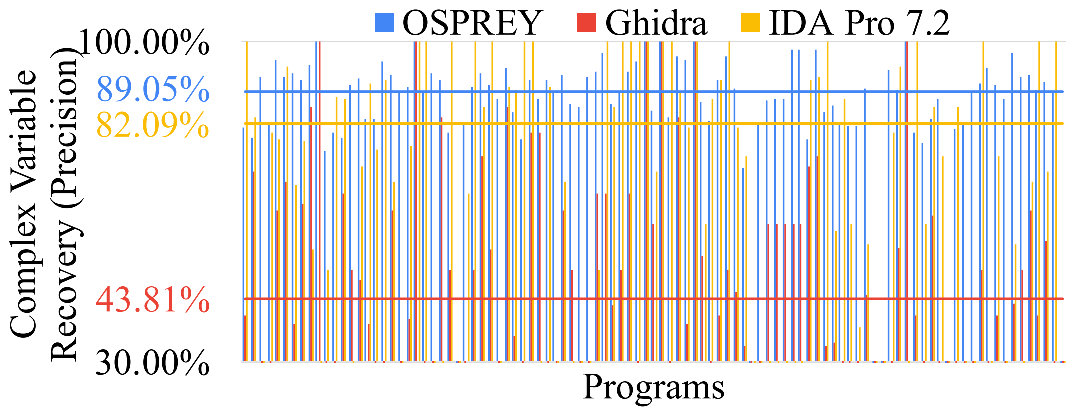
Evaluation: Complex Variable Recovery on CoreUtils

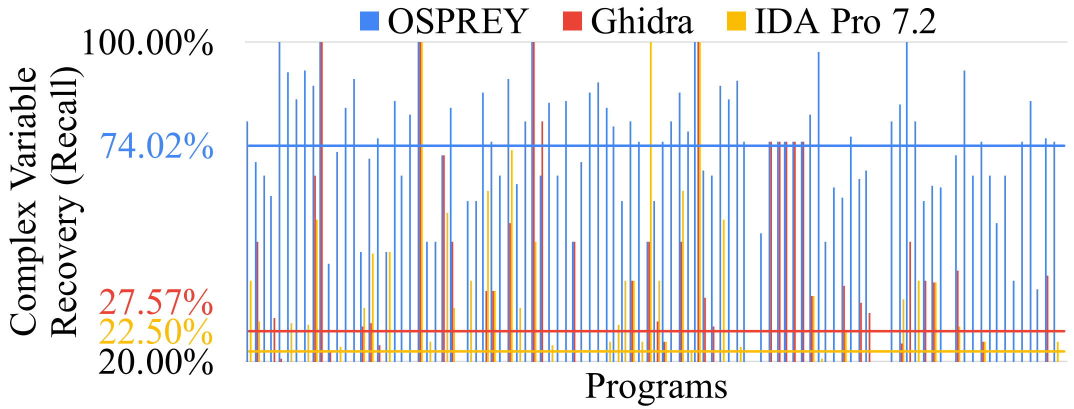


Evaluation: Complex Variable Recovery on CoreUtils

Similar to the standard in the literature, we inspect individual variables.

- Specially, if it is a pointer type, we inspect the structure that is being pointed to.
- Complex variables include structures, unions, arrays and pointers to structures and unions.

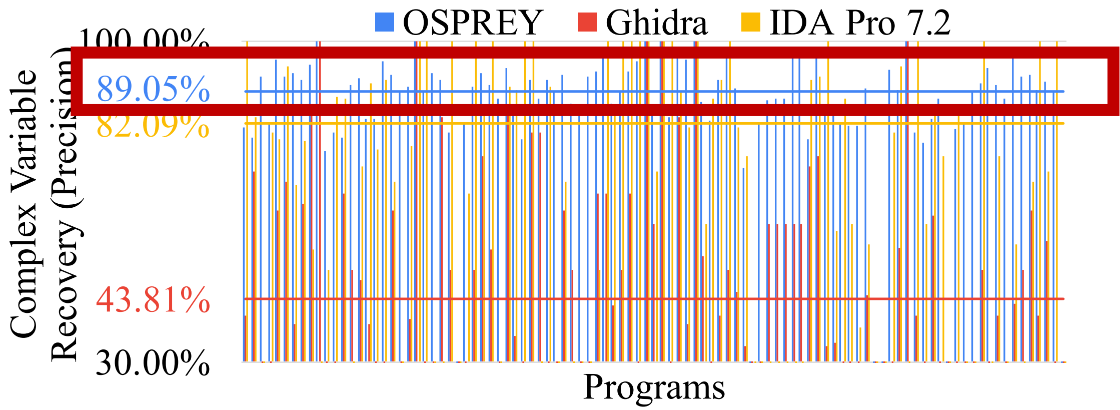


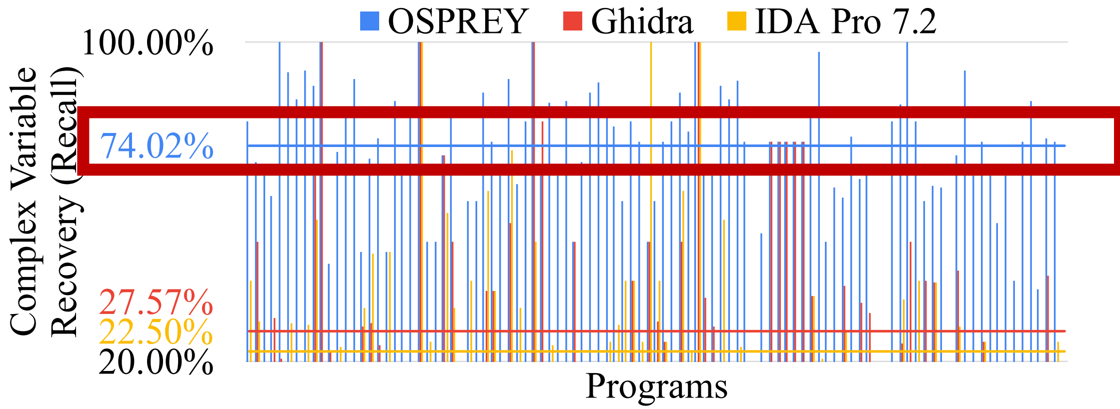


Evaluation: Complex Variable Recovery on CoreUtils

Similar to the standard in the literature, we inspect individual variables.

- Specially, if it is a pointer type, we inspect the structure that is being pointed to.
- Complex variables include structures, unions, arrays and pointers to structures and unions.

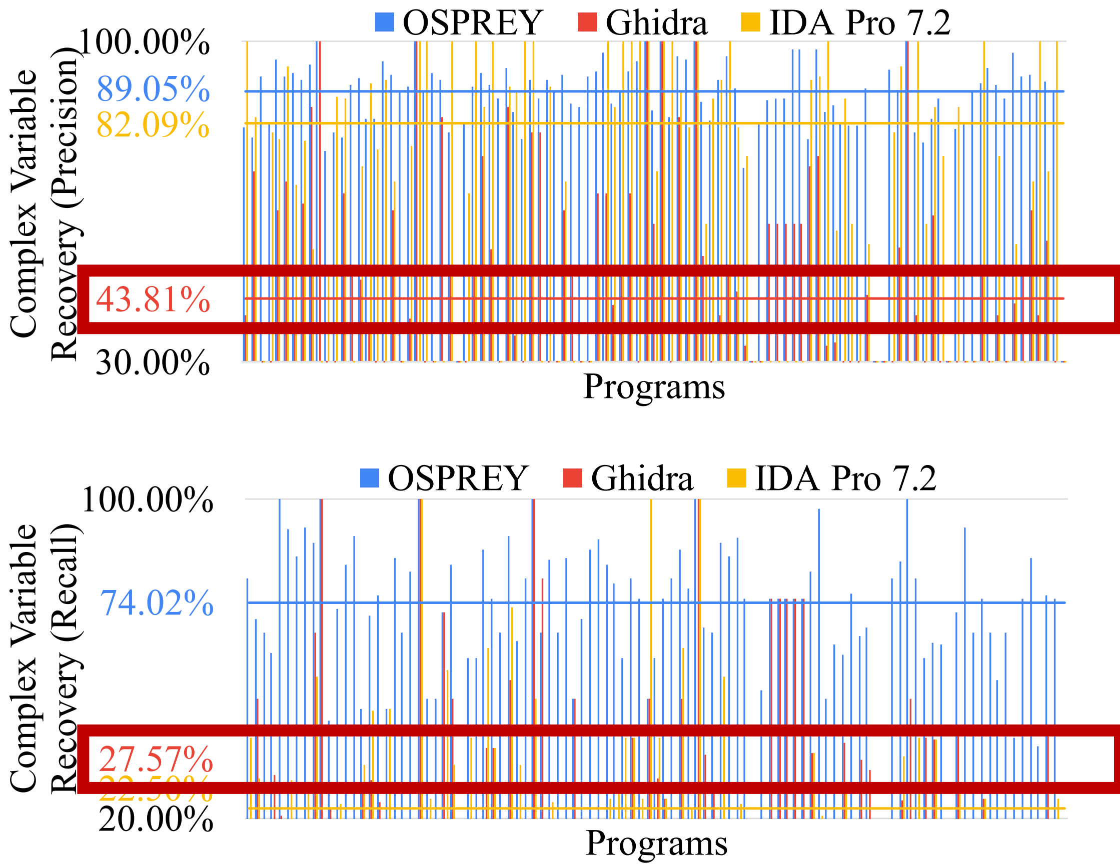




Evaluation: Complex Variable Recovery on CoreUtils

Similar to the standard in the literature, we inspect individual variables.

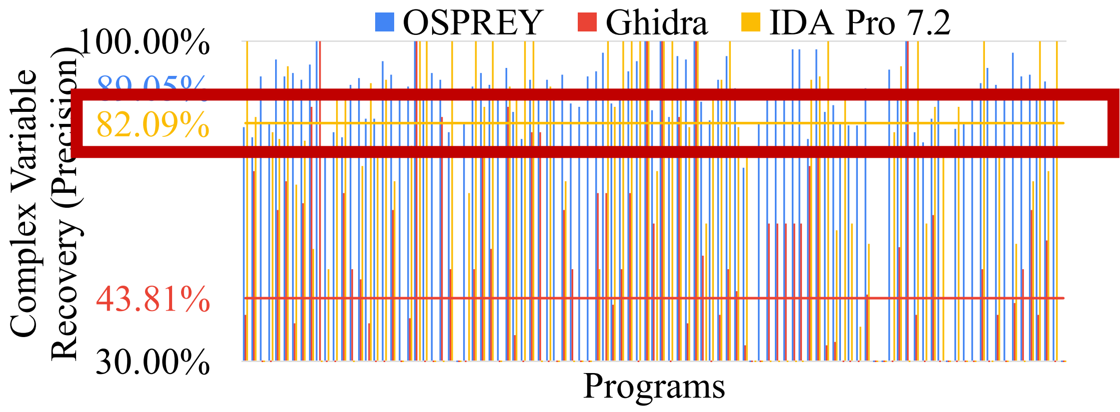
- Specially, if it is a pointer type, we inspect the structure that is being pointed to.
- Complex variables include structures, unions, arrays and pointers to structures and unions.

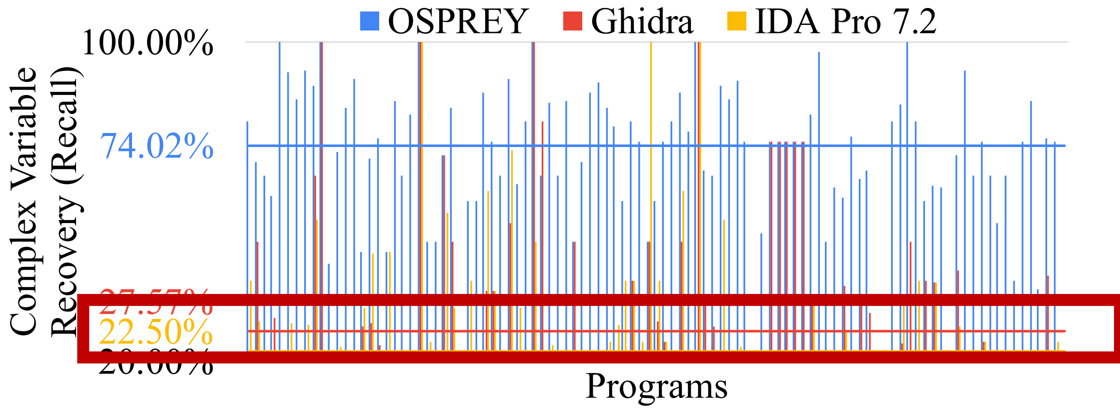


Evaluation: Complex Variable Recovery on CoreUtils

Similar to the standard in the literature, we inspect individual variables.

- Specially, if it is a pointer type, we inspect the structure that is being pointed to.
- Complex variables include structures, unions, arrays and pointers to structures and unions.

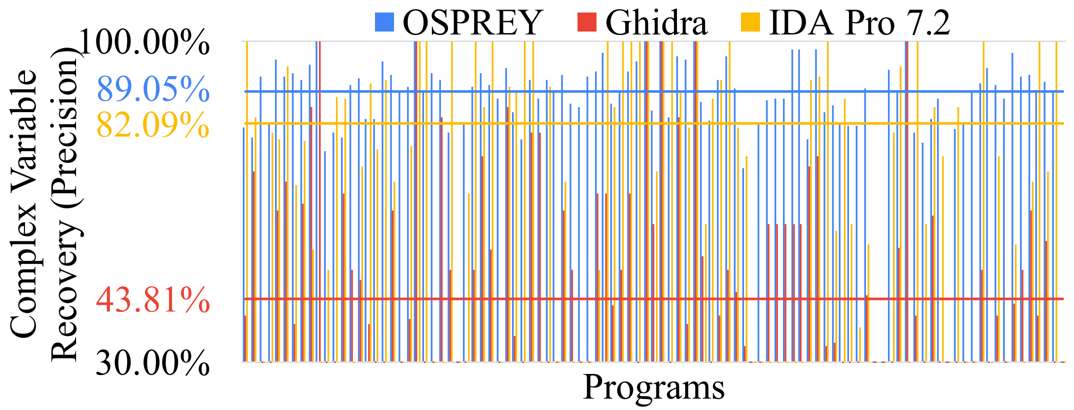


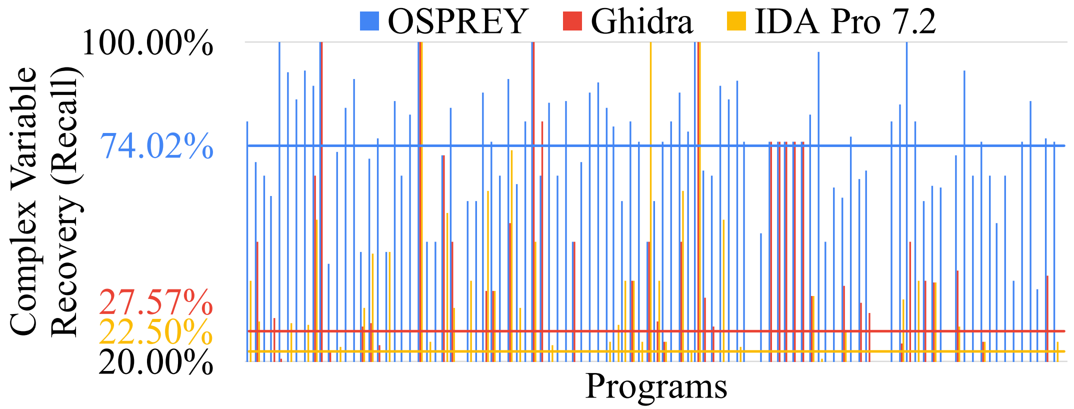


Evaluation: Complex Variable Recovery on CoreUtils

Similar to the standard in the literature, we inspect individual variables.

- Specially, if it is a pointer type, we inspect the structure that is being pointed to.
- Complex variables include structures, unions, arrays and pointers to structures and unions.



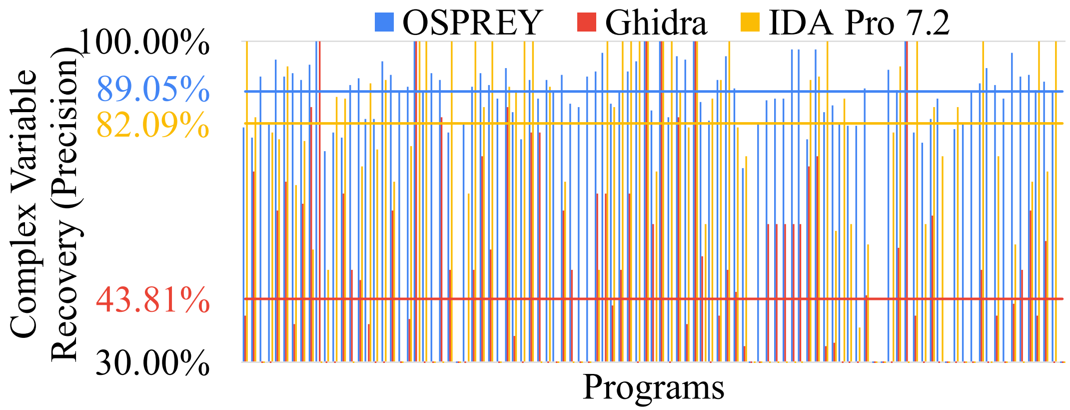


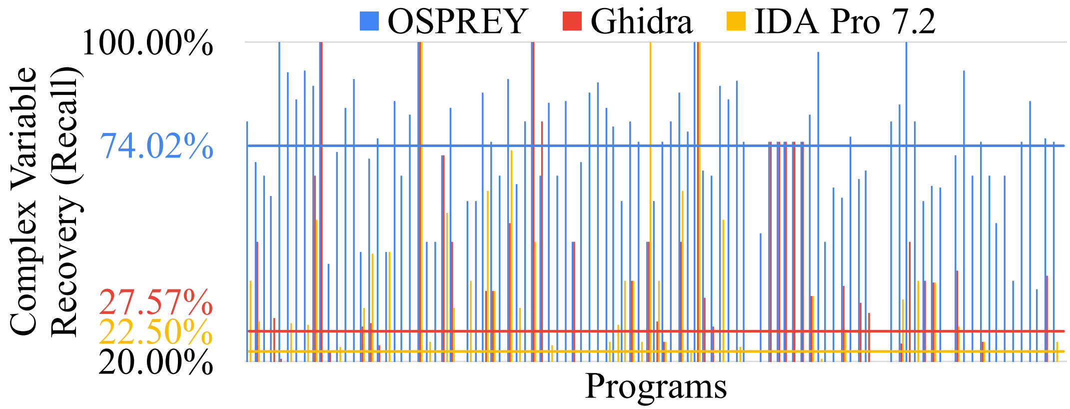
Evaluation: Complex Variable Recovery on CoreUtils

Similar to the standard in the literature, we inspect individual variables.

- Specially, if it is a pointer type, we inspect the structure that is being pointed to.
- Complex variables include structures, unions, arrays and pointers to structures and unions.

- 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.
    - We inspect why such improvement, and the main reason is that IDA and Ghidra do not perform complex analysis on recover structure while most complex variables are structure and pointers to structures.



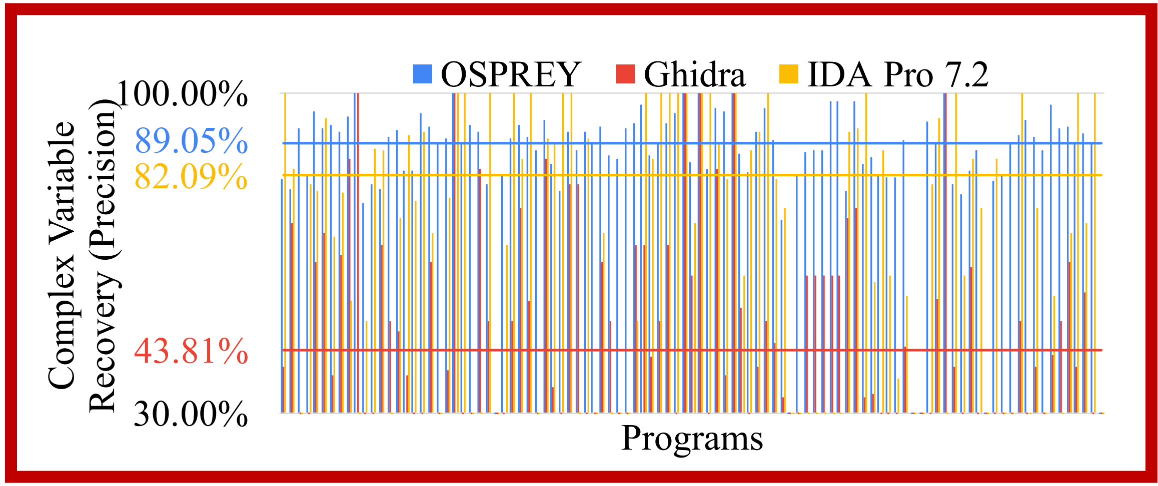


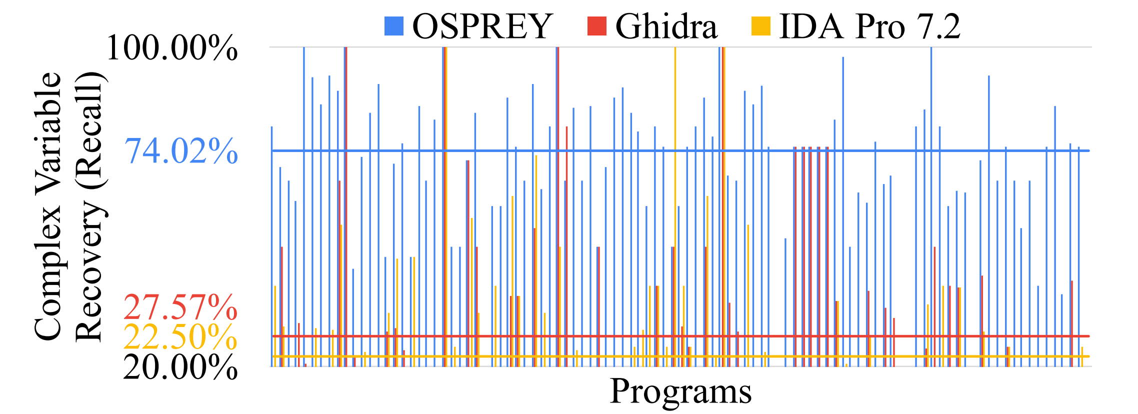
Evaluation: Complex Variable Recovery on CoreUtils

Similar to the standard in the literature, we inspect individual variables.

- Specially, if it is a pointer type, we inspect the structure that is being pointed to.
- Complex variables include structures, unions, arrays and pointers to structures and unions.

- 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.
  - We inspect why such improvement, and the main reason is that IDA and Ghidra do not perform complex analysis on recover structure while most complex variables are structure and pointers to structures.



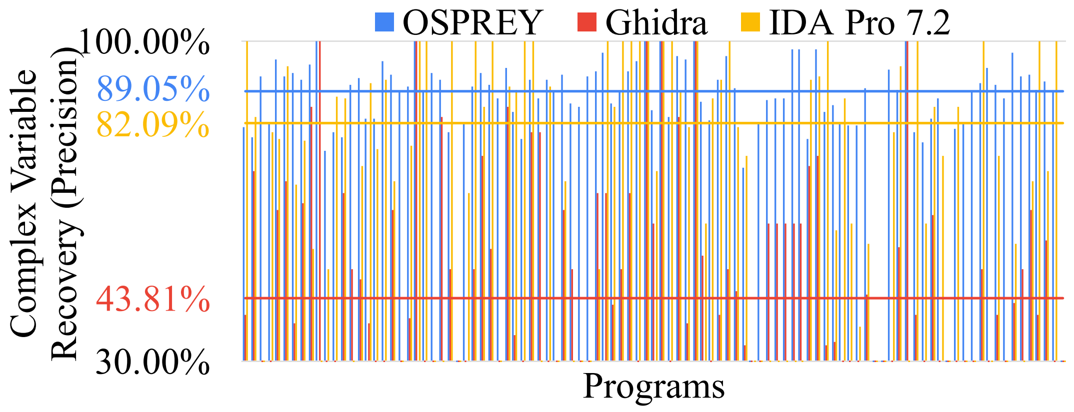


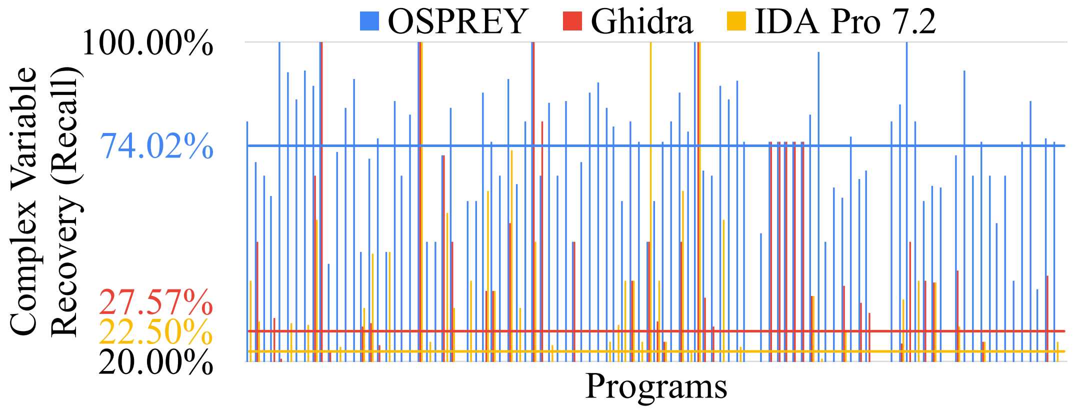
Evaluation: Complex Variable Recovery on CoreUtils

Similar to the standard in the literature, we inspect individual variables.

- Specially, if it is a pointer type, we inspect the structure that is being pointed to.
- Complex variables include structures, unions, arrays and pointers to structures and unions.

- 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.
  - We inspect why such improvement, and the main reason is that IDA and Ghidra do not perform complex analysis on recover structure while most complex variables are structure and pointers to structures.





Evaluation: Argument Decompilation Results

```c
int network_rxxx(server *srv)
{
  server *v1; // rbx
  int result; // rax
  size_t v3; // rbp
  server_socket *v4; // r12
  fdnode *v5; // rax
  fdevents *v6; // rdi

  v1 = srv;
  result = fdevent_sxxx(srv->ev);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !srv->sockets_disabled )
    {
      while ( v1->srv_sockets.
                    used > v3 )
      {
        v4 = v1->srv_sockets.
                 ptr[v3++];
        v5 = fdevent_gxxx(
                 v1->ev,
                 v4->fd,
                 network_sxxx, v4
        );
        v6 = v1->ev;
        v4->fdn = v5;
        fdevent_fxxx(v6, v5, 1);
      }
    }
    result = 0LL;
  }
  return result;
}
```

IDA Pro w/ debug information

```c
__int32 __fastcall sub_D840(__int64 a1)
{
  _QWORD *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  __int64 v4; // r12
  __int64 v5; // rax
  __int64 v6; // rdi

  v1 = (_QWORD *)a1;
  result = sub_12B7A(*(_QWORD *)(a1 + 24));
  if ( (_DWORD)result != -1 )
  {
    v3 = 0LL;
    if ( !*(_DWORD *)(a1 + 100) )
    {
      while ( v1[2] > v3 )

      {
        v4 = *(_QWORD *)(*v1 + 8 * v3++);

        v5 = sub_21860(
               v1[3],
               *(unsigned int *)(v4 + 112),
               sub_18F30, v4
        );
        v6 = v1[3];
        *(_QWORD *)(v4 + 120) = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

Vanilla IDA Pro

```c
__int32 __fastcall sub_D840(struct_C264 *a1)
{
  struct_C264 *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  struct_CF4A *v4; // r12
  struct_12A42 *v5; // rax
  struct_12A0E *v6; // rdi

  v1 = a1;
  result = sub_12B7A(a1->ptr_field_28);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !a1->dat_field_74 )
    {
      while ( v1->dat_field_10 > v3 )

      {
        v4 = v1->ptr_ptr_field_0[v3++];

        v5 = sub_21860(
                 v1->ptr_field_28,
                 v4->dat_field_10,
                 sub_18F30, v4
        );
        v6 = v1->ptr_field_28;
        v4->ptr_field_18 = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

IDA Pro w/ our tool

Evaluation: Argument Decompilation Results

```
int network_rxxx(server *srv)
{
  server *v1; // rbx
  int result; // rax
  size_t v3; // rbp
  server_socket *v4; // r12
  fdnode *v5; // rax
  fdevents *v6; // rdi

  v1 = srv;
  result = fdevent_sxxx(srv->ev);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !srv->sockets_disabled )
    {
      while ( v1->srv_sockets.
                    used > v3 )
      {
        v4 = v1->srv_sockets.
                  ptr[v3++];
        v5 = fdevent_gxxx(
                  v1->ev,
                  v4->fd,
                  network_sxxx, v4
        );
        v6 = v1->ev;
        v4->fdn = v5;
        fdevent_fxxx(v6, v5, 1);
      }
    }
    result = 0LL;
  }
  return result;
}
```

IDA Pro w/ debug information

```
__int32 __fastcall sub_D840(__int64 a1)
{
  _QWORD *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  __int64 v4; // r12
  __int64 v5; // rax
  __int64 v6; // rdi

  v1 = (_QWORD *)a1;
  result = sub_12B7A(*(_QWORD *)(a1 + 24));
  if ( (_DWORD)result != -1 )
  {
    v3 = 0LL;
    if ( !*(_DWORD *)(a1 + 100) )
    {
      while ( v1[2] > v3 )
      {
        v4 = *(_QWORD *)(*v1 + 8 * v3++);

        v5 = sub_21860(
               v1[3],
               *(unsigned int *)(v4 + 112),
               sub_18F30, v4
        );
        v6 = v1[3];
        *(_QWORD *)(v4 + 120) = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

Vanilla IDA Pro

```
__int32 __fastcall sub_D840(struct_C264 *a1)
{
  struct_C264 *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  struct_CF4A *v4; // r12
  struct_12A42 *v5; // rax
  struct_12A0E *v6; // rdi

  v1 = a1;
  result = sub_12B7A(a1->ptr_field_28);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !a1->dat_field_74 )
    {
      while ( v1->dat_field_10 > v3 )

      {
        v4 = v1->ptr_ptr_field_0[v3++];

        v5 = sub_21860(
                 v1->ptr_field_28,
                 v4->dat_field_10,
                 sub_18F30, v4
        );
        v6 = v1->ptr_field_28;
        v4->ptr_field_18 = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

IDA Pro w/ our tool

Evaluation: Argument Decompilation Results

```
int network_rxxx(server *srv)
{
  server *v1; // rbx
  int result; // rax
  size_t v3; // rbp
  server_socket *v4; // r12
  fdnode *v5; // rax
  fdevents *v6; // rdi

  v1 = srv;
  result = fdevent_sxxx(srv->ev);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !srv->sockets_disabled )
    {
      while ( v1->srv_sockets.
                    used > v3 )
      {
        v4 = v1->srv_sockets.
                  ptr[v3++];
        v5 = fdevent_gxxx(
                  v1->ev,
                  v4->fd,
                  network_sxxx, v4
        );
        v6 = v1->ev;
        v4->fdn = v5;
        fdevent_fxxx(v6, v5, 1);
      }
    }
    result = 0LL;
  }
  return result;
}
```

IDA Pro w/ debug information

```
__int32 __fastcall sub_D840(__int64 a1)
{
  _QWORD *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  __int64 v4; // r12
  __int64 v5; // rax
  __int64 v6; // rdi

  v1 = (_QWORD *)a1;
  result = sub_12B7A(*(_QWORD *)(a1 + 24));
  if ( (_DWORD)result != -1 )
  {
    v3 = 0LL;
    if ( !*(_DWORD *)(a1 + 100) )
    {
      while ( v1[2] > v3 )

      {
        v4 = *(_QWORD *)(*v1 + 8 * v3++);

        v5 = sub_21860(
               v1[3],
               *(unsigned int *)(v4 + 112),
               sub_18F30, v4
        );
        v6 = v1[3];
        *(_QWORD *)(v4 + 120) = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

Vanilla IDA Pro

```
__int32 __fastcall sub_D840(struct_C264 *a1)
{
  struct_C264 *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  struct_CF4A *v4; // r12
  struct_12A42 *v5; // rax
  struct_12A0E *v6; // rdi

  v1 = a1;
  result = sub_12B7A(a1->ptr_field_28);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !a1->dat_field_74 )
    {
      while ( v1->dat_field_10 > v3 )

      {
        v4 = v1->ptr_ptr_field_0[v3++];

        v5 = sub_21860(
                  v1->ptr_field_28,
                  v4->dat_field_10,
                  sub_18F30, v4
        );
        v6 = v1->ptr_field_28;
        v4->ptr_field_18 = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

IDA Pro w/ our tool

Evaluation: Argument Decompilation Results

```c
int network_rxxx(server *srv)
{
  server *v1; // rbx
  int result; // rax
  size_t v3; // rbp
  server_socket *v4; // r12
  fdnode *v5; // rax
  fdevents *v6; // rdi

  v1 = srv;
  result = fdevent_sxxx(srv->ev);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !srv->sockets_disabled )
    {
      while ( v1->srv_sockets.
                    used > v3 )
      {
        v4 = v1->srv_sockets.
                  ptr[v3++];
        v5 = fdevent_gxxx(
                  v1->ev,
                  v4->fd,
                  network_sxxx, v4
        );
        v6 = v1->ev;
        v4->fdn = v5;
        fdevent_fxxx(v6, v5, 1);
      }
    }
    result = 0LL;
  }
  return result;
}
```

IDA Pro w/ debug information

```c
__int32 __fastcall sub_D840(__int64 a1)
{
  _QWORD *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  __int64 v4; // r12
  __int64 v5; // rax
  __int64 v6; // rdi

  v1 = (_QWORD *)a1;
  result = sub_12B7A(*(_QWORD *)(a1 + 24));
  if ( (_DWORD)result != -1 )
  {
    v3 = 0LL;
    if ( !*(_DWORD *)(a1 + 100) )
    {
      while ( v1[2] > v3 )

      {
        v4 = *(_QWORD *)(*v1 + 8 * v3++);

        v5 = sub_21860(
               v1[3],
               *(unsigned int *)(v4 + 112),
               sub_18F30, v4
        );
        v6 = v1[3];
        *(_QWORD *)(v4 + 120) = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

Vanilla IDA Pro

```c
__int32 __fastcall sub_D840(struct_C264 *a1)
{
  struct_C264 *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  struct_CF4A *v4; // r12
  struct_12A42 *v5; // rax
  struct_12A0E *v6; // rdi

  v1 = a1;
  result = sub_12B7A(a1->ptr_field_28);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !a1->dat_field_74 )
    {
      while ( v1->dat_field_10 > v3 )

      {
        v4 = v1->ptr_ptr_field_0[v3++];

        v5 = sub_21860(
                  v1->ptr_field_28,
                  v4->dat_field_10,
                  sub_18F30, v4
        );
        v6 = v1->ptr_field_28;
        v4->ptr_field_18 = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

IDA Pro w/ our tool

Evaluation: Argument Decompilation Results

```
int network_rxxx(server *srv)
{
  server *v1; // rbx
  int result; // rax
  size_t v3; // rbp
  server_socket *v4; // r12
  fdnode *v5; // rax
  fdevents *v6; // rdi

  v1 = srv;
  result = fdevent_sxxx(srv->ev);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !srv->sockets_disabled )
    {
      while ( v1->srv_sockets.
                    used > v3 )
      {
        v4 = v1->srv_sockets.
                 ptr[v3++];
        v5 = fdevent_gxxx(
                 v1->ev,
                 v4->fd,
                 network_sxxx, v4
        );
        v6 = v1->ev;
        v4->fdn = v5;
        fdevent_fxxx(v6, v5, 1);
      }
    }
    result = 0LL;
  }
  return result;
}
```

IDA Pro w/ debug information

```
__int32 __fastcall sub_D840(__int64 a1)
{
  _QWORD *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  __int64 v4; // r12
  __int64 v5; // rax
  __int64 v6; // rdi

  v1 = (_QWORD *)a1;
  result = sub_12B7A(*(_QWORD *)(a1 + 24));
  if ( (_DWORD)result != -1 )
  {
    v3 = 0LL;
    if ( !*(_DWORD *)(a1 + 100) )
    {
      while ( v1[2] > v3 )
      {
        v4 = *(_QWORD *)(*v1 + 8 * v3++);
        v5 = sub_21860(
               v1[3],
               *(unsigned int *)(v4 + 112),
               sub_18F30, v4
        );
        v6 = v1[3];
        *(_QWORD *)(v4 + 120) = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

Vanilla IDA Pro

```
__int32 __fastcall sub_D840(struct_C264 *a1)
{
  struct_C264 *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  struct_CF4A *v4; // r12
  struct_12A42 *v5; // rax
  struct_12A0E *v6; // rdi

  v1 = a1;
  result = sub_12B7A(a1->ptr_field_28);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !a1->dat_field_74 )
    {
      while ( v1->dat_field_10 > v3 )
      {
        v4 = v1->ptr_ptr_field_0[v3++];
        v5 = sub_21860(
                 v1->ptr_field_28,
                 v4->dat_field_10,
                 sub_18F30, v4
        );
        v6 = v1->ptr_field_28;
        v4->ptr_field_18 = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

IDA Pro w/ our tool

Evaluation: Argument Decompilation Results

```
int network_rxxx(server *srv)
{
  server *v1; // rbx
  int result; // rax
  size_t v3; // rbp
  server_socket *v4; // r12
  fdnode *v5; // rax
  fdevents *v6; // rdi

  v1 = srv;
  result = fdevent_sxxx(srv->ev);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !srv->sockets_disabled )
    {
      while ( v1->srv_sockets.
              used > v3 )
      {
        v4 = v1->srv_sockets.
             ptr[v3++];
        v5 = fdevent_gxxx(
             v1->ev,
             v4->fd,
             network_sxxx, v4
        );
        v6 = v1->ev;
        v4->fdn = v5;
        fdevent_fxxx(v6, v5, 1);
      }
    }
    result = 0LL;
  }
  return result;
}
```

IDA Pro w/ debug information

```
__int32 __fastcall sub_D840(__int64 a1)
{
  _QWORD *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  __int64 v4; // r12
  __int64 v5; // rax
  __int64 v6; // rdi

  v1 = (_QWORD *)a1;
  result = sub_12B7A(*(_QWORD *)(a1 + 24));
  if ( (_DWORD)result != -1 )
  {
    v3 = 0LL;
    if ( !*(_DWORD *)(a1 + 100) )
    {
      while ( v1[2] > v3 )
      {
        v4 = *(_QWORD *)(*v1 + 8 * v3++);
        v5 = sub_21860(
               v1[3],
               *(unsigned int *)(v4 + 112),
               sub_18F30, v4
        );
        v6 = v1[3];
        *(_QWORD *)(v4 + 120) = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

Vanilla IDA Pro

```
__int32 __fastcall sub_D840(struct_C264 *a1)
{
  struct_C264 *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  struct_CF4A *v4; // r12
  struct_12A42 *v5; // rax
  struct_12A0E *v6; // rdi

  v1 = a1;
  result = sub_12B7A(a1->ptr_field_28);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !a1->dat_field_74 )
    {
      while ( v1->dat_field_10 > v3 )
      {
        v4 = v1->ptr_ptr_field_0[v3++];
        v5 = sub_21860(
               v1->ptr_field_28,
               v4->dat_field_10,
               sub_18F30, v4
        );
        v6 = v1->ptr_field_28;
        v4->ptr_field_18 = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

IDA Pro w/ our tool

Evaluation: Argument Decompilation Results

```c
int network_rxxx(server *srv)
{
  server *v1; // rbx
  int result; // rax
  size_t v3; // rbp
  server_socket *v4; // r12
  fdnode *v5; // rax
  fdevents *v6; // rdi

  v1 = srv;
  result = fdevent_sxxx(srv->ev);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !srv->sockets_disabled )
    {
      while ( v1->srv_sockets.
                    used > v3 )
      {
        v4 = v1->srv_sockets.
                 ptr[v3++];
        v5 = fdevent_gxxx(
                 v1->ev,
                 v4->fd,
                 network_sxxx, v4
        );
        v6 = v1->ev;
        v4->fdn = v5;
        fdevent_fxxx(v6, v5, 1);
      }
    }
    result = 0LL;
  }
  return result;
}
```

IDA Pro w/ debug information

```c
__int32 __fastcall sub_D840(__int64 a1)
{
  _QWORD *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  __int64 v4; // r12
  __int64 v5; // rax
  __int64 v6; // rdi

  v1 = (_QWORD *)a1;
  result = sub_12B7A(*(_QWORD *)(a1 + 24));
  if ( (_DWORD)result != -1 )
  {
    v3 = 0LL;
    if ( !*(_DWORD *)(a1 + 100) )
    {
      while ( v1[2] > v3 )
      {
        v4 = *(_QWORD *)(*v1 + 8 * v3++);
        v5 = sub_21860(
               v1[3],
               *(unsigned int *)(v4 + 112),
               sub_18F30, v4
        );
        v6 = v1[3];
        *(_QWORD *)(v4 + 120) = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

Vanilla IDA Pro

```c
__int32 __fastcall sub_D840(struct_C264 *a1)
{
  struct_C264 *v1; // rbx
  __int32 result; // rax
  unsigned __int64 v3; // rbp
  struct_CF4A *v4; // r12
  struct_12A42 *v5; // rax
  struct_12A0E *v6; // rdi

  v1 = a1;
  result = sub_12B7A(a1->ptr_field_28);
  if ( result != -1 )
  {
    v3 = 0LL;
    if ( !a1->dat_field_74 )
    {
      while ( v1->dat_field_10 > v3 )
      {
        v4 = v1->ptr_ptr_field_0[v3++];
        v5 = sub_21860(
               v1->ptr_field_28,
               v4->dat_field_10,
               sub_18F30, v4
        );
        v6 = v1->ptr_field_28;
        v4->ptr_field_18 = v5;
        sub_219C0(v6, v5, 1);
      }
    }
    result = 0;
  }
  return result;
}
```

IDA Pro w/ our tool

Related Works

Variable recovery and type inference for stripped binary:

- Slowinska, Asia, Traian Stancescu, and Herbert Bos. "Howard: A Dynamic Excavator for Reverse Engineering Data Structures." *NDSS*. 2011.
- Shoshitaishvili, Yan, et al. "Sok:(state of) the art of war: Offensive techniques in binary analysis." *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.
- Lee, JongHyup, Thanassis Avgerinos, and David Brumley. "TIE: Principled reverse engineering of types in binary programs." (2011).
- Lin, Zhiqiang, Xiangyu Zhang, and Dongyan Xu. "Automatic reverse engineering of data structures from binary execution." *Proceedings of the 11th Annual Information Security Symposium*. 2010.

Probabilistic Program Analysis:

- Geldenhuys, Jaco, Matthew B. Dwyer, and Willem Visser. "Probabilistic symbolic execution." *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. 2012.
- Borges, Mateus, et al. "Iterative distribution-aware sampling for probabilistic symbolic execution." *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015.
- Kwiatkowska, Marta, Gethin Norman, and David Parker. "PRISM 4.0: Verification of probabilistic real-time systems." *International conference on computer aided verification*. Springer, Berlin, Heidelberg, 2011.
- Filieri, Antonio, Carlo Ghezzi, and Giordano Tamburrelli. "Run-time efficient probabilistic model checking." *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011.

We develop a novel probabilistic variable and data structure recovery technique for stripped binaries.

- It leverages an advanced data-flow technique, BDA, to collect program behaviors.
- It features using random variables to denote the likelihood of recovery results such that a large number of various kinds of hints can be integrated with the inherent uncertainty considered.
- It achieves the state-of-the-art inversing results.

We develop a novel probabilistic variable and data structure recovery technique for stripped binaries.

- It leverages an advanced data-flow technique, BDA, to collect program behaviors.
- It features using random variables to denote the likelihood of recovery results such that a large number of various kinds of hints can be integrated with the inherent uncertainty considered.
- It achieves the state-of-the-art inversing results.

Thanks!