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

Variable and Data Structure Recovery

Bug Detection
typedef struct {
    long x;
    long y;
} elem_t;
typedef struct {
    long x;
    long y;
} elem_t;
typedef struct {
    long x;
    long y;
} elem_t;
OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary
Compilation

Stripped Binary

typedef struct {
    long x;
    long y;
} elem_t;
OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary

**Source Code**

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

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

**Compilation**

**Stripped Binary**

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

structural information is lost during compilation
OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for 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

01. `typedef struct {
02.    long x;
03.    long y;
04. } elem_t;
05.
06. `int main() {
07.    if (!rand(1000)) huft_build(...);
08. }
09.
10. `void huft_build(...) {
11.    elem_t *p, v = {.x=0, .y=1};
12.    `if (...) 
13.        p = &v;
14.    `else {
15.        p = malloc(sizeof(elem_t));
16.        *p = v;
17.    } 
18. }
19. 
20. `output(p->x, p->y);
21. }`
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

```c
01. typedef struct {
02.   long x;
03.   long y;
04. } elem_t;
05.
06. int main() {
07.   if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...) {
11.   elem_t *p, v = {.x=0, .y=1};
12.
13.   if (...) 
14.     p = &v;
15.   else {
16.     p = malloc(sizeof(elem_t));
17.     *p = v;
18.   }
19.
20.   output(p->x, p->y);
21. }
```
Motivation Example

```c
01. typedef struct {
02.   long x;
03.   long y;
04. } elem_t;
05. 
06. int main() {
07.   if (!rand(1000)) huft_build(...);
08. }
09. 
10. void huft_build(...) {
11.   elem_t *p, v = {.x=0, .y=1};
12. 
13.   if (...) 
14.       p = &v;
15.   else {
16.       p = malloc(sizeof(elem_t));
17.       *p = v;
18.   }
19. 
20.   output(p->x, p->y);
21. }
```
Motivation Example

```c
01. typedef struct {
02.    long x;
03.    long y;
04. } elem_t;
05.
06. int main() {
07.    if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...) {
11.    elem_t *p, v = {.x=0, .y=1};
12.    if (...) 
13.        p = &v;
14.    else {
15.        p = malloc(sizeof(elem_t));
16.        *p = v;
17.    }
18. }
19.
20.    output(p->x, p->y);
21. }
```
Motivation Example

```c
01. typedef struct {
02.     long x;
03.     long y;
04. } elem_t;
05.
06. int main() {
07.     if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...) {
11.     elem_t *p, v = {.x=0, .y=1};
12.
13.     if (...) 
14.         p = &v;
15.     else {
16.         p = malloc(sizeof(elem_t));
17.         *p = v;
18.     }
19.
20.     output(p->x, p->y);
21. }
```
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
01. typedef struct {
02.     long x;
03.     long y;
04. } elem_t;
05.
06. int main() {
07.     if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...) {
11.     elem_t *p, v = {.x=0, .y=1};
12.
13.     if (...) 
14.         p = &v;
15. else 
16.         p = malloc(sizeof(elem_t));
17.         *p = v;
18. }
19.
20.     output(p->x, p->y);
21. }
```
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
01. typedef struct { 
02.   long x; 
03.   long y; 
04. } elem_t; 
05. 
06. int main() { 
07.   if (!rand(1000)) huft_build(...);
08. }
09. 
10. void huft_build(...) { 
11.   elem_t *p, v = {.x=0, .y=1};
12. 
13.   if (...) 
14.     p = &v;
15.   else { 
16.     p = malloc(sizeof(elem_t));
17.     *p = v;
18.   }
19. 
20.   output(p->x, p->y);
21. }
```
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
01. typedef struct {
02.     long x;
03.     long y;
04. } elem_t;
05.
06. int main() {
07.     if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...) {
11.     elem_t *p, v = {.x=0, .y=1};
12.
13.     if (...) 
14.         p = &v;
15.     else {
16.         p = malloc(sizeof(elem_t));
17.         *p = v;
18.     }
19.
20.     output(p->x, p->y);
21. }
```
State-of-the-art: Ghidra and IDA Pro

```c
01. typedef struct {
02.   long x;
03.   long y;
04. } elem_t;
05.
06. int main() {
07.   if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...) {
11.   elem_t *p, v = {.x=0, .y=1};
12.
13.   if (...) 
14.     p = &v;
15.   else {
16.     p = malloc(sizeof(elem_t));
17.     *p = v;
18.   }
19.
20.   output(p->x, p->y);
21. }
```
State-of-the-art: Ghidra and IDA Pro

Ghidra and IDA 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 IDA 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

```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 IDA 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.

```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

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;
```

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

IDA Pro

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.

```c
01. typedef struct {
02.   long x;
03.   long y;
04. } elem_t;
05.
06. int main() {
07.   if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...) {
11.   elem_t *p, v = {.x=0, .y=1};
12.
13.   if (...)  
14.     p = &v;
15.   else {
16.     p = malloc(sizeof(elem_t));
17.     *p = v;
18.   }
19.
20.   output(p->x, p->y);
21. }
```
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.

```c
01. typedef struct {
02.     long x;
03.     long y;
04. } elem_t;
05. 06. int main() {
07.     if (!rand(1000)) huft_build(...);
08. }
09. 10. void huft_build(...) {
11.     elem_t *p, v = {.x=0, .y=1};
12. 13.     if (...)  
14.         p = &v;
15.     else {
16.         p = malloc(sizeof(elem_t));
17.         *p = v;
18.     }
19. 20.     output(p->x, p->y);
21. }
```
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.

```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.

```c
01. typedef struct {
02.   long x;
03.   long y;
04. } elem_t;
05.
06. int main() {
07.   if (!rand(1000)) huft_build(...);
08. }
09. void huft_build(...) {
10.   elem_t *p, v = {.x=0, .y=1};
11. }
12.
13. if (...)  
14.   p = &v;  
15. else {  
16.     p = malloc(sizeof(elem_t));  
17.     *p = v;  
18.   }
19.
20. output(p->x, p->y);
21. }
```
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.

```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.

```c
#include <stdlib.h>

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);
}
```
OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary State

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.

```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);
}
```

```assembly
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.

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

define struct {  
    long x;
    long y;
} elem_t;

define 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.

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

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

```c
01. typedef struct {
02.     long x;
03.     long y;
04. } elem_t;
05.
06. int main() {
07.     if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...) {
11.     elem_t *p, v = {.x=0, .y=1};
12.     if (...) p = &v;
13.     else {
14.         p = malloc(sizeof(elem_t));
15.         *p = v;
16.     }
17.     output(p->x, p->y);
18. }
19.
20. output(p->x, p->y);
21. }
```
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.
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.

```c
01. typedef struct {
02.     long x;
03.     long y;
04. } elem_t;
05.
06. int main() {
07.     if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...) {
11.     elem_t *p, v = {.x=0, .y=1};
12. 
13.     if (...) 
14.         p = &v;
15.     else {
16.         p = malloc(sizeof(elem_t));
17.         *p = v;
18.     }
19. 
20.     output(p->x, p->y);
21. }
```
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.

```
01. typedef struct {
02.     long x;
03.     long y;
04. } elem_t;
05.
06. int main() {
07.     if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...) {
11.     elem_t *p, v = {.x=0, .y=1};
12.
13.     if (...)  
14.         p = &v;
15.     else {
16.         p = malloc(sizeof(elem_t));
17.         *p = v;
18.     }
19.
20.     output(p->x, p->y);
21. }
```
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.
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.

```
01. typedef struct {
02.   long x;
03.   long y;
04. } elem_t;
05. 06. int main() {
07.   if (!rand(1000)) huft_build(...);
08. }
09. 10. void huft_build(...) {
11.   elem_t *p, v = {.x=0, .y=1};
12. 13.   if (...) 
14.      p = &v;
15.   else {
16.      p = malloc(sizeof(elem_t));
17.      *p = v;
18.   }
19. 20.   output(p->x, p->y);
21. }
```
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.

```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);
}
```

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

```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);
}
```
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.

```c
01. typedef struct {
02.   long x;
03.   long y;
04. } elem_t;
05.
06. int main() {
07.   if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...) {
11.   elem_t *p, v = {.x=0, .y=1};
12.   
13.   if (...) 
14.     p = &v;
15.   else {
16.     p = malloc(sizeof(elem_t));
17.     *p = v;
18.   }
19.   
20.   output(p->x, p->y);
21. }
```
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.

```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);
}
```

*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.

```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);
}
```
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.

```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);
}
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.

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.

```c
01. typedef struct {
02.        long x;
03.        long y;
04. } elem_t;
05.
06. int main() {
07.     if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...) {
11.     elem_t *p, v = {.x=0, .y=1};
12.     if (...) 
13.         p = &v;
14.     else {
15.         p = malloc(sizeof(elem_t));
16.         *p = v;
17.     }
18. }
19. 
20.     output(p->x, p->y);
21. }
```
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)

```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);
}
```
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?

01. typedef struct
02.     long x;
03.     long y;
04. } elem_t;
05.
06. int main()
07. { if (!rand(1000)) huft_build(...);
08. }
09.
10. void huft_build(...)
11. { elem_t *p, v = {.x=0, .y=1};
12.     if (...) 
13.         p = &v;
14.     else {
15.         p = malloc(sizeof(elem_t));
16.         *p = v;
17.     }
18. }
19. output(p->x, p->y);
BDABDA [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. 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. 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. **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. 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.

Naïve Sampling:
- B1 – B3: $\frac{1}{2}$
- B1 – B2 – B4: $\frac{1}{4}$
- B1 – B2 – B5: $\frac{1}{4}$
**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:
- \( B1 \rightarrow B3: \frac{1}{2} \)
- \( B1 \rightarrow B2 \rightarrow B4: \frac{1}{4} \)
- \( B1 \rightarrow B2 \rightarrow B5: \frac{1}{4} \)
**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.

---

**Diagram:**

```
   B1
  / \  /  \
 B2  B3 B4  B5
```

**Naïve Sampling:**
- $B1 \rightarrow B3$: $\frac{1}{2}$
- $B1 \rightarrow B2 \rightarrow B4$: $\frac{1}{4}$
- $B1 \rightarrow B2 \rightarrow B5$: $\frac{1}{4}$

**BDA Sampling:**
- $B1 \rightarrow B3$: $\frac{1}{3}$
- $B1 \rightarrow B2 \rightarrow B4$: $\frac{1}{3}$
- $B1 \rightarrow B2 \rightarrow B5$: $\frac{1}{3}$
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)
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.

\begin{align*}
\text{AllocationHint} & \quad 0.9 \\
\text{PointToHint} & \quad 0.7 \\
\text{DataFlowHint} & \quad 0.6 \\
\text{UnifiedAccessHint} & \quad 0.6
\end{align*}

\begin{align*}
\rightarrow \quad \text{Probabilistic Inference}
\end{align*}
Our Techniques

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

![Probabilistic Inference Diagram]

```
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; 0.3
int64 v_y;
```
Our Techniques

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

<table>
<thead>
<tr>
<th>Hint Type</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocationHint</td>
<td>0.9</td>
</tr>
<tr>
<td>PointToHint</td>
<td>0.7</td>
</tr>
<tr>
<td>DataFlowHint</td>
<td>0.6</td>
</tr>
<tr>
<td>UnifiedAccessHint</td>
<td>0.6</td>
</tr>
</tbody>
</table>

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

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

elem_t *p_1;
union_t *p_2;
elem_t v;
int64 v_x;
int64 v_y;
```
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 \textit{probabilistic inference}.

\textit{Random Variables}: Random variables are introduced to describe variable properties, type properties, and structural properties.  
  \begin{itemize} 
    \item They are comprehensive and can be used to express the needed symbol information. 
  \end{itemize}
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

\[ \text{PrimitiveVar}(a, s) \]: 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

\[ \text{PrimitiveVar}(a, s) \]: a memory region at address \( a \) with size \( s \) is a primitive variable.

\[ \text{FieldOf}(a_1, s, a_2) \]: a memory region at address \( a_1 \) with size \( s \) is a field of a structure at address \( a_2 \).
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.

\[
\text{PrimitiveVar}(a, s) \quad \text{FieldOf}(a_1, s, a_2) \quad \text{HomoRegion}(a_1, a_2, s)
\]
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.

- \( \text{PrimitiveVar}(a, s) \)
- \( \text{FieldOf}(a_1, s, a_2) \)
- \( \text{HomoRegion}(a_1, a_2, s) \)

*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.

\[
\begin{align*}
\text{PrimitiveVar}(a, s) & \quad \text{FieldOf}(a_1, s, a_2) & \quad \text{HomoRegion}(a_1, a_2, s)
\end{align*}
\]

**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.

\[
\text{PrimitiveVar}(a, s) \quad \text{FieldOf}(a_1, s, a_2) \quad \text{HomoRegion}(a_1, a_2, s)
\]

**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.

- $\text{PrimitiveVar}(a, s)$
- $\text{FieldOf}(a_1, s, a_2)$
- $\text{HomoRegion}(a_1, a_2, s)$

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

\[ DataFlowHint(a1, a2, s) \xrightarrow{0.6} HomoRegion(a1, a2, s) \]
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.

- \text{PrimitiveVar}(a, s)
- \text{FieldOf}(a_1, s, a_2)
- \text{HomoRegion}(a_1, a_2, 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

\text{DataFlowHint}(a_1, a_2, s) \xrightarrow{0.6} \text{HomoRegion}(a_1, a_2, s)

\text{FieldOf}(a_1, s, a_2) \land \text{HomoRegion}(a_2, a_3, _) \xrightarrow{0.9} \text{FieldOf}(a_3+(a_1-a_2), s, a_3)
Osprey

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

**ELF**

1. **Stripped Binary**
2. **BDA**
3. **Basic Facts about Program Behaviors**
4. **Facts**
5. **Probabilistic Inference**
6. **Recovered Structures**

```c
typedef struct {
    long x;
    long y;
} elem_t;
```
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.
OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary

Evaluation: Argument Decompilation Results

```
int network_rxxx(server *srv)
{
    server *v1; // rbx
    int result; // rax
    size_t v3; // rbp
    server_socket *v4; // r12
    fdmode *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;
    }

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

```
struct_C264 *a1)
{
    struct_C264 *v1;
    __int32 result;
    unsigned __int64 v3;
    struct_CF4A *v4;
    struct_12A42 *v5;
    struct_12A0E *v6;

    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;
}
```

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

IDA Pro w/ debug information  Vanilla IDA Pro  IDA Pro w/ our tool
Evaluation: Argument Decompilation Results

```c
#include <stdio.h>

int network_rxxx(server *srv)
{
    // rbx
    int result = fdevent_sxxx(srv->ev);
    if (result != -1)
    {
        // rax
        size_t v3 = 0LL;
        if (!srv->sockets_disabled)
        {
            while (v1->srv_sockets.used > v3)
            {
                // rbp
                server *v1 = srv;
                // rdi
                result = fdevent_sxxx(srv->ev);
                if (result != -1)
                {
                    v3 = 0LL;
                    if (!srv->sockets_disabled)
                    {
                        while (v1->srv_sockets.used > v3)
                        {
                            // rax
                            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->fd = v5;
                                        fdevent_fxxx(v6, v5, 1);
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    return result;
}
```

```
__int32 __fastcall sub_D840(__int64 a1)
{
    // rbx
    struct_C264 *v1 = a1;
    // rax
    __int32 result = sub_12B7A((__int64)a1 + 24);
    if (result != -1)
    {
        // rbp
        v3 = 0LL;
        if (!*(__int64*)a1 + 100)
        {
            while (v1[2] > v3)
            {
                // rdi
                v4 = v1->ptr_prt_field[0][v3++];
                v5 = sub_21860(v1[3],
                    *(unsigned int*)(v4 + 112),
                    sub_18F30, v4);
                v6 = v1[3];
                *(__int64*)(v4 + 128) = v5;
                sub_219C0(v6, v5, 1);
            }
        }
    }
    result = 0;
    return result;
}
```

```
__int32 __fastcall sub_D840(struct_C264 *a1)
{
    // rbx
    __int32 result = sub_12B7A((__int64)a1 + 24);
    if (result != -1)
    {
        // rbp
        v3 = 0LL;
        if (!a1->dat_field_74)
        {
            while (v1->dat_field_10 > v3)
            {
            }
        }
    }
    result = 0;
    return result;
}
```
Evaluation: Argument Decompilation Results

```c
int network_rxxx(server *srv) {
    server *v1;  // rbx
    int result;  // rax
    size_t v3;   // rbp
    fmode *v5;  // rax
    sockfd *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->fd = v5;
        fdevent_fxxx(v6, v5, 1);
    }
    result = 0LL;
    return result;
}
```
Evaluation: Argument Decomposition Results

```c
int network_rxxx(server *srv) {
    server *v1; // rbx
    int result; // rax
    size_t v3; // rbp
    fnode *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;
    }
    return result;
}

__int32 __fastcall sub_D840(__int64 a1) {
    struct_C264 *v1;
    __int32 result;
    unsigned __int64 v3;
    struct_CF4A *v4;
    struct_12A42 *v5;
    struct_12A0E *v6;

    v1 = a1;
    result = sub_12B7A(*(_QWORD *)a1);
    if ( result != -1 ) {
        v3 = 0LL;
        if ( !a1->dat_field_74 ) {
            while ( v1->ptr_field_28 > v3 ) {
                v4 = v1->ptr_ptr_field_0[v3++];
                v5 = sub_21860(v1[3], *(unsigned int *)(v4 + 112), sub_18F30, v4);
                v6 = v1[3];
                *(unsigned int *)(v4 + 128) = v5;
                sub_219C0(v6, v5, 1);
            }
            result = 0;
        }
        return result;
    }
    return result;
}
```

IDA Pro w/ debug information

Vanilla IDA Pro

IDA Pro w/ our tool
**Evaluation: Argument Decompilation Results**

```c
int network_rxax(server *srv)
{
    server *v1; // rbx
    int result; // rax
    size_t v3; // rbp
    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->fd = v5;
                fdevent_fxxx(v6, v5, 1);
            }
        }
    }
    return result;
}
```

```c
__int32 __fastcall sub_D840(unsigned __int64 a1)
{
    struct_C264 *v1;
    unsigned __int64 v3;
    struct_CF4A *v4;
    struct_12A42 *v5;
    struct_12A0E *v6;

    v1 = a1;
    result = sub_12B7A(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[3],
                               *(unsigned long*v4 + 112),
                               sub_1B8F30, v4);
                v6 = v1[3];
                *(unsigned long*v4 + 120) = v5;
                sub_219C0(v6, v5, 1);
            }
        }
    }
    result = 0;
    return result;
}
```

**IDA Pro w/ debug information**

**Vanilla IDA Pro**

**IDA Pro w/ our tool**
```c
int network_rxxx(server *srv)
{
    server *v1; // rbx
    int result; // rax
    size_t v3; // rbp
    server_socket *v4; // r12
    fdmode *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->fd = v5;
                fdevent_fxxx(v6, v5, 1);
            }
            result = 0LL;
        }
        return result;
    }
    v1 = (_QWORD *)a1;
    result = sub_12B7A(_QWORD *)(a1 + 24);
    if ( _QWORD Result != -1 )
    {
        v3 = 0LL;
        if ( *(DWORD *)(a1 + 180) )
        {
            while ( v1[2] > v3 )
            {
                v4 = _QWORD *(_QWORD *)(v1[1] + 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;
    }
    return result;
}
```
Evaluation: Argument Decompilation Results

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

    v1 = srv;
    result = fdevent_sxxx(srv->ev);
    if ( result != -1 ) {
        v3 = 0LL;
        if ( !srv->sockets_disabled ) {
            while ( v1->srv_socket_used > v3 ) {
                v4 = v1->srv_socket_used;
                v5 = fdevent_sxxx(v4->ev, v4->fd, network_sxxx, v4);
                v6 = v4->ev;
                v4->fd = v5;
                fdevent_fxxx(v6, v5, 1);
            }
            result = 0LL;
        }
        return result;
    }
    return result;
}

__int32 __fastcall sub_D840(__int64 a1) {
    struct_C264 *v1;
    __int32 result;
    unsigned __int64 v3;
    struct_CF4A *v4;
    struct_12A42 *v5;
    struct_12A0E *v6;

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

**Variable recovery and type inference for stripped binary:**

**Probabilistic Program Analysis:**
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!