Automatic Reverse Engineering of Data Structures from Binary Execution

Zhiqiang Lin   Xiangyu Zhang   Dongyan Xu
Department of Computer Science and CERIAS
Purdue University, West Lafayette, IN
{zlin,xyzhang,dxu}@cs.purdue.edu

Abstract

With only the binary executable of a program, it is useful to discover the program's data structures and infer their syntactic and semantic definitions. Such knowledge is highly valuable in a variety of security and forensic applications. Although there exist efforts in program data structure inference, the existing solutions are not suitable for our targeted application scenarios. In this paper, we propose a reverse engineering technique to automatically reveal program data structures from binaries. Our technique, called REWARDS, is based on dynamic analysis. More specifically, each memory location accessed by the program is tagged with a timestamped type attribute*. Following the program's runtime data flow, this attribute is propagated to other memory locations and registers that share the same type. During the propagation, a variable's type gets resolved if it is involved in a type-revealing execution point or "type sink". More importantly, besides the forward type propagation, REWARDS involves a* backward type resolution *procedure where the types of some previously accessed variables get recursively resolved starting from a type sink. This procedure is constrained by the timestamps of relevant memory locations to disambiguate variables re-using the same memory location. In addition, REWARDS is able to reconstruct in-memory data structure layout based on the type information derived. We demonstrate that REWARDS provides unique benefits to two applications:* memory image forensics *and* binary fuzzing for vulnerability discovery.

1 Introduction

A desirable capability in many security and forensics applications is automatic reverse engineering of data structures given only the binary. Such capability is expected to identify a program's data structures and reveal their syntax (e.g., size, structure, offset, and layout) and semantics (e.g., "this integer variable represents a process ID"). Such knowledge about program data structures is highly valuable. For example, in memory-based forensics, this knowledge will help locate specific information of interest (e.g., IP addresses) in a memory core dump without symbolic information; In binary vulnerability discovery, this knowledge will help construct a meaningful view of in-memory data structure layout and identify those semantically associated with external input for guided fuzz testing.

Despite the usefulness of automatic data structure reverse engineering, solutions that suit our targeted application scenarios fall short. First, a large body of work on type inference [29, 3, 13, 33, 32, 24] requires program source code. Second, in the binary-only scenario, variables are mapped to low-level entities such as registers and memory locations with no syntactic information, which makes static analysis difficult. In particular, alias analysis is hard at binary level while it is essential to type inference – especially semantics inference – because precise data flow cannot be decided without accurate alias information. Variable discovery [5] is a static, binary level technique that recovers syntactic characteristics of variables, such as a variable's offset in its activation record, size, and hierarchical structure. This technique relies on alias analysis and abstract interpretation at binary level and is hence heavy-weight. Moreover, due to the conservative nature of binary alias analysis, the technique does not infer variable semantics. More recently, Laika [16] aims at dynamically discovering the syntax of observable data structures through unsupervised machine learning on program execution. The accuracy of this technique, however, may fall below the expectation of our applications. It does not consider data structure semantics either. The limitations of these efforts motivate us to develop new techniques for our targeted application scenarios.

In this paper, we propose a reverse engineering scheme to automatically reveal program data structures from binaries. Our technique, called REWARDS[1], is based on dynamic analysis. Given a binary executable, REWARDS executes the binary, monitors the execution, aggregates and analyzes runtime information, and finally recovers both the syntax and semantics of data structures observed in the execution. More specifically, each memory location

[1]REWARDS is the acronym for Reverse Engineering Work for Automatic Revelation of Data Structures.

accessed by the program is tagged with a *timestamped type attribute*. Following the program's runtime data flow, this attribute is propagated to other memory addresses and registers that share the same type in a forward fashion, i.e., the execution direction. During the propagation, a variable's type gets resolved if it is involved in a type-revealing execution point or "type sink" (e.g., a system call, a standard library call, or a type-revealing instruction). Besides leveraging the forward type propagation technique, to expand the coverage of program data structures, REWARDS involves the following key techniques:

- An on-line *backward type resolution* procedure where the types of some previously accessed variables get *recursively* resolved starting from a type sink. Since many variables are dynamically created and de-allocated at runtime, and the same memory location may be re-used by different variables, it is complicated to track and resolve variable types based on memory locations alone. Hence, we constraint the resolution process by the timestamps of relevant memory locations such that variables sharing the same memory location in different execution phases can be disambiguated.
- An off-line resolution procedure that complements the on-line procedure. Some variables cannot be resolved during their lifetime by our on-line algorithm. However, they may later get resolved when other variables having the same type are resolved. Hence, we propose an off-line backward resolution procedure to resolve the types of some "dead" variables.
- A method for typed variable abstraction that maps multiple typed variable instances to the same static abstraction. For example, all N nodes in a linked list actually share the same type, instead of having N distinct types.
- A method that reconstructs the structural and semantic view of in-memory data, driven by the derived type definitions. Once a program's data structures are identified, it is still not clear exactly how the data structures would be laid out in memory – this is a useful piece of knowledge in many application scenarios such as memory forensics. Our method creates an "organization chart" that illustrates the hierarchical layout of those data structures.

We have developed a prototype of REWARDS and used it to analyze a number of binaries. Our evaluation results show that REWARDS is able to correctly reveal the types of a high percentage of variables observed during a program's execution. Furthermore, we demonstrate the unique benefits of REWARDS to a variety of application scenarios: In *memory image forensics*, REWARDS helps recovering semantic information from the memory dump of a binary program. In *binary fuzzing for vulnerability discovery*, REWARDS helps identifying vulnerability "suspects" in a binary for guided fuzzing and confirmation.

2 REWARDS Overview

REWARDS infers both syntax and semantics of data structures from binary execution. More precisely, we aim at reverse engineering the following information:

- **Data types.** We first aim to infer the primitive data types of variables, such as `char`, `short`, `float`, and `int`. In a binary, the variables are located in various segments of the virtual address space, such as `.stack`, `.heap`, `.data`, `.bss`, `.got`, `.rodata`, `.ctors`, and `.dtors` sections. (Although we focus on ELF binary on Linux platform, REWARDS can be easily ported to handle PE binary on Windows.) Hence, our goal is essentially to annotate memory locations in these data sections with types and sizes, following program execution. For our targeted applications, REWARDS also infers composite types such as socket address structures and FILE structures.
- **Semantics.** Moreover, we aim to infer the semantics (meaning) of program variables, which is critical to applications such as computer forensics. For example, in a memory dump, we want to decide if a 4-byte integer denotes an IP address.
- **Abstract representation.** Although we type memory locations, it is undesirable to simply present typed memory locations to the user. During program execution, a memory location may be used by multiple variables at different times; and a variable may have multiple instances. Hence we derive an abstract representation for a variable by aggregating the type information at multiple memory locations instantiated based on the same variable. For example, we use the offset of a local variable in its activation record as its abstract representation. Type information collected in all activation records of the same function is aggregated to derive the type of the variable.

Given only the binary, what can be observed at runtime from each instruction includes (1) the addresses accessed and the width of the accesses, (2) the semantics of the instruction, and (3) the execution context such as the program counter and the call stack. In some cases, data types can be partially inferred from instructions. For example, a floating point instruction (e.g., FADD) implies that the accessed locations must have floating point numbers. We also observe that the parameters and return values of standard library calls and system calls often have their syntax and semantics

```
1 struct {
2     unsigned int pid;
3     char data[16];
4 }test;
5
6 void foo(){
7     char *p="hello world";
8     test.pid=my_getpid();
9     strcpy(test.data,p);
10 }
```

```
1 extern foo
2 section .text
3     global _start
4
5 _start:
6     call foo
7     mov eax,1
8     mov ebx,0
9     int 80h
```

(a) Source code of function `foo` and the `_start` assembly code

```
[Nr] Name      Type       Addr     Off    Size
...
[ 1] .text     PROGBITS   080480a0 0000a0 000078
[ 2] .rodata   PROGBITS   08048118 000118 00000c
[ 3] .bss      NOBITS     08049124 000124 000014
...
```

(c) Section map of the `example` binary

```
rodata_0x08048118{
+00:    char[12]
}
bss_0x08049124{
+00:    pid_t,
+04:    char[12],
+16:    unused[4]
}
fun_0x080480b4{
-28:    unused[20],
-08:    char *,
-04:    stack_frame_t,
+00:    ret_addr_t
}
```

```
fun_0x08048110{
+00:   ret_addr_t
}

fun_0x080480e0{
-08:   unused[4],
-04:   stack_frame_t,
+00:   ret_addr_t,
+04:   char*,
+08:   char*
}
```

(d) Output of REWARDS

```
 1  80480a0:   e8 0f 00 00 00          call   0x80480b4
 2  80480a5:   b8 01 00 00 00          mov    $0x1,%eax
 3  80480aa:   bb 00 00 00 00          mov    $0x0,%ebx
 4  80480af:   cd 80                   int    $0x80
 5  ...
 6  80480b4:   55                      push   %ebp
 7  80480b5:   89 e5                   mov    %esp,%ebp
 8  80480b7:   83 ec 18                sub    $0x18,%esp
 9  80480ba:   c7 45 fc 18 81 04 08    movl   $0x8048118,0xfffffffc(%ebp)
10  80480c1:   e8 4a 00 00 00          call   0x8048110
11  80480c6:   a3 24 91 04 08          mov    %eax,0x8049124
12  80480cb:   8b 45 fc                mov    0xfffffffc(%ebp),%eax
13  80480ce:   89 44 24 04             mov    %eax,0x4(%esp)
14  80480d2:   c7 04 24 28 91 04 08    movl   $0x8049128,(%esp)
15  80480d9:   e8 02 00 00 00          call   0x80480e0
16  80480de:   c9                      leave
17  80480df:   c3                      ret
18  80480e0:   55                      push   %ebp
19  80480e1:   89 e5                   mov    %esp,%ebp
20  80480e3:   53                      push   %ebx
21  80480e4:   8b 5d 08                mov    0x8(%ebp),%ebx
22  80480e7:   8b 55 0c                mov    0xc(%ebp),%edx
23  80480ea:   89 d8                   mov    %ebx,%eax
24  80480ec:   29 d0                   sub    %edx,%eax
25  80480ee:   8d 48 ff                lea    0xffffffff(%eax),%ecx
26  80480f1:   0f b6 02                movzbl (%edx),%eax
27  80480f4:   83 c2 01                add    $0x1,%edx
28  80480f7:   84 c0                   test   %al,%al
29  80480f9:   88 04 0a                mov    %al,(%edx,%ecx,1)
30  80480fc:   75 f3                   jne    0x80480f1
31  80480fe:   89 d8                   mov    %ebx,%eax
32  8048100:   5b                      pop    %ebx
33  8048101:   5d                      pop    %ebp
34  8048102:   c3                      ret
35  ...
36  8048110:   b8 14 00 00 00          mov    $0x14,%eax
37  8048115:   cd 80                   int    $0x80
38  8048117:   c3                      ret
```

(b) Disassembly code of the `example` binary

Figure 1. An example showing how REWARDS works

well defined and publicly known. Hence we define the type revealing instructions, system calls, and library calls as *type sinks*. Furthermore, the execution of an instruction creates a dependency between the variables involved. For instance, if a variable with a resolved type (from a type sink) is copied to another variable, the destination variable should have a compatible type. As such, we model our problem as a type information flow problem.

To illustrate how REWARDS works, we use a simple program compiled from the source code shown in Figure 1(a). According to the code snippet, the program has a global variable `test` (line 1-4) which consists of an `int` and a `char` array. It contains a function `foo` (line 6-10) that calls `my_getpid` and `strcpy` to initialize the global variable. The full disassembled code of the example is shown in Figure 1(b) (a dotted line indicates a "`NOP`" instruction). The address mapping of code and data is shown in Figure 1(c).

When `foo` is called during execution, it first saves `ebp` and then allocates `0x18` bytes of memory for the local variables (line 8 in Figure 1(b)), and then initializes one local variable (at address `0xfffffffc(%ebp)=ebp-4`) with an immediate value `0x8048118` (line 9). Since `0x8048118` is in the address range of the `.rodata` section (it is actually the starting address of string "`hello world`"), `ebp-4` can be typed as a pointer, based on the heuristics that instruction executions using similar immediate values within a code or data section are considered type sinks. Note that the type of the pointer is unknown yet. At line 10, `foo` calls `0x8048110`. Inside the body of the function invocation (lines 36-38), our algorithm detects a `getpid` system call (a type sink) with `eax` being `0x14` at line 36. The return value of the function call is resolved as `pid_t` type, i.e., register `eax` at line 11 is typed `pid_t`. When `eax` is copied to address `0x8049124` (a global variable in `.bss` section as shown in Figure 1(c)), the algorithm further resolves `0x8049124` as `pid_t`. Before the function call `0x80480e0` at line 15 (`strcpy`), the parameters are initialized in lines 12-14. As `ebp-4` has been typed as a pointer at line 9, the data flow in lines 12 and 13 dictates that location `esp+4` at line 13 is a pointer as well. At line 14, as `0x8049128` is in the global variable section and of a known type, location `esp` has an unknown pointer type. At line 15, upon the call to `strcpy` (a type sink), both `esp` and `esp+4` are resolved to `char*`. Through a *backward* transitive resolution, `0x8049128` is resolved as `char`, `ebp-4` as `char*`, and `0x8048118` as `char`. Also at line 26, inside the function body of `strcpy`, the instruction "`movzbl (%edx),%eax`" can be used as another type sink as it moves between `char` variables.

When the program finishes, we resolve all data types (including function arguments, and those implicit variables such as return address and stack frame pointer) as shown in Figure 1(d). The derived types for variables in `.rodata`, `.bss` and functions are presented in the figure. Each function is denoted by its entry address. `fun_0x080480b4`, `fun_0x08048110`, and `fun_0x080480e0` denote `foo()`, `my_getpid()`, and `strcpy()`, respectively. The number before each derived type denotes the offset. Variables are listed in increasing order of their addresses. Type `stack_frame_t` indicates a frame pointer stored at that location. Type `ret_addr_t` means that the location holds a return address. Such semantic information is useful in applications such as vulnerability fuzz. Locations that are not accessed during execution are annotated with the `unused` type. In `fun_0x080480e0`, the two `char*` below the `ret_addr_t` represent the two actual arguments of `strcpy()`. Although it seems that our example can be statically resolved due to its simplicity, it is very difficult in practice to analyze data flows between instructions (especially those involving heap locations) due to the difficulty of binary points-to analysis.

3 REWARDS Design

In this section, we describe the design of REWARDS. We first identify the type sinks used in REWARDS and then present the on-line type propagation and resolution algorithm, which will be enhanced by an off-line procedure that recovers more variable types not reported by the on-line algorithm. Finally, we present a method to construct a typed hierarchical view of memory layout.

3.1 Type Sinks

A type sink is an execution point of a program where the types (including semantics) of one or more variables can be directly resolved. In REWARDS, we identify three categories of type sinks: (1) system calls, (2) standard library calls, and (3) type-revealing instructions.

System calls. Most programs request OS services via system calls. Since system call conventions and semantics are well-defined, the types of arguments of a system call are known from the system call's specification. By monitoring system call invocations and returns, REWARDS can determine the types of parameters and return value of each system call at runtime. For example, in Linux, based on the system call number in register `eax`, REWARDS will be able to type the parameter-passing registers (i.e., `ebx`, `ecx`, `edx`, `esi`, `edi`, and `ebp`, if they are used for passing the parameters). From this type sink, REWARDS will further type those variables that are determined to have the same type as the parameter passing registers. Similarly, when a system call returns, REWARDS will type register `eax` and, from there, those having the same type as `eax`. In our type propagation and resolution algorithm (Section 3.2), a type sink will lead to the recursive type resolution of relevant variables accessed before and after the type sink.

Standard library calls. With well-defined API, standard library calls are another category of type sink. For example, the two arguments of `strcpy` must both be of the `char*` type. By intercepting library function calls and returns, REWARDS will type the registers and memory variables involved. Standard library calls tend to provide richer type information than system calls – for example, Linux-2.6.15 has 289 system calls whereas `libc.so.6` contains 2016 functions (note some library calls wrap system calls).

Type-revealing instructions. A number of machine instructions that require operands of specific types can serve as type sinks. Examples in x86 are as follows: (1) *String instructions* perform byte-string operations such as moving/storing (`MOVS/B/D/W`, `STOS/B/D/W`), loading (`LOADS/B/D/W`), comparison (`CMPS/B/D/W`), and scanning (`SCAS/B/D/W`). Note that `MOVZBL` is also used in string movement. (2) *Floating-point instructions* operate on floating-point, integer, and binary coded decimal operands (e.g. `FADD`, `FABS`, and `FST`). (3) *Pointer-related instructions* reveal pointers. For a `MOV` instruction with an indirect memory access operand (e.g., `MOV (%edx), %ebx` or `MOV [mem], %eax`), the value held in the source operand must be a pointer. Meanwhile, if the target address is within the range of data sections such as `.stack`, `.heap`, `.data`, `.bss` or `.rodata`, the pointer must be a data pointer; If it is in the range of `.text` (including library code), the pointer must be a function pointer. Note that the concrete type of such a pointer will be resolved through other constraints.

3.2 Online Type Propagation and Resolution Algorithm

Given a binary program, our algorithm reveals variable types, including both syntactic types (e.g., `int` and `char`) and semantics (e.g., `return address`), by propagating and resolving type information along the data flow during program execution. Each type sink encountered leads to both direct and transitive type resolution of variables. More specifically, at the binary level, variables exist in either memory locations or registers without their symbolic names. Hence, the goal of our algorithm is to type these memory addresses and registers. We attach three *shadow variables* – as the type attribute – to each memory address at byte granularity (registers are treated similarly): (1) *Constraint set* is a set of other memory addresses that should have the same type as this address; (2) *Type set* stores the set of resolved types of the address[2], including both syntactic and semantic types; (3) *Timestamp* records the birth time of the variable currently in this address. For example, the timestamp of a stack variable is the time when its residence method is invoked and the stack frame is allocated. Timestamps are needed because the same memory address may be reused by multiple variables (e.g., the same stack memory being reused by stack frames of different method invocations). More precisely, a variable instance should be uniquely identified by a tuple <address, timestamp>. These shadow variables are updated during program execution, depending on the semantics of executed instructions.

Algorithm 1 *On-line Type Propagation and Resolution*

```
 1: /* S_v: constraint set for memory cell (or register) v; T_v: type set of v; ts_v:
    (birth) time stamp of v; MOV(v,w): moving v to w; BIN_OP(v,w,d): a binary
    operation that computes d from v and w; Get_Sink_Type(v,i): retrieving the
    type of argument/operand v from the specification of sink i; ALLOC(v,n):
    allocating a memory region starting from v with size n – the memory region
    may be a stack frame or a heap struct; FREE(v,n): freeing a memory region –
    this may be caused by eliminating a stack frame or de-allocating a heap struct*/
 2: Instrument(i){
 3:     case i is a Type_Sink:
 4:         for each operand v
 5:             T ← Get_Sink_Type(v, i)
 6:             Backward_Resolve (v, T)
 7:     case i has indirect memory access operand o
 8:             T_o ← T_o ∪ {pointer_type_t}
 9:     case i is MOV(v, w):
10:         if w is a register
11:             S_w ← S_v
12:             T_w ← T_v
13:         else
14:             Unify(v, w)
15:     case i is BIN_OP(v, w, d):
16:         if pointer_type_t ∈ T_v
17:             Unify(d, v)
18:             Backward_Resolve (w, {int, pointer_index_t})
19:         else
20:             Unify3(d, v, w)
21:     case i is ALLOC(v, n):
22:         for t=0 to n − 1
23:             ts_{v+t} ← current timestamp
24:             S_{v+t} ← φ
25:             T_{v+t} ← φ
26:     case i is FREE(v, n):
27:         for t=0 to n − 1
28:             a ← v+t
29:             if (T_a) log (a, ts_a, T_a)
30:             log (a, ts_a, S_a)
31: }
32: Backward_Resolve(v,T){
33:     for < w, t > ∈ S_v
34:         if (T ⊄ T_w and t ≡ ts_w) Backward_Resolve(w,T-T_w)
35:     T_v ← T_v ∪ T
36: }
37: Unify(v,w){
38:     Backward_Resolve(v, T_w-T_v)
39:     Backward_Resolve(w, T_v-T_w)
40:     S_v ← S_v ∪ {< w, ts_w >}
41:     S_w ← S_w ∪ {< v, ts_v >}
42: }
```

The algorithm is shown in Algorithm 1. The algorithm takes appropriate actions to resolve types on the fly according to the instruction being executed. For a memory address or a register v, its constraint set is denoted as S_v, which is a set of <address, timestamp> tuples each representing a variable instance that should have the same type as v; its type set T_v represents the resolved types for v; and the birth time of the current variable instance is denoted as ts_v.

1. If the current execution point i is a type sink (line 3). The arguments/operands/return value of the sink will be directly typed according to the sink's definition (**Get_Sink_Type()** on line 5)[3]. Type resolution is then triggered by calling the recursive method **Backward_Resolve()**. The method recursively types all variables that should have the same type (lines 32-36): It tests if each variable w in the constraint set of v has been resolved as type T of v. If not, it recursively calls itself to type all the variables that should have the same type as w. Note that at line 34, it checks if the current birth timestamp of w is equal to the one stored in the constraint set to ensure the memory has not been re-used by a different variable. If w is re-used ($t \neq ts_w$), the algorithm does not resolve the current w. Instead, the resolution is done by a different off-line procedure (Section 3.3). Since variable types are resolved according to constraints derived from data flows in the past, we call this step backward type resolution.

2. If i contains an indirect memory access operand o (line 7), either through registers (e.g., using `(%eax)` to access the address designated by `eax`) or memory (e.g., using `[mem]` to indirectly access the memory pointed to by `mem`), then the corresponding operand will have a pointer *type tag* (`pointer_type_t`) as a new element in T_o.

3. If i is a move instruction (line 9), there are two cases to consider. In particular, if the destination operand w is a register, then we just move the properties (i.e., the S_v and T_v) of the source operand to the destination (i.e., the register); otherwise we need to unify the types of the source and destination operands because the destination is now a memory location that may have already contained some resolved types. The intuition is that the source operand v should have the same type as the destination operand w if the destination is a memory address. Hence, the algorithm calls method **Unify()** to unify the types of the two. In **Unify()** (lines 37-42), the algorithm first unions the two type sets by performing backward resolution at lines 38 and 39. Intuitively, the call at line 38 means that if there are any new types in T_w that are not in T_v (i.e. T_w-T_v), those new types need to be propagated to v and transitively to all variables that share the same type as v, mandated by v's constraint set. Such unification is not performed if the w is a register to avoid over-aggregation.

[2]We need a set to store the resolved types because one variable may have multiple compatible types.

[3]The sink's definition also reveals the semantics of some arguments/operands, e.g., a PID.

4. If i is a binary operation, the algorithm first tests if an operand has been identified as a pointer. If so, it must be a pointer arithmetic operation, the destination must have the same type as the pointer operand and the other operand must be a pointer index – denoted by a semantic type `pointer_index_t` (line 18). The semantic type is useful in vulnerability fuzz to overflow buffers. If i is not related to pointers, the three operands shall have the same type. The method **Unify3()** unifies three variables. It is very similar to **Unify()** and hence not shown. Note that in cases where the binary operation implicitly casts the type of some operand (e.g., an addition of a float and an integer), the unification induces over-approximation (e.g., associating the float point type with the integer variable). In practice, we consider such cases reasonable and allow multiple types for one variable as long as they are compatible.

5. If i allocates a memory region (line 21) – either a stack frame or a heap struct, the algorithm updates the birth time stamps of all the bytes in the region, and resets the memory constraint set (S_v) and type set (T_v) to empty. By doing so, we prevent the type information of the old variable instance from interfering with that of the new instance at the same address.

6. If i frees a memory region (line 26), the algorithm traverses each byte in the region and prints out the type information. In particular, if the type set is not empty, it is emitted. Otherwise, the constraint set is emitted. Later, the emitted constraints will be used in the off-line procedure (Section 3.3) to resolve more variables.

Example. Table 1 presents an example of executing our algorithm. The first column shows the instruction trace with the numbers denoting timestamps. The other columns show the type sets and the constraint sets after each instruction execution for three sample variables, namely the global variable $g1$ and two local variables $l1$ and $l2$. For brevity, we abstract the calling sequence of `strcpy` to a `strcpy` instruction. After the execution enters method M at timestamp 10, the local variables are allocated and hence both $l1$ and $l2$ have the birth time of 10. The global variable $g1$ has the birth time of 0. After the first `mov` instruction, the type sets of $g1$ and $l1$ are unified. Since neither was typed, the unified type set remains empty. Moreover, $l1$, together with its birth time 10, is added to the constraint set of $g1$ and vice versa, denoting they should have the same type. Similar actions are taken after the second `mov` instruction. Here, the constraint set of $l1$ has both $g1$ and $l2$. The `strcpy` invocation is a type sink and $g1$ must be of type `char*`, the algorithm performs the backward resolution by calling **Backward_Resolve()**. In particular, the variable in S_{g1}, i.e. $l1$, is typed to `char*`. Note that the timestamp 10 matches ts_{l1}, indicating the same variable is still alive. Transitively, the variables in S_{l1}, i.e. $g1$ and $l2$, are resolved to the same type. Note that if the backward resolution was not conducted, we would not be able to resolve the type of $l2$ because when the move from $l1$ to $l2$ (timestamp 12) occurred, $l1$ was not typed and hence $l2$ was not typed.

3.3 Off-line Type Resolution

Most variables accessed during the binary's execution can be resolved by our online algorithm. However, there are still some cases in which, when a memory variable gets freed (and its information gets emitted to the log file), its type is still unresolved. We realize that there may be enough information from later phases of the execution to resolve those variables. We propose an off-line procedure to be performed *after* the program execution terminates. It is essentially an off-line version of the **Backward_Resolve()** method in Algorithm 1. The difference is that it has to traverse the log file to perform the recursive resolution.

Consider the example in Table 2. It shares the same execution as the example in Table 1 before timestamp 13. At time instance 13, the execution returns from M, deallocating the local variables $l1$ and $l2$. According to the online algorithm, their constraint sets are emitted to a log file since neither is typed at that point. Later at timestamp 99, another method N is called. Assume it reuses $l1$ and $l2$, namely, N allocates its local variables at the locations of $l1$ and $l2$. The birth time of $l1$ and $l2$ becomes 99. Their type sets and constraint sets are reset. When the sink is encountered at 100, $l1$ and $l2$ are not typed as their current birth timestamp is 99, not 10 as in S_{g1}, indicating they are re-used by other variables. Fortunately, the variable represented by $< l1, 10 >$ can be found in the log and hence resolved. Transitively, $< l2, 10 >$ can be resolved as well.

3.4 Typed Variable Abstraction

Our algorithm is able to annotate memory locations with syntax and semantics. However, multiple variables may occupy the same memory location at different times and a static variable may have multiple instances at runtime[4]. Hence it is important to organize the inferred type information according to abstract, location-independent variables other than specific memory locations. In particular, primitive global variables are represented by their offsets to the base of the global sections (e.g., `.data` and `.bss` sections). Stack variables are abstracted by the offsets from their residence activation record, which is represented by the function name (as shown in Figure 1).

For heap variables, we use the execution context, i.e., the PC (instruction address) of the allocation point of a heap

[4]A local variable has the same life time of a method invocation and a method can be invoked multiple times, giving rise to multiple instances.

instruction	T_{g1}	S_{g1}	ts_{g1}	T_{l1}	S_{l1}	ts_{l1}	T_{l2}	S_{l2}	ts_{l2}
10. enter M	ϕ	ϕ	0	ϕ	ϕ	10	ϕ	ϕ	10
11. mov $g1, l1$	ϕ	$\{< l1, 10 >\}$	0	ϕ	$\{< g1, 0 >\}$	10	ϕ	ϕ	10
12. mov $l1, l2$	ϕ	$\{< l1, 10 >\}$	0	ϕ	$\{< g1, 0 >, < l2, 10 >\}$	10	ϕ	$\{< l1, 10 >\}$	10
...	...	...	...	...	...	...	...	...	...
100. strcpy($g1$,...)	{char*}	$\{< l1, 10 >\}$	0	{char*}	$\{< g1, 0 >, < l2, 10 >\}$	10	{char*}	$\{< l1, 10 >\}$	10

Table 1. Example of running the online algorithm. Variable $g1$ is a global, $l1$ and $l2$ are locals.

instruction	T_{g1}	S_{g1}	ts_{g1}	T_{l1}	S_{l1}	ts_{l1}	T_{l2}	S_{l2}	ts_{l2}
...	...	...	...	...	...	...	...	...	...
12. mov $l1, l2$	ϕ	$\{< l1, 10 >\}$	0	ϕ	$\{< g1, 0 >, < l2, 10 >\}$	10	ϕ	$\{< l1, 10 >\}$	10
13. Exit M	ϕ	$\{< l1, 10 >\}$	0	ϕ	$\{< g1, 0 >, < l2, 10 >\}$	10	ϕ	$\{< l1, 10 >\}$	10
...	...	...	...	...	...	...	...	...	...
99. Enter N	ϕ	$\{< l1, 10 >\}$	0	ϕ	ϕ	99	ϕ	ϕ	99
100. strcpy($g1$,...)	{char*}	$\{< l1, 10 >\}$	0	ϕ	ϕ	99	ϕ	ϕ	99

Table 2. Example of running the off-line type resolution procedure. The execution before timestamp 12 is the same as Table 1. Method N reuses $l1$ and $l2$

structure plus the call stack at that point, as the abstraction of the structure. The intuition is that the heap structure instances allocated from the same PC in the same call stack should have the same type. Fields of the structure are represented by the allocation site and field offsets. As an allocated heap region may be an array of a data structure, we use the recursion detection heuristics in [9] to detect the array size. Specifically, the array size is approximated by the maximum number of accesses *by the same PC* to unique memory locations in the allocated region. The intuition is that array elements are often accessed through a loop in the source code and the same instruction inside the loop body often accesses the same field across all array elements. Finally, if heap structures allocated from different sites have the same field types, we will heuristically cluster these heap structures into one abstraction.

3.5 Constructing Hierarchical View of In-Memory Data Structure Layout

An important feature of REWARDS is to construct a hierarchical view of a memory snapshot, in which the primitive syntax of individual memory locations, as well as their semantics and the integrated hierarchical structure are visually represented. This is highly desirable in applications like memory forensics as interesting queries, e.g., "`find all IP addresses`", can be easily answered by traversing the view (examples in Section 5.1). So far, REWARDS is able to reverse engineer the syntax and semantics of data structures, represented by their abstractions. Next, we present how we leverage such information to construct a hierarchical view.

Our method works as follows. It first types the top level global variables. In particular, a root node is created to represent a global section. Individual global variables are represented as children of the root. Edges are annotated with offset, size, primitive type, and semantics of the corresponding children. If a variable is a pointer, the algorithm further recursively constructs the sub-view of the data structure being pointed to, leveraging the derived type of the pointer. For instance, assume a global pointer p is of type `T*`, our method creates a node representing the region pointed to by p. The region is typed based on the reverse engineered definition of `T`. The recursive process terminates when none of the fields of a data structure is a pointer. Stack is similarly handled: A root node is created to represent each activation record. Local variables of the record are denoted as children nodes. Recursive construction is performed until all memory locations through pointers are traversed. Note that all live heap structures can be reached (transitively) through a global pointer or a stack pointer. Hence, the above two steps essentially also construct the structural views of live heap data.

Our method can also type some of the unreachable memory regions, which represent "dead" data structures, e.g., activation records of previous method invocations whose space has been freed but not reused. Such dead data is as important as live data as they disclose what had happened in the past. In particular, our method scans the stack beyond the current activation record to identify any pointers to the code section, which often denote return addresses of method invocations. With a return address, the function invocation can be identified and we can follow the aforementioned steps to type the activation record.

4 Implementation and Evaluation

We have implemented REWARDS on PIN-2.6 [27], with 12.1K lines (LOC) of C code and 1.2K LOC of Python code. In the following, we present several key implementation details. REWARDS is able to reveal variable semantics. In our implementation, variable semantics are represented as special *semantic tags* complementary to regular type tags such as `int` and `char`. Both semantic tags and regular tags are stored in the variable's type set. Tags are enumerated to save space. The vast diversity of program semantics makes it infeasible to consider them all. Since we are mainly interested in forensics and security applications, we focus on the following semantic tags: (1) file system related (e.g., FILE pointer, file descriptor, file name, file status); (2) network communication related (e.g., socket descriptor, IP address, port, receiving and sending buffer, host info, msghdr); and (3) operating systems related (e.g., PID, TID, UID, system time, system name, and device info).

Meanwhile, we introduce some of our own semantic tags, such as `ret_addr_t` indicating that a memory location is holding a return address, `stack_frame_t` indicating that a memory location is holding a stack frame pointer, `format_string_t` indicating that a string is used in format string argument, and `malloc_arg_t` indicating an argument of `malloc` function (similarly, `calloc_arg_t` for `calloc` function, etc.). Note that these tags reflect the properties of variables at those specific locations and hence do not particitate in the type information propagation. They can bring important benefits to our targeted applications (Section 5).

REWARDS needs to know the program's address space mapping, which will be used to locate the addresses of global variables and detect pointer types. In particular, REWARDS checks the target address range when determining if a pointer is a function pointer or a data pointer. Thus, when a binary starts executing with REWARDS, we first extract the coarse-grained address mapping from the `/proc/pid/maps` file, which defines the ranges of code and data sections including those from libraries, and the ranges of stack and heap (at that time). Then for each detailed address mapping such as `.data`, `.bss` and `.rodata` for all loaded files (including libraries), we extract the mapping using the API provided by PIN when the corresponding image file is loaded.

We have performed two sets of experiments to evaluate REWARDS: one is to evaluate its correctness, and the other is to evaluate its time and space efficiency. All the experiments were conducted on a machine with two 2.13Ghz Pentium processors and 2GB RAM running Linux kernel 2.6.15.

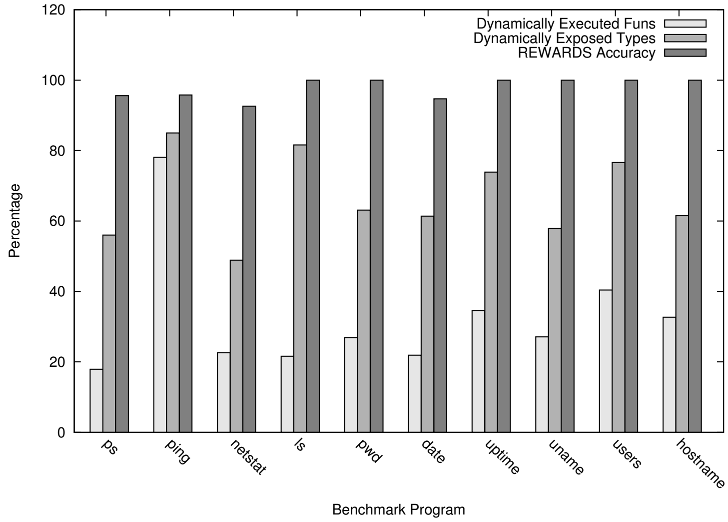
We select 10 widely used utility programs from the following packages: procps-3.2.6 (with 19.1K LOC and containing command `ps`), iputils-20020927 (with 10.8K LOC and containing command `ping`), net-tools-1.60 (with 16.8K LOC and containing `netstat`), and coreutils-5.93 (with 117.5K LOC and containing the remaining test commands such as `ls`, `pwd`, and `date`). The reason for selecting these programs is that they contain many data structures related to the operating system and network communications. We run these utilities without command line option except `ping`, which is run with a localhost and a packet count 4 option.

4.1 Evaluation of Accuracy

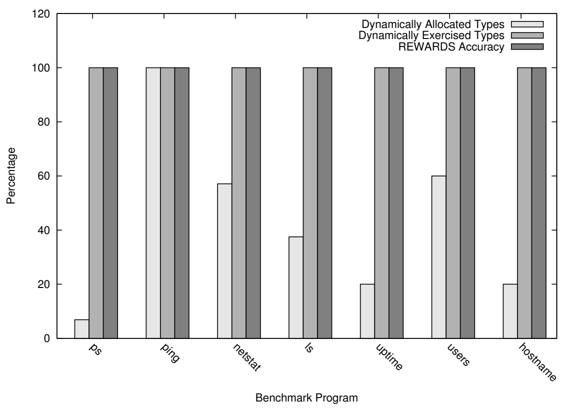
To evaluate the reverse engineering accuracy of REWARDS, we compare the derived data structure types with those declared in the program source code. To acquire the oracle information, we recompile the programs with debugging information, and then use `libdwarf` [1] to extract type information from the binaries. The `libdwarf` library is capable of presenting the stack and global variable mappings after compilation. For instance, global variables scattering in various places in the source code will be organized into a few data sections. The library allows us see the organization. In particular, `libdwarf` extracts stack variables by presenting the mapping from their offsets in the stack frame and the corresponding types. For global variables, the output by `libdwarf` is program virtual addresses and their types. Such information allows us to conduct direct and automated comparison. Note that we only verify the types in `.data`, `.bss`, and `.rodata` sections, other global data in sections such as `.got`, `.ctors` are not verified. For heap variables, since we use the execution context at allocation sites as the abstract representation, given an allocation context, we can locate it in the disassembled binary, and then correlate it with program source code to identify the heap data structure definition, and finally compare it with REWARDS's output. Although REWARDS extracts variable types for the entire program address space (including libraries), we only compare the results for user-level code.

The result for stack variables is presented in Figure 2(a). The figure presents the percentage of (1) functions that are actually executed, (2) data structures that are used in the executed functions (over all structures declared in those functions), and (3) data structures whose types are accurately recovered by REWARDS (over those in (2)). At runtime, it is often the case that even though a buffer is defined in the source code with size n, only part of the n bytes are used. Consequently, only those used ones are typed (the others are considered `unused`). We consider the buffer is correctly typed if its bytes are either correctly typed or `unused`. From the figure, we can observe that, due to the nature of dynamic analysis, not all functions or data structures in a function are exercised and hence amenable to REWARDS. More importantly, REWARDS achieves an average of 97% accuracy (among these benchmarks) for the data structures that get exercised. For heap variables, the result is presented in Figure 2(b), the bars are similarly defined. REWARDS's output perfectly matches the types in the original definitions when they are exercised. Note some of the benchmarks are missing in Figure 2(b) (e.g., `date`) because their executions do not allocate any user-level heap structures. The result for global variables is presented in Figure 2(c), and REWARDS achieves over 85% accuracy.

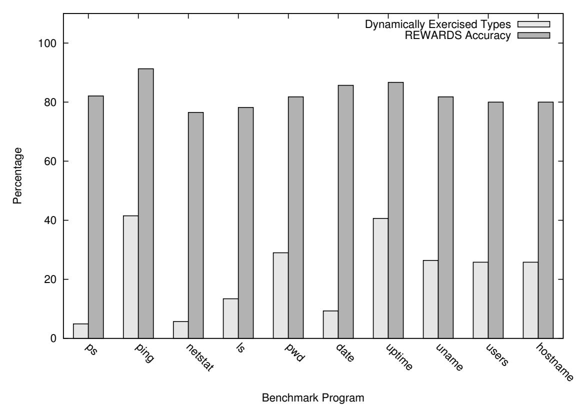
To explain why REWARDS cannot achieve 100% accu-



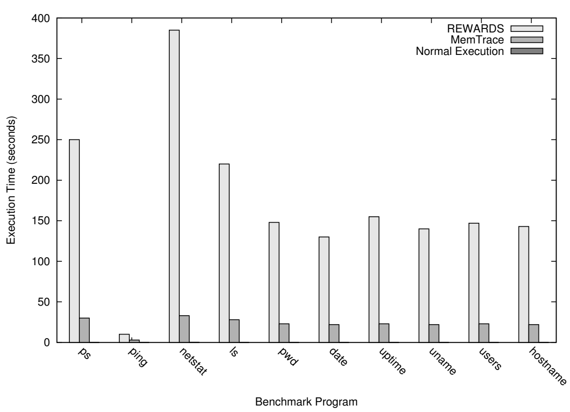
(a) Accuracy on Stack Variables



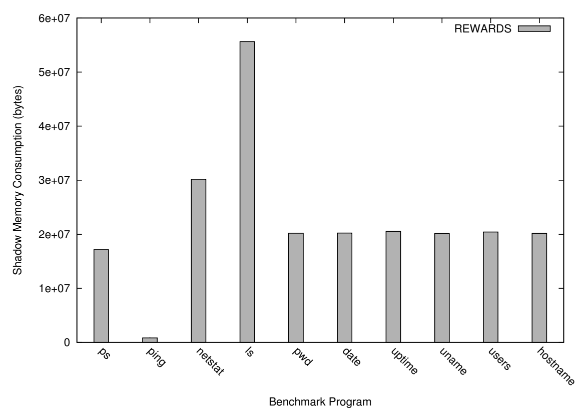
(b) Accuracy on Heap Variables



(c) Accuracy on Global Variables



(d) Performance Overhead



(e) Space Overhead

Figure 2. Evaluation results for REWARDS accuracy and efficiency

racy, we carefully examined the benchmarks and identified the following two reasons:

- **Hierarchy loss**. If a hierarchical structure becomes flat after compilation, we are not able to identify its hierarchy. This happens to structures declared as global variables or stack variables. And the binary never accesses such a variable using the base address plus a local offset. Instead, it directly uses a global offset (starting from the base address of the global data section or a stack frame). In other words, multiple composite structures are flattened into one large structure. In contrast, such flattening does not happen to heap structures.

- **Path-sensitive memory reuse**. This often happens to stack variables. In particular, the compiler might assign different local variables declared in different program paths to the same memory address. As a result, the types of these variables are undesirably unified in our current design. A more thorough design would use a *path-sensitive* local offset to denote a stack variable.

Despite the imperfect accuracy, REWARDS still suits our targeted application scenarios, i.e., memory forensics and vulnerability fuzzing. For example, although REWARDS outputs a flat layout for all global and stack variables, we can still conduct vulnerability fuzzing because the absolute offsets of these variables are sufficient; and we can still construct hierarchical views of memory images as pointer types can be obtained.

4.2 Evaluation of Efficiency

We also measured the time and space overhead of REWARDS. We compared it with (1) a standard memory trace tool, MemTrace (shipped along with PIN-2.6) and (2) the normal execution of the program, to evaluate the performance overhead. The result is shown in Figure 2(d). Note the normal execution data is nearly not visible in this figure because they are very small (roughly at the 0.01 second level). We can observe that REWARDS causes slowdown in the order of ten times compared with MemTrace, and in the order of thousands (or tens of thousands) times compared with the normal execution.

For space overhead, we are interested in the space consumption by shadow type sets and constraint sets. Hence, we track the peak value of the shadow memory consumption. The result is shown in Figure 2(e). We can observe that the shadow memory consumption is around 10 Mbytes for these benchmarks. A special case is `ping`, which uses much less memory. The reason is that it has fewer function calls and memory allocations, which is also why it runs much faster than the other programs shown in Figure 2(d).

5 Applications of REWARDS

REWARDS can be applied to a number of applications. In this section, we demonstrate how REWARDS provides unique benefits to (1) memory image forensics and (2) binary vulnerability fuzz.

5.1 Memory Image Forensics

Memory image forensics is a process to extract meaningful information from a memory dump. Examples of such information are IP addresses that the application under investigation is talking to and files being accessed. Data structure definitions play a critical role in the extraction process. For instance, without data structure information, it is hard to decide if four consecutive bytes represent an IP address or just a regular value. REWARDS enables analyzing memory dumps for a binary without symbolic information. In this subsection, we demonstrate how REWARDS can be used to type reachable memory as well as some of the unreachable (i.e., dead) memory.

5.1.1 Typing Reachable Memory

In this case study, we demonstrate how we use REWARDS to discover IP addresses from a memory dump using the hierarchical view (Section 3.5). We run a web server `nullhttpd-0.5.1`. A client communicates with this server through wget (`wget-1.10.2`). The client has IP `10.0.0.11` and the server has IP `10.0.0.4`. The memory dump is obtained from the server at the moment when a system call is invoked to close the client connection. Part of the memory dump is shown in Figure 3. The IPs are underlined in the figure. From the memory dump, it is very hard for human inspectors to identify those IPs without a meaningful view of the memory. We use REWARDS to derive the data structure definitions for `nullhttpd` and then construct a hierarchical view of the memory dump following the method described in Section 3.5.

The relevant part of the reconstructed view is presented in Figure 4(a). The root represents a pointer variable in the global section. The outgoing edge of the root leads to the data structure being pointed to. The edge label "`struct_0x0804dd4f *`" denotes that this is a heap data structure whose allocation PC (also its abstraction) is `0x0804dd4f`. According to the view construction method, the memory region being pointed to is typed according to the derived definition of the data structure denoted by `0x0804dd4f`, resulting in the second layer in Figure 4(a). The memory region starts at `0x08052170` is denoted by the node with the address label. The individual child nodes represent the different fields of the structure, e.g. the first field is a thread id according to the semantic tag `pthread_t`, the fourth field (with offset +12) denotes

```
...
08052170  b0 5b fe b7 b0 5b fe b7  05 00 00 00 02 00 92 7e
08052180  0a 00 00 0b 00 00 00 00  00 00 00 00 c7 b0 af 4a
08052190  c7 b0 af 4a 00 00 00 00  58 2a 05 08 00 00 00 00
080521a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
...
08052a50  00 00 00 00 59 31 01 00  4b 65 65 70 2d 41 6c 69
08052a60  76 65 00 00 00 00 00 00  00 00 00 00 00 00 00 00
08052a70  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08052ee0  00 00 00 00 00 00 00 00  00 00 00 00 31 30 2e 30
08052ef0  2e 30 2e 34 00 00 00 00  00 00 00 00 00 00 00 00
08052f00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08052fe0  00 00 00 00 00 00 00 00  00 00 00 00 48 54 54 50
08052ff0  2f 31 2e 30 00 00 00 00  00 00 00 00 00 00 00 00
08053000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08053470  00 00 00 00 00 00 00 00  00 00 00 00 31 30 2e 30
08053480  2e 30 2e 31 31 00 00 00  00 00 00 00 00 00 00 00
08053490  47 45 54 00 00 00 00 00  2f 00 00 00 00 00 00 00

080534a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08053910  00 00 00 00 00 00 00 00  57 67 65 74 2f 31 2e 31
08053920  30 2e 32 00 00 00 00 00  00 00 00 00 00 00 00 00
08053930  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08053990  00 00 00 00 00 00 00 00  c8 00 00 00 00 00 00 00
080539a0  00 00 00 00 00 00 00 00  00 00 43 6c 6f 73 65 00
080539b0  00 00 00 00 00 00 00 00  00 00 00 00 52 00 00 00
080539c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08053a90  48 54 54 50 2f 31 2e 30  00 00 00 00 00 00 00 00
08053aa0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08053b20  74 65 78 74 2f 68 74 6d  6c 00 00 00 00 00 00 00
08053b30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
08063ba0  01 00 01 00 01 00 00 00  00 00 00 00 00 00 00 00
08063bb0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
...
```

Figure 3. Part of a memory dump from null-httpd

a `sockaddr` structure. The last field (with offset +40) denotes another heap structure whose allocation site is `0x0804ddfb`. Transitively, our method reconstructs the entire hierarchy.

The extraction of IP addresses is translated into a traversal over the view to identify those with the IP address semantic tags. Along the path `08050260` → `08052170` → `7e9200...0` → `0x0b0000a`, a variable with the `sin_addr` type can be identified, which stores the client IP. The same IP can also be identified along the path `08050260` → `08052170` → `08052a58` → `10.0.0.11`, with the field offset `+2596`. The field has the `ip_addr_str_t` tag, which is resolved at the return of a call to `inet_ntoa()`. REWARDS is able to isolate the server IP `10.0.0.4` as a string along the path `08050260` → `08051170` → `10.0.0.4` with the field offset `+1172`. Interestingly, this field does not have a semantic tag related to an IP address. The reason is that the field is simply a part of the request string (the host field in HTTP Request Message), but it is not used in any type sinks that can resolve it as an IP. However, isolating the string also allows a human inspector to extract it as an IP.

To validate our result, we present in Figure 4(b) the corresponding symbolic definitions extracted from the source for comparison. Fields that are underlined are used during execution. In particular, struct `CONNECTION` corresponds to the abstraction `struct_0x0804dd4f` (node `08052170`) and struct `CONNDATA` corresponds to `struct_0x0804ddfb` (node `08052a58`). Observe that all fields of `CONNECTION` are precisely derived, except the pointer `PostData`, which is represented as an unused array in the inferred definition because the field is not used during execution. For the `CONNDATA` structure, all the exercised fields are extracted and correctly typed. Recall that we consider a field is correctly typed if its offset is correctly identified and its composition bytes are either correctly typed or unused.

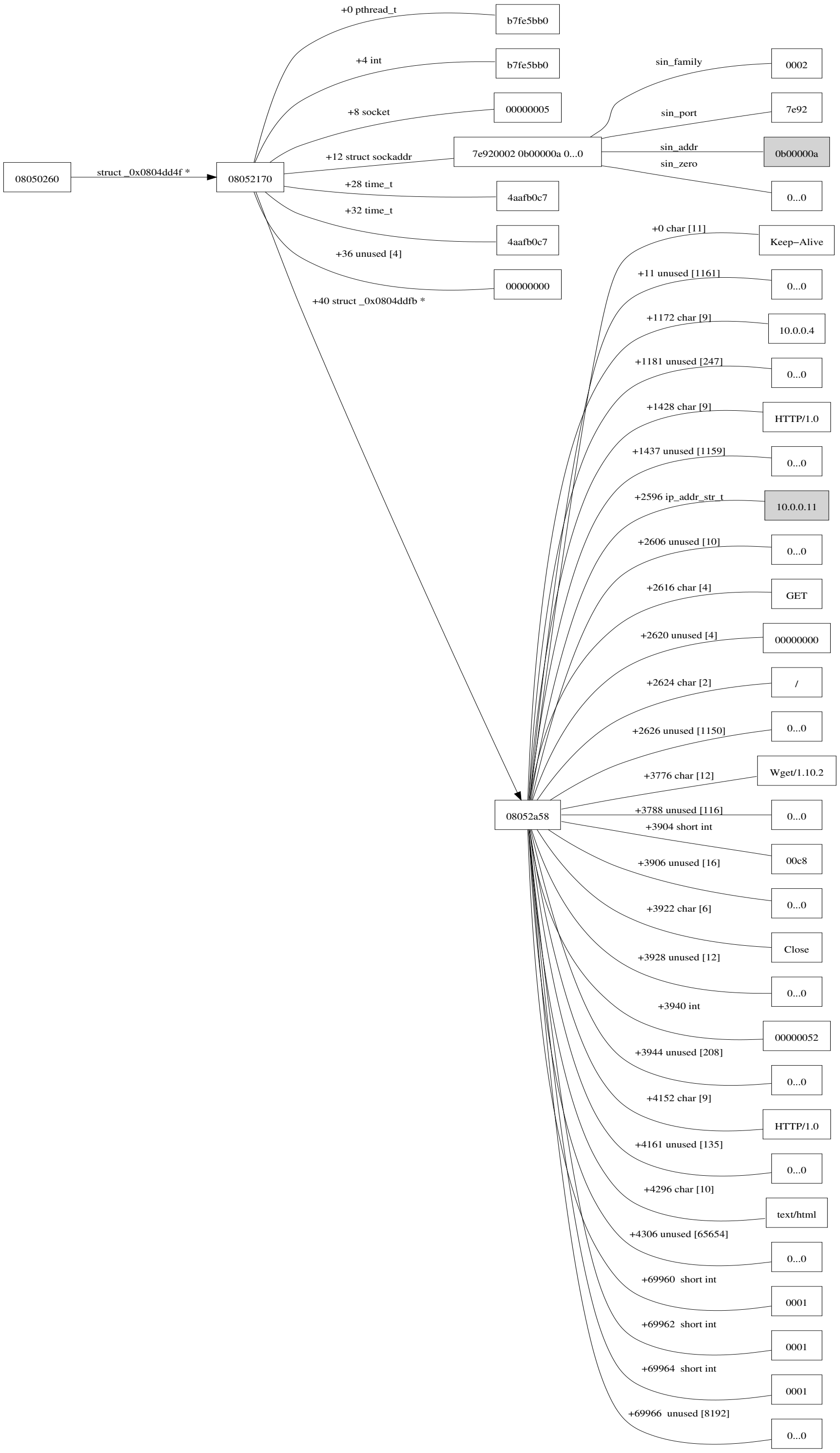
5.1.2 Typing Dead Memory

In this case, we demonstrate how to type dead memory, i.e., memory regions containing dead variables, using the slapper worm bot-master program. Slapper worm relies on P2P communications. The bot-master uses a program called `pudclient` to control the P2P botnet, such as launching TCP-flood, UDP-flood, and DNS-flood attacks. Our goal is to extract evidence from a memory dump of `pudclient` from the attacker's machine.

Our experiment has two scenes: the investigator's scene and the attacker's scene. More specifically,

- Scene I: In the lab, the investigator runs the bot-master program `pudclient` to communicate with slapper bots to derive the data structures of `pudclient`.
- Scene II: In the wild, the attacker runs `pudclient` to control real slapper bots.

In Scene I, we run a number of slapper worm instances in a contained environment (at IP addresses ranging from `10.0.0.1` - `10.0.1.255`). Then we launch `pudclient` with REWARDS and issue a series of commands such as listing the compromised hosts, and launching the UDPFlood, TCPFlood, and DNSFlood attacks. REWARDS extracts the data structure definitions for `pudclient`. Then in Scene II, we run `pudclient` again without REWARDS. Indeed, the attacker's machine does not have any forensics tool running. Emulating the attacker, we issue some commands and then hibernate the machine. We then get the memory image of `pudclient` and use the data structure information derived in Scene I to investigate the image.



(a) Hierarchical view from REWARDS

```
typedef struct {
    pthread_t handle;
    unsigned long int id;
    short int socket;
    struct sockaddr_in ClientAddr;
    time_t ctime; // Creation time
    time_t atime; // Last Access time
    char *PostData;
    CONNDATA *dat;
} CONNECTION;


CONNECTION *conn;  //matched the root node


typedef struct {
    // incoming data
    char in_Connection[16];
    int  in_ContentLength;
    char in_ContentType[128];
    char in_Cookie[1024];

    char in_Host[64];
    char in_IfModifiedSince[64];
    char in_PathInfo[128];

    char in_Protocol[16];
    char in_QueryString[1024];
    char in_Referer[128];

    char in_RemoteAddr[16];
    int  in_RemotePort;

    char in_RequestMethod[8];

    char in_RequestURI[1024];
    char in_ScriptName[128];

    char in_UserAgent[128];

    // outgoing data
    short int out_status;
    char out_CacheControl[16];

    char out_Connection[16];

    int  out_ContentLength;
    char out_Date[64];
    char out_Expires[64];
    char out_LastModified[64];
    char out_Pragma[16];


    char out_Protocol[16];
    char out_Server[128];


    char out_ContentType[128];
    char out_ReplyData[MAX_REPLYSIZE];

    short int out_headdone;
    short int out_bodydone;
    short int out_flushed;

    // user data
    char envbuf[8192];
} CONNDATA;
```

(b) Data structure definition

Figure 4. Comparison between the REWARDS-derived hierarchical view and source code definition

```
bfffd140  05 00 00 00 6b 00 00 00  69 00 00 00 00 00 00 00
bfffd150  00 00 00 00 38 ea ff bf  00 00 00 00 00 00 00 01
bfffd160  2c 00 00 00 67 45 8b 6b  0e 00 00 00 00 00 00 00
bfffd170  0a 00 00 63 0f 27 00 00  9f 86 01 00 9f 86 01 00
bfffd180  1c ea ff bf 10 ea ff bf  6a f2 b2 4a 7a 4a 0e 00
bfffd190  22 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
bfffd1a0  6a f2 b2 4a 7a 4a 0e 00  f2 f3 8d 8c 00 00 00 00
bfffd1b0  00 00 00 00 00 00 00 00  01 00 00 00 02 00 00 00
bfffd1c0  64 6e 73 66 6c 6f 6f 64  00 00 00 00 00 00 00 00
bfffd1d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
bfffd5c0  c0 d1 ff bf 00 00 00 00  02 ca 04 08 00 00 00 00
bfffd5d0  00 00 00 00 00 00 00 00  02 ca 04 08 02 ca 04 08
bfffd5e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
bfffd5f0  00 00 00 00 00 00 00 00  00 00 00 00 04 d6 ff bf
bfffd600  64 6e 73 66 6c 6f 6f 64  00 00 00 00 00 00 00 00
bfffd610  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
bfffe5b0  00 00 00 00 00 00 00 00  0e 00 00 00 00 00 00 00
bfffe5c0  00 00 00 00 02 00 4e 34  0a 00 00 0b 00 00 00 00
bfffe5d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
bfffe5e0  00 00 00 00 00 00 00 00  00 00 00 00 e0 f5 ff bf
bfffe5f0  a0 2d 05 08 e0 f5 ff bf  a0 13 05 08 00 00 00 00
bfffe600  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
bfffea00  00 00 00 00 00 00 00 00  00 00 00 00 10 ea ff bf
bfffea10  01 00 00 00 00 00 00 00  e5 de f2 49 46 00 00 00
bfffea20  67 45 8b 6b 10 00 00 00  e8 be e6 71 0a 00 00 34
bfffea30  0a 00 01 33 0a 00 00 0b  0a 00 00 04 00 00 00 00
bfffea40  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
*
...
bffff5c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
bffff5d0  01 00 00 00 80 00 00 00  80 00 00 00 ff f7 ff bf
bffff5e0  00 00 00 00 00 00 00 00  f3 f7 ff bf 67 45 8b 6b
bffff5f0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
bffff600  01 00 00 00 c0 f6 ff bf  28 f6 ff bf fb c7 04 08
bffff610  02 00 00 00 dc 3a 1f b6  d4 df 04 08 dc 3a 1f b6
bffff620  00 00 00 00 dc 3a 1f b6  88 f6 ff bf a2 de 0d b6
bffff630  02 00 00 00 b4 f6 ff bf  c0 f6 ff bf f6 5b ff b7
```

Figure 5. Memory dump for Slapper worm control program when exiting the control interface

We construct the hierarchical view and try to identify IP addresses from the view. However, the hierarchical view can only map the memory locations that are alive, namely they are reachable from global and stack (pointer) variables. Here, we take an extra step to type the dead (unreachable) data. As described in Section 3.5, our technique scans the stack space lower than the current (the lowest and live) activation record and looks for values that are in the range of the code section, as they are very likely return addresses. Four such values are identified. One example and its memory context is shown in Figure 5. In this memory dump snippet, the return address, as underlined, is located at address `0xbffff62c`. Our technique further identifies that the corresponding function invocation is to `0x0804a708`. Hence, we use the data structure definition of `fun_0x0804a708` to type the activation record. The definition and the typed values are shown in Table 3. Observe that a number of IPs (fields with `ip_addr_t`) are identified. We also spot the bot command "`dnsflood`" at `-9324` and `-8236`. Note that these two fields have the `input_t` tag as part of their derived definition, indicating they hold values from input.

5.2 Vulnerability Fuzz

It is a challenging task to detect and confirm vulnerabilities in a given binary without symbolic information. Previously in [26], we have proposed a dynamic analysis approach that can decide if a vulnerability suspect is true positive by generating a concrete exploit. The basic idea is to first use existing static tools to identify vulnerability candidates, which are often of large quantity; then benign executions are mutated to generate exploits. Mutations are directed by dynamic information called *input lineage*, which denotes the set of input elements that is used to compute a value at a given execution point, usually a vulnerability candidate. Vulnerability-specific patterns are followed during mutation. One example pattern is to exponentially expand an input string in the lineage of a candidate buffer with the goal of generating an overflow exploit. In that project, we had difficulty finding publicly available, *binary-level* vulnerability detectors to use as the front end. REWARDS helps address this issue by deriving both variable syntax and semantics from a subject binary. Next, we present our experience of using REWARDS to identify vulnerability suspects and then using our prior system (a fuzzer) to confirm them.

For this study, we design a static vulnerability suspect detector that relies on the variable type information derived by REWARDS. The result of the detector is passed to our lineage-based fuzzer to generate exploits. In the following, we present how REWARDS helps identify various types of vulnerability suspects.

- **Buffer overflow vulnerability.** Buffer overflows could happen in three different places: stack, heap, and global areas. As such, we define three types of buffer overflow vulnerability patterns. Specifically, for stack overflow, if a stack layout contains a buffer and its content comes from user input, we consider it a suspect. Note that this can be easily facilitated by REWARDS's typing algorithm: A semantics tag `input_t` is defined to indicate that a variable receives its value from external input. *The tag is only susceptible to the forward flow but not the backward flow*. In the stack layout derived by REWARDS, if a buffer's type set contains an `input_t` tag, it is considered vulnerable. For heap overflow, we consider two cases: one is to exploit heap management data structure outside the user-allocated heap chunk; and the other is to exploit user-defined function pointers inside the heap chunk. Detecting the former case is simply to check if a heap structure contains a buffer

Offset	Type	Size	Mem Addr	Content	Offset	Type	Size	Mem Addr	Content
-9432	void*	4	bfffd154	38 ea ff bf	-9324	char[9],input_t	9	bfffd1c0	64 6e..64
-9428	char*	4	bfffd158	00 00 00 00	-8300	char*	4	bfffd5c0	c0 d1 ff bf
-9420	int	4	bfffd160	2c 00 00 00	-8236	char[9],input_t	9	bfffd600	64 6e..64
-9416	int	4	bfffd164	67 45 8b 6b	-8227	char[28]	28	bfffd609	00 .. 00
-9412	int	4	bfffd168	0e 00 00 00	-4236	void*	4	bfffe5a0	00 00 00 00
-9408	int	4	bfffd16c	00 00 00 00	-4156	struct_0x804834e*	4	bfffe5f0	a0 2d 05 08
-9404	ip_addr_t	4	bfffd170	0a 00 00 63	-4152	void*	4	bfffe5f4	e0 f5 ff bf
-9300	port_t	4	bfffd174	0f 27 00 00	-3104	char*	4	bfffea0c	10 ea ff bf
-9396	int	4	bfffd178	9f 86 01 00	-3088	char[16]	16	bfffea1c	46 00 00 00
-9392	int	4	bfffd17c	9f 86 01 00	-3068	ip_addr_t	4	bfffea30	0a 00 01 33
-9388	void*	4	bfffd180	1c ea ff bf	-3064	ip_addr_t	4	bfffea34	0a 00 00 0b
-9384	void*	4	bfffd184	10 ea ff bf	-3058	ip_addr_t	4	bfffea38	0a 00 00 04
-9376	timeval.tv_sec	4	bfffd18c	7a 4a 0e 00	-3054	ip_addr_t	4	bfffea3c	0a 00 00 04
	timeval.tv_usec	4	bfffd190	22 00 00 00	-0088	int	4	bffff5d4	80 00 00 00
-9368	int	4	bfffd194	00 00 00 00	-0084	int	4	bffff5d8	80 00 00 00
-9352	int	4	bfffd1a4	7a 4a 0e 00	-0080	int	4	bffff5dc	ff f7 ff bf
-9348	int	4	bfffd1a8	f2 f3 8d 8c	-0004	stack_frame_t	4	bffff628	88 f6 ff bf
-9344	int	4	bfffd1ac	00 00 00 00	+0000	ret_addr_t	4	bffff62c	a2 de 0d b6
-9332	int	4	bfffd1b8	01 00 00 00	+0004	int	4	bffff630	02 00 00 00
-9328	int	4	bfffd1bc	02 00 00 00	+0008	char*	4	bffff634	b4 f6 ff bf

Table 3. Result on the unreachable memory type using type fun_0x804a708

field that is input-relevant, in a way similar to stack vulnerability detection. For the later case, the detector scans the derived layout of a heap structure to check the presence of both an input-relevant buffer field and a function pointer field. Vulnerabilities in the global memory region are handled similarly.

- **Integer overflow vulnerability.** Integer overflow occurs when an integer exceeds the maximum value that a machine can represent. Integer overflow itself may not be harmful (e.g., `gcc` actually leverages integer overflow to manipulate control flow path condition [38]), but if an integer variable is dependent on user input without any sanity check and it is used as an argument to `malloc`-family functions, then an integer overflow vulnerability is likely. In particular, overflowed values passed to malloc functions usually result in heap buffers being smaller than they are supposed to be. Consequently, heap overflows occur. For this type of vulnerabilities, our detector checks the actual arguments to malloc family function invocations: if an integer parameter has both `malloc_arg_t` and `input_t` tags, an integer overflow vulnerability suspect will be reported.

- **Format string vulnerability.** The format string vulnerability pattern involves a user input flowing into a format string argument. Thus, we introduce a semantics tag `format_string_t`, which is only resolved at invocations to `printf`-family functions. If a variable's type set contains both `input_t` and `format_string_t` tags, a format string vulnerability suspect is reported.

Besides facilitating vulnerability suspect identification, the information generated by REWARDS can also help *composing exploits*. For instance, it is critical to know the distance between a vulnerable stack buffer and a return address, i.e., a variable with the `ret_addr_t` tag, in order to construct a stack overflow exploit. Similarly, it is important to know the distance between a heap buffer and a heap function pointer for composing a heap overflow-based code injection attack. Such information is provided by REWARDS.

Program	#Buffer Overflow	#Integer Overflow	#Format String
ncompress-4.2.4	1	0	0
bftpd-1.0.11	3	0	0
gzip-1.2.4	3	0	0
nullhttpd-0.5.0	5	2	0
xzgv-5.8	3	8	0
gnuPG-1.4.3	0	3	0
ipgrab-0.9.9	0	5	0
cfingerd-1.4.3	4	0	1
ngircd-0.8.2	12	0	1

Table 4. Number of vulnerability suspects reported with help of REWARDS

We applied our REWARDS-based detector to examine several programs shown in the 1^{st} column of Table 4. The detector reported a number of vulnerable suspects based on the aforementioned vulnerability patterns. The total number of vulnerabilities of each type is presented in the remaining columns. Observe that our detector does not produce many suspects for these programs and hence can serve as a tractable front end for our fuzzer. The fuzzer then tries to generate exploits to convict the suspects. Details of each confirmed vulnerable data structure is shown in the 2^{nd} column of Table 5. The field symbols do not represent their symbolic names, which we do not know, but rather the type tags derived for these fields. For instance, `format_string_t` denotes that the field is essentially a format string; `sockaddr_in` indicates that the field holds a socket address. The 3^{rd} column presents the input category that is relevant to the vulnerable data structure.

Benchmark	Suspicious Data Structure	Input	Offset	Vulnerability Type
ncompress-4.2.4	fun_0x08048e76 { -1052: **char[13]**, -1039: unused[1023],... -0008: char*, -0004: stack_frame_t, +0000: ret_addr_t, +0004: char**}	argv[1]	{0..11}	Stack overflow
bftpd-1.0.11	fun_0x080494b8 { -0064: char*, -0060: **char[12]**, -0048: unused [44], -0004: stack_frame_t, +0000: ret_addr_t, +0004: char*}	recv	{0..3}	Stack overflow
gzip-1.2.4	bss_0x08053f80 { ... +244128: **char[8]**, +244136: unused[1016], +245152: char*,...}	argv[1]	{0..6}	Global overflow
nullhttpd-0.5.0	heap_0x0804f205 { +0000: char[11], +0011: unused[5], +0016: **int**,... }	recv	{607,608}	Integer overflow
	heap_0x0804c41f {+0000: **void[29]**, +0029: unused[1024]}	recv	{661..690}	Heap Overflow
xzgv-5.8	bss_0x0809ac80 { ... +91952: **int**, +91956: **int**,...}	fread	{4..11}	Integer overflow
gnuPG-1.0.5	fun_0x080673fc { ..., -0176: char[6],unused[2], -0168: **int**,int,...}	fread	{2..5}	Integer overflow
	heap_0x080afec1 { +0000:int,..., +0036: **void[5]** }	fread	{6..10}	Heap overflow
ipgrab-0.9.9	fun_0x0804d06b { ..., -0056: int, -0052: **int**, int,...}	fread	{20..23}	Integer overflow
	heap_0x0805a976 {+0000: **void[60]** }	fread	{40..100}	Heap overflow
cfingerd-1.4.3	fun_0x080496b8 { ..., -0440: struct sockaddr_in, -0424: **format_string_t[34]**, -0390: unused[174], -0216: char[4],,...}	read	{0..3}	Format String
ngircd-0.8.2	fun_0x0805f9a5 { ..., -0284: **format_string_t[76]** -0208: unused[204], -0004: stack_frame_t, +0000: ret_addr_t,...}	recv	{12..15}	Format String

Table 5. Result from our vulnerability fuzzer with help of REWARDS

For example, the char[12] buffer in bftpd denotes a packet received from outside (the recv category). Note that the input categories are conveniently implemented as semantics tags in REWARDS. The 4^{th} column offset represents the input offsets reported by our fuzzer. They represent the places that are mutated to generate the real exploits. The REWARDS-based vulnerability detector also emits vulnerability types (shown in the 5^{th} column) based on the vulnerability patterns matched. Consider the first benchmark ncompress: Its entry in the table indicates that the char[13] buffer inside a function starting with PC 0x08048e76 is vulnerable to stack buffer overflow. The buffer receives values from the second command line option (argv[1]). Our data lineage fuzzer mutates the lineage of the buffer, which are the first 12 input items (offset 0 to 11) to generate the exploit. From the data structure in the 2^{nd} column, the exploit has to contain a byte string longer than 1052 bytes to overwrite the return address at the bottom. Other vulnerabilities can be similarly apprehended.

6 Discussion

REWARDS has a number of limitations: (1) As a dynamic analysis-based approach, REWARDS cannot achieve full coverage of data structures defined in a program. Instead, the coverage of REWARDS relies on those data structures that are actually created and accessed during a particular run of the binary. (2) REWARDS is not fully on-line as our timestamp-based on-line algorithm may leave some variables unresolved by the time they are de-allocated, and thus the off-line companion procedure is needed to make the system sound. A fully on-line type resolution algorithm is our future work. (3) Based on PIN, REWARDS does not support the reverse engineering of kernel-level data structures. (4) REWARDS does not work with obfuscated code. Thus it is possible that an adversary can write an obfuscated program to dodge REWARDS – for example, by avoiding touching the type sinks we define. (5) Besides the general data structures, REWARDS has yet to support the extraction of other data types, such as the format of a specific type of files (e.g., ELF files, multimedia files), and browser-related data types (e.g., URL, cookie). Moreover, REWARDS does not distinguish between sign and unsigned integers in our current implementation.

7 Related Work

Type inference. Some programming languages, such as ML, do not explicitly declare types. Instead, types are inferred from programs. Typing constraints are derived from program statements statically and programs are typed by solving these constraints. Notable type inference algorithms include Hindley-Milner algorithm [29], Cartesian Product algorithm [3], iterative type analysis [13], object oriented type inference [33], and aggregate structure identification [35].

These techniques, like REWARDS, rely on type unification, namely, variables connected by operators shall have the same type. However, these techniques assume program source code and they are static, that is, typing constraints are generated from source code at compile time. For REWARDS, we only assume binaries without symbolic information, in which high level language artifacts are all broken down to machine level entities, such as registers, memory addresses, and instructions. REWARDS relies on type sinks to obtain the initial type and semantics information. Variables are then typed through unification with type sinks during execution.

Lately, Balakrishnan et al. [4, 5, 36] showed that analyzing executables alone can largely discover syntactic structures of variables, such as sizes, field offsets, and simple structures. Their technique entails points-to analysis and abstract interpretation at binary level. They cannot handle obfuscated binaries and dynamically loaded libraries. Furthermore, the inaccuracy of binary points-to analysis makes it hard to type heap variables. In comparison, our technique is relatively simple, with the major hindrances to static analysis (e.g., points-to relations and dynamically loaded libraries) addressed via dynamic analysis.

Abstract type inference. Abstract type inference [32] is to group typed variables according to their semantics. For example, variables that are meant to store money, zip codes, ages, etc., are clustered based on their intention's, even though they may have the same integer type. Such an intention is called an abstract type. The technique relies on the Hindley-Milner type inference algorithm. Recently, dynamic abstract type inference is proposed [24] to infer abstract types from execution. Regarding the goal of performing semantics-aware typing, these techniques and ours are similar. However, they work at the source code level whereas ours works at the binary level. Our technique further derives syntactic type structures.

Decompilation. Decompilation is a process of reconstructing program source code from lower-level languages (e.g., assembly or machine code) [14, 20, 6]. It usually involves reconstructing variable types [31, 19]. By using unification, Mycroft [31] extends the Hindley-Milner algorithm [29] and delays unification until all constraints are available. Recently, Dolgova and Chernov [19] present an iterative algorithm that uses a lattice over the properties of data types for reconstruction.

All these techniques are static and hence share the same limitations of static type inference and they only derive simple syntactic structures. Moreover, they aim to get an execution-equivalent code and do not pay attention to whether the recovered types reflect the original declarations and have the same structures.

Protocol format reverse engineering. Recent efforts in protocol reverse engineering involve using dynamic binary analysis (in particular input data taint analysis) to reveal the format of protocol messages, facilitated by instruction semantics (e.g., Polyglot [9]) or execution context (e.g., AutoFormat [25]). Recently, it has been shown that the BNF structure of a given protocol with multiple messages can be derived [40, 17, 28]; and the format of out-going messages as well as encrypted messages can be revealed [8, 39]. In particular, REWARDS shares the same insight as Dispatcher [8] for type inference and semantics extraction. The difference is that Dispatcher and other protocol reverse engineering techniques mainly focus on live input and output messages, whereas we strive to reveal general data structures in a program. Meanwhile, we care more about the detailed in-memory layout of program data, motivated by our different targeted application scenarios.

Memory forensics and vulnerability discovery. FATKit [34] is a toolkit to facilitate the extraction, analysis, aggregation, and visualization of forensic data. Their technique is based on pre-defined data structures extracted from program source code to type memory dumps. This is also the case for other similar systems (e.g., [12, 30, 2]). KOP [11] is an effective system that can map dynamic kernel objects with nearly complete coverage and perfect accuracy. It also relies on program source code and uses an inter-procedural points-to analysis to compute all possible types for generic pointers. There are several other efforts [37, 18] that use data structure signatures to scan and type memory. Complementing these efforts, REWARDS extracts data structure definitions and reconstructs hierarchical in-memory layouts from binaries.

There is a large body of research in vulnerability discovery such as Archer [41], EXE [10], Bouncer [15], BitScope [7], DART [22], and SAGE [23, 21]. REWARDS complements these techniques by enabling identification of vulnerability suspects directly from binaries.

8 Conclusion

We have presented the REWARDS reverse engineering system that automatically reveals data structures in a binary based on dynamic execution. REWARDS involves an algorithm that performs data flow-based type attribute forward propagation and backward resolution. Driven by the type information derived, REWARDS is also capable of reconstructing the structural and semantic view of in-memory data layout. Our evaluation using a number of real-world programs indicates that REWARDS achieves high accuracy in revealing data structures accessed during an execution. Furthermore, we demonstrate the benefits of REWARDS to two application scenarios: memory image forensics and binary vulnerability discovery.

9 Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. We are grateful to Xuxian Jiang and Heng Yin for earlier discussions and help on this and related problems. This research is supported, in part, by the Office of Naval Research (ONR) under grant N00014-09-1-0776 and by the National Science Foundation (NSF) under grant 0720516. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the ONR or NSF.

References

[1] Libdwarf. *http://reality.sgiweb.org/davea/dwarf.html*.

[2] Mission critical linux. In Memory Core Dump, http://oss.missioncriticallinux.com/projects/mcore/.

[3] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, pages 2–26, London, UK, 1995. Springer-Verlag.

[4] G. Balakrishnan, , G. Balakrishnan, and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of International Conference on Compiler Construction (CC'04)*, pages 5–23. Springer-Verlag, 2004.

[5] G. Balakrishnan and T. Reps. Divine: Discovering variables in executables. In *Proceedings of International Conf. on Verification Model Checking and Abstract Interpretation (VMCAI'07)*, Nice, France, 2007. ACM Press.

[6] P. T. Breuer and J. P. Bowen. Decompilation: the enumeration of types and grammars. *ACM Trans. Program. Lang. Syst.*, 16(5):1613–1647, 1994.

[7] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Bitscope: Automatically dissecting malicious binaries, 2007. Technical Report CMU-CS-07-133, Carnegie Mellon University.

[8] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and and Communications Security (CCS'09)*, pages 621–634, Chicago, Illinois, USA, 2009.

[9] J. Caballero and D. Song. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and and Communications Security (CCS'07)*, pages 317–329, Alexandria, Virginia, USA, 2007.

[10] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS'06)*, pages 322–335, Alexandria, Virginia, USA, 2006. ACM.

[11] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *The 16th ACM Conference on Computer and Communications Security (CCS'09)*, pages 555–565, Chicago, IL, USA, 2009.

[12] A. Case, A. Cristina, L. Marziale, G. G. Richard, and V. Roussev. Face: Automated digital evidence discovery and correlation. *Digital Investigation*, 5(Supplement 1):S65 – S75, 2008. The Proceedings of the Eighth Annual DFRWS Conference.

[13] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 150–164, 1990.

[14] C. Cifuentes. Reverse Compilation Techniques. *PhD thesis, Queensland University of Technology*, 1994.

[15] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *Proceedings of the 21st ACM SIGOPS symposium on Operating systems principles (SOSP'07)*, pages 117–130, Stevenson, Washington, USA, 2007. ACM.

[16] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *Proceeding of 8th Symposium on Operating System Design and Implementation (OSDI'08)*, pages 231–244, San Diego, CA, December, 2008.

[17] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*, pages 391–402, Alexandria, Virginia, USA, October 2008.

[18] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS'09)*, pages 566–577, Chicago, Illinois, USA, 2009. ACM.

[19] E. N. Dolgova and A. V. Chernov. Automatic reconstruction of data types in the decompilation problem. *Program. Comput. Softw.*, 35(2):105–119, 2009.

[20] M. V. Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 27–36, 2004.

[21] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN*

Conference on Programming Language Design and Implementation (PLDI'08), pages 206–215, Tucson, AZ, USA, 2008. ACM.

[22] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pages 213–223, Chicago, IL, USA, 2005. ACM.

[23] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.

[24] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *Proceedings of the 2006 international symposium on Software testing and analysis (ISSTA'06)*, pages 255–265, Portland, Maine, USA, 2006. ACM.

[25] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.

[26] Z. Lin, X. Zhang, and D. Xu. Convicting exploitable software vulnerabilities: An efficient input provenance based approach. In *Proceedings of the 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'08)*, Anchorage, Alaska, USA, June 2008.

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pages 190–200, Chicago, IL, USA, 2005.

[28] P. Milani Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol Specification Extraction. In *IEEE Symposium on Security & Privacy*, pages 110–125, Oakland, CA, 2009.

[29] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[30] P. Movall, W. Nelson, and S. Wetzstein. Linux physical memory analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 39–39, Anaheim, CA, 2005. USENIX Association.

[31] A. Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems (ESOP'99)*, pages 208–223, London, UK, 1999. Springer-Verlag.

[32] R. O'Callahan and D. Jackson. Lackwit: a program understanding tool based on type inference. In *Proceedings of the 19th international conference on Software engineering*, pages 338–348, Boston, Massachusetts, United States, 1997. ACM.

[33] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 146–161, Phoenix, Arizona, United States, 1991. ACM.

[34] N. L. Petroni, Jr., A. Walters, T. Fraser, and W. A. Arbaugh. Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory. *Digital Investigation*, 3(4):197 – 210, 2006.

[35] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'99)*, pages 119–132, San Antonio, Texas, 1999. ACM.

[36] T. W. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In *Proceedings of International Conference on Compiler Construction (CC'08)*, pages 16–35, 2008.

[37] A. Schuster. Searching for processes and threads in microsoft windows memory dumps. *Digital Investigation*, 3(Supplement-1):10–16, 2006.

[38] T. Wang, T. Wei, Z. Lin, and W. Zou. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, San Diego, CA, February 2009.

[39] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proceedings of 14th European Symposium on Research in Computer Security (ESORICS'09)*, Saint Malo, France, September 2009. LNCS.

[40] G. Wondracek, P. Milani, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, San Diego, CA, February 2008.

[41] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-10)*, pages 327–336, Helsinki, Finland, 2003.