Technical Report IR-CS-55, February 2010

Dynamic data structure excavation

Asia Slowinska Traian Stancescu Herbert Bos

Vrije Universiteit Amsterdam

{asia,tsu500,herbertb}@few.vu.nl

Abstract

Dynamic Datastructure Excavation (DDE) is a new approach to extract datastructures from C binaries without any need for debugging symbols. Unlike most existing tools, DDE builds on dynamic analysis. Its results are much more accurate than those of previous methods.

1. Introduction

Debugging and reverse engineering of C binaries is difficult, especially in the absence of debugging symbols. Since programs tend to be developed around the data structures, they arguably represent the most important information that we need to recover. Unfortunately, data structure recovery is exceedingly hard. Even the most state of the art disassemblers and decompilers (like IDA Pro [9], CodeSurfer [3] and boomerang [10]), while fairly good at recovering code blocks, are hopeless at identifying data structures.

Data structure recovery is difficult, due to the gap between how data appears in the source and in the binary. The compilation process turns all variables into chunks of anonymous bytes. Data structure excavation is the art of mapping them back into meaningful data structures. To our knowledge, no existing work can do this.

In this paper, we sketch our solution for data structure excavation in x86 C binaries. Unlike most other approaches, we build DDE primarily on dynamic rather than static analysis following the simple intuition that memory access patterns reveal much about the layout of the data structures. DDE is able to recover most data structures in arbitrary (gcc-generated) binaries with a very high degree of precision. While it is too early to claim that the problem of data structure identification is solved, our work advances the state of the art significantly. For instance, we are the first to extract:

- precise data structures on both heap and stack;
- not just aggregate structures, also individual fields;
- complicated structures like nested arrays.

All dynamic analysis techniques were implemented in an instrumented processor emulator based on Qemu [5]. Since the processor emulator is available only for Linux, the implementation is also for Linux. However, the approach is not specific to a particular operating system.

2. Challenges

DDE recovers data structures by observing how memory is *used* at runtime. In the CPU, all memory accesses occur via pointers either using direct addressing or indirectly, via registers. The intuition behind our approach is that memory access patterns provide clues about the layout of data in memory. For instance, if `A` is a pointer, then a dereference of `*(A+4)` suggests that the programmer (and compiler) created a field of size `4` at `A`. Intuitively, if `A` is a function frame pointer, `*(A+4)` and `*(A-8)` are likely to point to a function argument passed via the stack, and a local variable, respectively. Likewise, if `A` is an address of a structure, `*(A+4)` presumably accesses a field in this structure, and finally, in the case of an `int[]` array, `*(A+4)` is its second element. Distinguishing between these three scenarios is one of the challenges we need to address. In this section, we discuss the major problems, and in Section 3 we explain how we tackle them.

Memory allocation context Our work aims to analyse a program's use of memory, which includes local function variables allocated on the stack, memory allocated on heap, and static variables. Both the runtime stack and heap are reused constantly, and so a description of data structures here needs to be coupled with a *context*.

For the stack, each invocation of a function usually holds the same set of local variables and therefore start addresses of functions are sufficient to identify function frames. A possible exception occurs with memory allocated by calls to functions like `alloca`, which *may* depend on the control flow. As a result, the frames of different invocations could differ. DDE handles these cases in a generic way, which is discussed in Section 3.3.3.

Heap memory, however, is more complicated. Consider a `my_malloc` wrapper function which invokes

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

void fun() {
 elem_t elem, *pelem;
 elem.x = 1;
 elem.y = 2;
 pelem = &elem;
 pelem->y = 3;
}
```

```
 <fun>:
[1] push %ebp
[2] mov %esp, %ebp
[3] sub $0x10, %esp
[4] mov $0x1, -0xc(%ebp)
[5] mov $0x2, -0x8(%ebp)
[6] mov -0x4(%ebp), %eax
[7] mov $0x3, 0x4(%eax)
[8] leave
[9] ret
```

Figure 1. The function initializes its local variable `elem`. Pointer `pelem` is located at offset `-4` in the function frame, and structure `elem` at `-0xc`. Instructions 4 and 5 initialize `x` and `y`, respectively. Register `eax` is loaded with the address of `pelem` in instruction 6, and used to update field `y` in 7.

`malloc` and checks whether the return value is null. Since `my_malloc` can be used to allocate memory for various structures and arrays, we should not associate the memory layout of a data structure allocated by `my_malloc` to `my_malloc` itself, but rather to its caller. As we do not know the number of such `malloc` wrappers in advance, we associate heap memory with a call stack (typically, the top 3 or 4 function calls are sufficient for accurate identification).

Static memory, finally, is not reused, and so can be uniquely identified solely with its address.

Pointer identification To analyze memory access patterns, we need to identify pointers in the running binary. Moreover, for a given address `B=A+4`, we need to know `A`, the *base* pointer from which `B` was derived. However, on architectures like `x86`, there is little distinction between registers used as addresses and scalars. Worse, the instructions to manipulate them are the same. We only know that a particular register holds a valid address when it is dereferenced. In our emulator, we thus track how new pointers are derived from existing ones.

Missing base pointers Recall that we recover data structures by observing memory accesses: new structure fields are detected when they are referenced from the structure base. However, structure fields are sometimes used without reference to a pointer to the data structure, and thus we may overlook the link between the fields. Figure 1 illustrates the problem. Notice that field `elem.y` is initialized via the frame pointer register `ebp` rather than the address of `elem`. Only the update instruction `7` hints at the existence of the structure. Otherwise, we would characterize this memory region as composed of 3 separate variables: `pelem`, `x`, `y` (on the other hand, since in that case the program does not actually use the connection between the fields `x` and `y`, this partially inaccurate result would be innocuous).

Multiple base pointers Another issue is that memory locations can be accessed through multiple base pointers, so we need to decide on the most appropriate one. Observe that field `elem.y` from Figure 1 is already referred to using two different base pointers, the frame pointer `ebp` and `pelem` (`eax`). Even though this particular case seems tractable (as `pelem` is itself based on `ebp`), the problem in general is knotty. For instance, programs often use functions like `memset` and `memcpy` to initialize and copy data structures. Such functions access all bytes in a structure sequentially, typically with a stride of one word. Clearly, we should not classify each access as a separate word-sized field. In fact, this is a serious problem for all, even the most advanced, approaches to date (e.g., DIVINE [4]). In our opinion, treating such functions in a special way (by blacklisting, say) is a bad solution, as it will handle the known functions, but not similar ones that are part of the application itself. Instead, as we shall see, DDE uses a heuristic that lets us dynamically select the "less common" layout. In other words, it favours data structures with different fields over an array of integers.

Code coverage As our analysis is performed dynamically, the accuracy increases if we execute more of the program's code paths. Code coverage techniques (using symbolic execution and constraint solving) force a program to execute all/most of its code [6]. We do not discuss this further in this paper. Recent work at EPFL explains how to do this for binaries [7].

3. Architecture

In this section, we sketch our solution, while addressing the above problems.

3.1 Function call stack

As a first step in the analysis, our technique keeps track of the function call stack. As DDE runs the program in an instrumented processor emulator, it can dynamically observe `call` and `ret` instructions, and the current position of the runtime stack. A complicating factor is that sometimes `call` is used not to invoke a real function, but rather only to push the return address. For instance, a `call` to a nearby instruction which immediately pops the return address lets a program determine the current value of EIP. Similarly, not every `ret` has a corresponding `call` instruction.

We define a *function* as the target of a `call` instruction which returns with a `ret` instruction. Values of the stack pointer at the time of the call and at the time of the return match, giving a simple criterion for detecting uncoupled `call` and `ret` instructions. Note that a function reached by means of a `jump` instruction is merged with the caller. We discuss impact of that on the analysis in Section 3.7.

3.2 Pointer tracking

We determine base pointers dynamically by tracking the way in which new pointers are derived from existing ones, and observing how the program dereferences them. In addition, we extract *root* pointers that are not derived from any other pointers. Root pointers initialize statically allocated memory, heap and stack.

For pointer tracking, we extended the processor emulator so that each memory location has a *tag*, `bp_memtag(addr)`, which stores its base pointer. In other words, a tag specifies how the address of a memory location was calculated. Likewise, if a general purpose register holds an address, an associated tag, `bp_regtag(reg)`, identifies its base pointer.

In the remainder of this section, we present tag propagation rules, while we explain how root pointers are determined in Section 3.3.

When a new root pointer `A` is encountered, we set `bp_memtag(A)` to a constant value `root` to mark that `A` has been accessed, but not derived from any other pointers. When a pointer `A` (root or not) is loaded from memory to a register `reg`, we set `bp_regtag(reg)` to `A`.

The program may now manipulate the pointer using pointer arithmetic (`add`, `sub`, or `and`). To simplify the explanation, we assume the common case, where pointers are manipulated completely *before* they are stored to memory, i.e., the intermediate results of pointer arithmetic operations are kept in registers only. This is not a limitation; it is easy to handle the case where a program stores the pointer to memory first, and then manipulates and uses it later.

During pointer arithmetic, we do *not* update the `bp_regtag(reg)`, but we do propagate the tag to destination registers. As an example, let us assume that after a number of arithmetic operations, the new value of `reg` is `B`. Only when the program dereferences `reg` or stores it to memory, do we associate `B` with its base pointer which is still kept in `bp_regtag(reg)`. In other words, we set `bp_memtag(B)` to `A`. This way we ensure that base pointers always indicate valid application pointers, and not intermediate results of pointer arithmetic operations.

3.3 Extracting root pointers

We distinguish between three types of root pointers: (a) those that point to statically allocated memory and are present in the ELF binary, (b) those that point to newly allocated dynamic memory, (c) the start of a function frame which serves as a pseudo root pointer for the local variables.

3.3.1 Dynamically allocated memory

To allocate memory at runtime, user code in Linux invokes either one of the memory allocation system calls (e.g., `mmap`, `mmap2`) directly, or it uses one of the `libc` memory allocation routines (e.g., `malloc`). Since each memory region is analyzed as a single entity, we need to retrieve their base addresses and sizes. DDE uses the emulator to intercept both. Intercepting the system calls is easy - we need only inspect the number of each syscall made. For `libc` routines, we determine the offsets of the relevant functions in the library, and interpose on the corresponding instructions pointers once the library is loaded.

3.3.2 Statically allocated memory

Root pointers to statically allocated memory appear in two parts of an object file: the data section which contains all variables initialized by the user - including pointers to statically allocated memory, and the code section - which contains instructions used to access these data. One solution to extract these pointers is scanning the memory area once the binary is loaded and marking all values that *could* be pointers. If we mistakenly mark a non-pointer value as a pointer, this is not a problem - as it will never be dereferenced, it will not serve in our later analysis. Our current implementation is different, and does not rely on guessing pointer values. To extract root pointers, we initially load pointers stored in well-defined places in a binary, e.g., `ELF` headers, or relocation tables, if present. Next, during execution, if an address `A` is dereferenced, `bp_memtag(A)` is not set, and `A` does not belong to the stack, we conclude that we have just encountered a new root pointer to statically allocated memory. Later, if we come across a better base pointer for `A` than `A` itself, `bp_memtag(A)` gets adjusted.

3.3.3 Stack memory

Function frames contain arguments, local variables, and possibly temporary data used in calculations. Typically, local variables are accessed via the function frame pointer, `EBP`, while the remaining regions are relative to the current stack position (`ESP`).

As we do not analyze temporary variables, we need to keep track of pointers rooted (directly or indirectly) at the beginning of a function frame only (often, but not always, indicated by EBP). Usually, when a new function is called, 8 bytes of the stack are used for the return address and the caller's `EBP`, so the callee's frame starts at `(ESP-8)`. However, other calling conventions are also possible. This means that we cannot determine where the function frame will start. To deal with this uncertainty, we overestimate the set of possible new base pointers, and mark all of them as possible roots. Thus, we emphasise that DDE does not rely on the actual usage of the `EBP` register. If, due to optimizations, `EBP` does not point to the beginning of the function frame, nothing bad happens.

About ESP Since we do not intend to analyze temporary local variables stored on the stack, we need a means to distinguish them from local variables and arguments belonging to functions. To make it explicit, we treat the stack pointer register, ESP, in a special way:

1. bp_regtag(ESP) is always equal to ESP;
2. we set bp_memtag(ESP) to a constant dummy value, which gets propagated on pointer dereferences relative to ESP.

This way addresses of temporary variables stored on the stack have the associated bp_memtags set to dummy, which lets us simply determine whether a memory location should be analyzed or not.

Notice that this also solves the problem of distinguishing memory regions allocated on stack by calls to functions like alloca. As we said before, existence and location of buffers allocated in such a way might depend on the control flow, and so we do not mean to include these regions in the analysis. Since these memory regions have base addresses derived from ESP, they are simply tagged as dummy.

Recall that once a new function is called, it often sets its own frame pointer to the current value of ESP. Observe that even though this frame pointer is derived from ESP, its associated bp_memtag is not equal to dummy. Indeed, we marked the anticipated beginnings of this function frame as root pointers. It means that local variables of the new function are correctly tracked.

Normally, assuming that a particular register fulfills a certain purpose is a bad idea, as due to optimizations registers can be used in "unpredicted" ways. However, ESP is a *sacred* register, whose value is required by the important push, pop, call, and ret instructions. Thus, in pratice, there is no reason why a program would ever use ESP for anything else than the current stack position, and we can therefore safely use these special rules in this case.

3.4 Multiple base pointers

As a memory location A is often accessed through multiple base pointers, we need to pick the most appropriate one. Intuitively, selecting the base pointer that is *closest* to the location, usually increases the *number of hops* to the root pointer, and so provides a more detailed description of a (nested) data structure.

However, functions like memset and memcpy often process composite data structures. These functions are completely unaware of the actual structure and simply access the memory in word-size strides. Thus, for 32 bit machines, such functions continuously calculate the next address to dereference by adding 4 to the previous one covering the entire data structure in 4 byte strides. By applying the aforementioned heuristic of choosing the closest base pointer, we could easily build a meaningless recursively nested data structure.

For structs the solution is often simple. When the program accesses the memory twice, once with constant stride equal to the word size (e.g., in memset) and once in a different manner (when the program accesses the individual fields), we should pick the latter. In arrays, however, multiple loops may access the array. To deal with this problem, we use a similar intuition and detect arrays and structures dynamically with a heuristic preference for non-regular accesses and/or accesses at strides not equal to the word size. For instance, if a program accesses a chunk of memory in two loops with strides 4, and 12, respectively, we will pick as base pointers those addresses that correspond to the latter loop. Intuitively, a stride of 12 is more likely to be specific to a data structure layout than the generic 4. Our current array detection introduces three kinds of loop accesses: (1) accesses with non-constant stride, e.g., an array of strings, (2) accesses with a constant stride not equal to the word-size, e.g., 1 or 12, and (3) accesses with stride equal to the word-size. Our heuristic, then, is as follows. First select the base pointers in the best possible category (lower is better), and next, if needed, pick the base pointer closest to the memory location.

It is worth mentioning that when selecting base pointers, we cannot allow loops. That is, for addresses A and B, we should never set both bp_memtag(A) to B, and bp_memtag(B) to A, as it would prevent us from understanding how these addresses were derived. Such a scenario might occur when the program uses the popular base_of macro. In detail, let us assume that B points to a field of a structure, f123, and bp_memtag(B) is already set to its base, A. Next, A is derived to hold this base, A = B-offset(f123), suggesting that B is the base pointer of A. If A is dereferenced now, we could end up with bp_memtag(A) set to B. To avoid this scenario, our heuristic additionally opts for base pointers located *below* the memory location, which solves the problem.

Note that the whole above strategy for picking base pointers is sound only if we assume that neither the programmer nor the compiler exploit the distance between two unrelated variables to access one of them as relative to the other. In other words, if A and B are addresses of two variables such that A is the base pointer of B, we presume that these *belong together*. For instance, they are elements of the same array, or represent the base of a structure and its field, respectively. If, on the contrary, we deal with distinct variables, these might get wrongly classified as belonging to one structure.

Even though the undesirable sequence of instructions *seems* unlikely in practice, we came across a scenario where it is realized, but also, in general, handled cor-

rectly by DDE. Namely, the `stpcpy` function[1] first calculates the distance, `d`, between source and destination buffers, `d=src-dst`, and next accesses the source buffer as relative to the destination buffer, `*(dst+d)`. Thus, when a pointer to the source buffer is dereferenced, the base pointer is set to `dst`, which is wrong. In practice, the source buffer is accessed not only in this way though. It is often used by more than one function, also with the proper base pointer. Consequently, our heuristic that picks the base pointer closest to the memory location yields the correct results.

3.5 Loop detection

As indicated above, DDE needs to detect memory accesses in loops. In this section we sketch our loop detection mechanism, while in Section 3.6 we discuss the algorithms for analyzing memory accesses in discovered loops.

Statically detecting loops in binaries is hard due to the presence of indirect `jump` and `call` instructions, which hinder static control flow graph extraction, and also static loop detection. Since we do not know the address to which to jump, the analysis might completely miss loop blocks, or a loop back edge, when that is realized with an indirect jump. Dynamically, all loops can be detected. However, this is expensive. In DDE, we decided to use a hybrid solution, where we apply static analysis for all easy cases - functions without indirect jumps - and dynamic analysis in all other situations. LESE [13] adopts a different approach to loop detection: the static control flow graph extraction is supplemented with indirect jump targets observed during tracing. However, in order to provide accurate results, LESE should be coupled with code coverage techniques.

Throughout loop detection, we assume that both head and tail of a loop's back edge belong to the same function. In other words, we require that a loop starts and ends in the same function. We have not encountered scenarios where that assumption would be a limitation, while it simplifies the loop detection mechanism. That boils down to analyzing control flow graphs representing single functions, and does not require examining callees (if any) along with callers.

Static analysis As we do not know function boundaries in binaries, even the static analysis is non-trivial. Since we can neither say where functions start, nor determine targets of indirect `jump` instructions, we also cannot foretell entry points to sequences of basic blocks. Consequently, we cannot predict which loop layout will be realized by the program - Figure 2 illustrates the problem.

[1] To be more precise, we mean the function in `strcpy.S` in `libc`, including `libc-2.9` and `libc-2.10`. Depending on the configuration, it it either defined as `stpcpy` or `strcpy`.

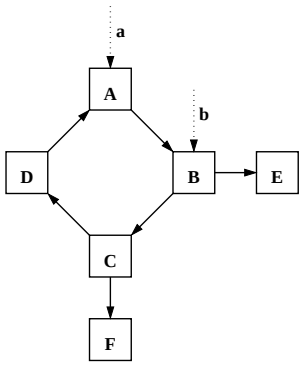

Figure 2. A simple control flow graph which is always reached with an indirect `jump` instruction. Squares represent basic blocks. Dashed arrows, `a` and `b`, denote possible indirect jumps, determinable at runtime only. Solid arrows indicate direct jumps, known during the static analysis. The graph illustrates that using solely static analysis, we cannot predict which loop layout will be adopted by the program. Indeed, if jump `a` is taken, then `A` should be assigned a loop head, and `D` $\rightarrow$ `A` the loop's back edge. Alternatively, if `b` is realized by the program, `B` and `A` $\rightarrow$ `B` should be considered, respectively. Finally, if both `a` and `b` are taken, we deal with an irreducible loop.

To deal with this lack of information, for each instruction `i` in the binary, we first determine set of basic blocks B_i reachable from `i` in a direct way, i.e., blocks containing indirect `jump` and `ret` instructions are not extended further. As mentioned already, we do not explore targets of encountered `call` instructions (even in the case of direct `call` s), but we continue adding basic blocks to B_i. Next, in each B_i we perform the standard loop detection analysis [1]. Finally, results are loaded to the program run in the processor emulator on demand. Only when the target of an indirect `jump` or a `call` (direct or indirect) instruction is determined at runtime, do we load information about loops reachable from the target instruction to the processor emulator.

We optimized the static analysis phase so that it scales to real world programs including the binaries of `libc` or the `Apache` web server.

Dynamic analysis Not all targets of `jump` and `call` instructions are analyzed statically. If the set of basic blocks reachable from the target contains indirect `jump` instructions, we switch the function containing this target to the dynamic loop detection mode.

The loop detection mechanism employs an auxiliary stack maintained by DDE, and used to monitor the sequence of functions and basic blocks executed by the program. In general, the stack is updated on each function call and return - both for direct and indirect `call` instructions - so that it reflects the original call stack in

the program. (We follow here the mechanism described in Section 3.1.) Additionally, it stores all basic blocks executed by functions containing indirect jumps and, as we will see later, just heads of loops for functions analyzed statically.

In the dynamic loop detection mode, when a basic block b_A is executed, we examine the part of the stack associated with the current function to check whether b_A is already on the stack. If not, it is pushed there. Otherwise, b_A is made the head of a loop L_A. Basic blocks stored on the stack and executed after b_A belong to L_A's body[2]. At this point we pop these basic blocks from the stack, and leave b_A on top. Note that if a basic block b_B from the body of L_A was designated head of a loop L_B, only then now is the right time to claim that loops L_A and L_B are nested. Indeed, when the head of one loop precedes the head of another loop on the stack, we cannot say whether these loops are subsequent or nested, as we always have to regard a back edge of the *deeper* loop as still probable to be taken. In general, this issue is a limitation of the dynamic loop detection method: when a certain path of a function control flow graph is executed for the first time, we cannot say whether the currently executing basic block belongs to a loop or not, until a back edge or the function `ret` instruction is encountered.

3.6 Array detection

As indicated above, DDE needs to detect memory accesses in loops. Specifically, we should detect how loops access arrays. This is quite hard and a general solution must handle the following difficult cases: (a) multiple loops placed in sequence, (b) nested sequences of loops, (c) inner loops and outer loops not iterating over the same array, and (d) first and last array element handling *outside* the loop.

In real code, there are two popular schemes for deriving array element addresses:

1. relative to the previous element, realized in instructions like `new=*(pprev++)`,
2. relative to the base of an array, `elem=array[i]`.

In this section we discuss the algorithms that handle both of these cases. In each of the two schemes, we start with the simpler case - when the array is accessed in the inner-most loop.

Before delving into the details, we discuss what kind of meta information is kept by DDE. The meta information is later used by the array discovery algorithms.

DDE identifies each loop with a loop id (`lid`) which it assigns to the loop head at runtime when the back edge is taken for the first time. At this point the loop head is pushed on the auxiliary stack. So, if a loop executes just once and never branches back for a second iteration, it does not get a new `lid`. We emphasize that `lid` is associated with a particular execution of a loop, and not only with the loop's head start address. It means that when an outer loop is executed multiple times, `lid`s assigned to an inner loop are different in each iteration of the outer loop.

[2] Additionally, by checking whether a loop has exactly one entry point, we detect irreducible loops. However, these appear rarely in practice, and we skip the discussion.

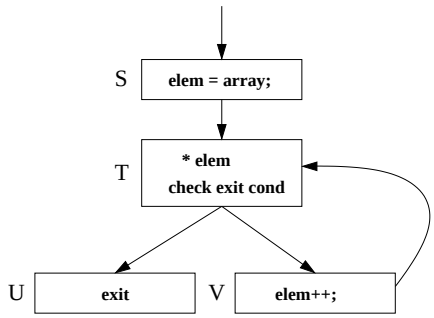

Figure 3. An example control flow graph representing a loop, which derives array element addresses relative to the previous array element. Basic block `T` is the loop head, and $V \rightarrow T$ the back edge.

DDE assigns the top `lid` as a tag to each byte of memory that the code accesses. These tags are again kept in a shadow memory available only to the emulator, similarly to `bp_memtag`. Note that memory accesses in the first iteration of a loop get the parent `lid`. By *parent* we mean the top-most loop head on the stack, it can represent both an outer loop and a sequentially preceding loop, which has already completed.

As an example, consider Figures 3 and 4. For simplicity's sake, the code snippet is in a high level language (pseudo `C`), but keep in mind that the whole analysis is performed in the processor emulator, on the level of assembly instructions. The first element of the array, `a`, is accessed before back edge $V \rightarrow T$ is taken, and thus gets the parent's `lid`: `(P,*)`, where `*` denotes an unknown parent's iteration number. Elements `b` . . . `f` are accessed in subsequent iterations of the loop, thus are tagged with loop L_T's `lid` coupled with the current iteration number: `(T, iter)`.

3.6.1 `new=*(pprev++)` in the inner-most loop

First, reconsider the simplified, yet representative, example of an array accessed in a loop, presented in Figures 3 and 4. The crux of this example is that addresses of array elements are derived relative to the previous element. The pointer to each array element is first dereferenced, and later increased to point to the next one. Thus base pointers hold addresses of previous elements (refer also to Figure 4). Intuitively, detecting an array accessed

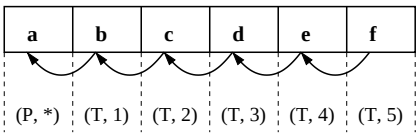

Figure 4. An example six-element array accessed by the loop in Figure 3. `a` . . . `f` are array elements. Arrows represent base pointers as indicated by the loop. The pairs below the array are assigned during the loop execution, and indicate the loop head, and iteration number, valid when a certain element was accessed. Rules for assigning these are further explained in the text. Loop head `P` stands for a *parent* `lid`, and `*` for an unknown iteration number.

in such way boils down to looking for chains of base pointers dereferenced in subsequent iterations of a loop. We first sketch DDE's mechanism, and later discuss a few design decisions.

Let's assume that again a pointer `B`, derived from base pointer `A`, is dereferenced, and that a loop with head `L` is currently being executed. Then DDE's core algorithm for detecting arrays is as follows:

1. if `B` was already accessed in loop `L`, do nothing,
2. if `B` is equal to `A`, do nothing; it means that `B` was handled already,
3. when `B` is dereferenced in iteration i, while `A` was dereferenced in a previous iteration of loop `L`, DDE treats `A` as a likely array element. It stores information about the array in the loop head kept in the auxiliary stack. The more iterations are executed, the more array elements DDE discovers.

Example. We can infer now that elements `b` . . . `e` belong to one array. Indeed, in the first loop iteration, `b` is dereferenced, but since `b`'s base pointer `a` is not tagged with $\texttt{L}_\texttt{T}$'s `lid`, we continue. In the second iteration, because `c`'s base pointer `b` was dereferenced in a previous iteration, we start building an array containing `b`. The information is stored in loop head `T`. Subsequent iterations let us similarly add `c`, `d`, and `e` to the array.

Because we often access the first and last elements in an array outside the loop, DDE explicitly checks to see if it can extend the array. In other words, since we cannot say which instructions before and after the loop body handle corner array elements, we try to extend the chain of base pointers in both directions as long as tags associated with the potential new values match the code *just before* or *just after* the loop. For an array detected in loop `L`, DDE first looks for earlier memory accesses at the base pointers used to recursively derive the first element of the array, and checks whether they have a `lid` that matches the last iteration of `L`'s parent. Note that this way we also add the elements accessed

```
int array[256], *elem = array;

while(condition1){
  access elem; elem++;
}

while(condition2) {
  access elem; elem++;
}
```

Figure 5. An example of an array accessed in two subsequent loops.

in the very first iteration of loop `L`. Second, DDE looks for a memory location based at the last element of the array, and checks whether associated `lid` matches the last iteration of `L`.

Example. After this step, we conclude that `a` and `f` belong to the array as well, giving us the complete data structure.

Remarks. We conclude this section with a few remarks on the array detection mechanism.

First, observe that by requiring the base pointer to be dereferenced in a previous iteration, we avoid treating lots of structure field accesses as potential array elements. Otherwise, when the base of an arbitrary structure and one of its fields were used in the same loop iteration, we would try to consider the base as an array element, which would be highly suboptimal.

To further avoid overestimating array elements, we attempt to extend an array with a base pointer (`A`) rather than the address currently dereferenced (`B`). To understand the reasoning behind this assumption, let's consider an array of structures and a loop which again calculates a pointer to the `(i+1)`$^{\text{th}}$ element in iteration `i`, and further stores it to memory. In iteration `(i+1)`, first a pointer `A` to the data structure is loaded from memory, its base pointer is set to `A` itself, and the associated loop tag indicates iteration `i`. Next, a pointer `B` to a field of the structure is calculated, and based on `A`. At this point we should not conclude that `B` is an array element, even though `A` belongs to a previous loop iteration.

Let's consider the case of multiple loops accessing consecutive parts of one array (refer to Figure 5), and see how we conclude that the loops retrieve a single data structure. Note that, in the common case (Figure 5), when loops are executed one after another, and there are no intermediate extra loops, both the last element accessed in the first loop and the first element accessed in the second loop are classified as belonging to both arrays detected. Indeed, when DDE tries to extend arrays, `lid`s of these two elements *match* both of them. As we shall see in Section 3.7.2, two interleaving arrays get merged in the final mapping stage, to give a single data structure. The other case, when there are unrelated loops in between is also often solvable - the final mapping phase requires a matching base pointer of the beginning of one

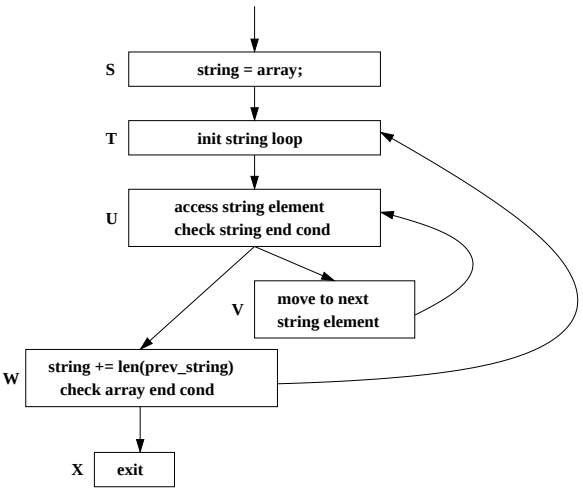

Figure 6. A control flow graph representing two nested loops iterating over an array of strings. The inner loop, with the head at `U`, and back edge `V` → `U`, accesses characters of a string. The outer loop, with the head at `T` and back edge `W` → `T`, determines addresses of subsequent strings. These pointers are calculated as relative to the previous array element, i.e., the previous string.

array and the last element of another array - but the detailed discussion is beyond the scope of this document.

Finally, observe that the array detection mechanism never relies on matching any strides, and so can be successfully applied to detect arrays having fields of varying sizes.

3.6.2 `new=*(pprev++)` in an outer loop

Detecting arrays accessed in an outer loop is very similar to the previous case (explored in Section 3.6.1). In fact, the only difference is that now we may need to choose carefully the loop head whose `lid` is used for comparisons, and which stores information about arrays detected. Previously it was simply the top loop head, while now we need to traverse the stack upwards and search for an appropriate one.

First, Figures 6 and 7 illustrate `lid`s assigned to an array of strings - the example contains nested loops iterating over a nested structure. We aim at concluding that `a0`, `b0`, `c0`, and `d0` are elements of one array. Observe that, in this case we do not need to implement any extra measures than the mechanism described in Section 3.6.1. Indeed, `a0`, `b0`, `c0`, and `d0` are accessed before the back edge of inner loop `U` is taken, and so the top loop head on the stack is still `T`. Thus the outer array elements get the same `lid`, and we know already how to solve this case.

Array detection becomes more complicated when there is an extra inner *sibling* loop in between `S` and

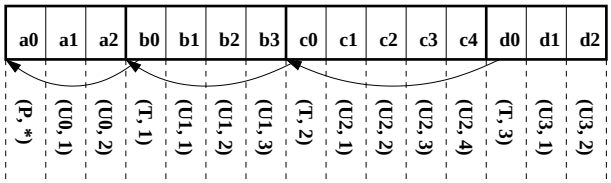

Figure 7. An example array of four strings accessed by the loop in Figure 6. The notation follows Figure 4. Loop heads `U1` . . . `U4` denote `lid`s of subsequent executions of the inner loop `U`. Base pointers determined by the inner loop, and describing how strings are accessed, are not marked down in the figure. They are irrelevant now.

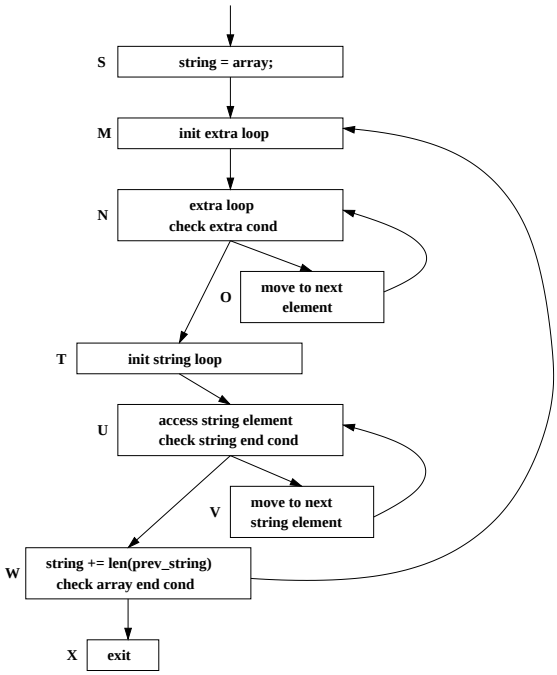

Figure 8. An extension of the control flow graph from Figure 6. It contains one extra inner loop consisting of basic blocks `M`, `N` and `O`.

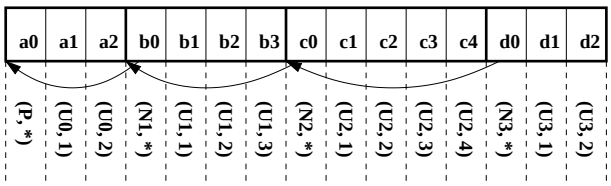

Figure 9. An example array of four strings accessed by the loop in Figure 8. The notation follows Figures 4 and 7.

T, i.e., before subsequent strings are accessed - refer to Figures 8 and 9. In this case elements a0, b0, c0, and d0 get different lids. We need to spot that these lids belong to different iterations of the outer loop with head M. Then, information about the outer array has to be stored at M, and not at the inner loop N, as the subsequent executions of the latter get flushed from the stack each time M's back edge is taken.

Remember that in Section 3.6.1, we explained that the criterion for adding array elements specifies that an address and its base share lid, and have different iteration numbers. To detect outer arrays, we allow an address and its base to represent various iterations of an outer loop.

The only question left now is how we determine the proper outer loop. Say that again B is dereferenced, and based on A. We want to search for a loop head such that A, and B belong to its different iterations. To deal with that, DDE associates with each loop head the current value of lid each time the head's back edge is taken, prev_lid. As lids form an increasing sequence of numbers, they introduce the notion of *time*. For a given loop head L, we can say that all memory locations tagged with lid higher or equal than L's lid and lower or equal than L's prev_lid were accessed in one of the previous and completed L's iterations. Because of that, as soon as we find on the stack a loop head L such that

- A's lid $\geq$ L's lid, and
- A's lid $\leq$ L's prev_lid,

we are done, and L is the loop head we are looking for. Otherwise we do not have reasons for trying to join A and B in one array.

Remarks. Since this algorithm is very similar to the one presented in Section 3.6.1, all remarks discussed there are also valid here. It is only worth emphasizing that the algorithm applied to multiple levels of nested loops still behaves correctly.

3.6.3 elem=array[i] in the inner-most loop

The second popular scheme for calculating array elements addresses in loops is relative to the base of an array, elem = array[i]. Figures 10 and 11 present a simplified example of such case. Observe that the array base is tagged with loop head T's parent's lid.

Let's assume that again pointer B, derived from base pointer A, is dereferenced, and that a loop with head L is currently being executed. Then DDE's core algorithm for detecting arrays is as follows:

1. if B was already accessed in loop L, do nothing,
2. if B is equal to A, do nothing; it means that B was handled already,

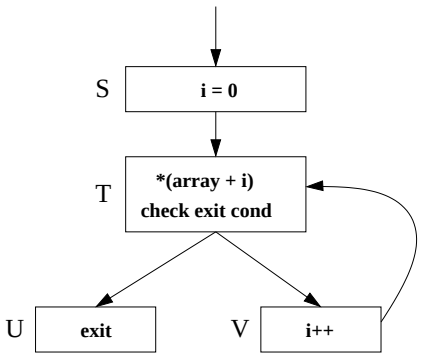

Figure 10. An example control flow graph representing a loop, which derives array elements addresses as relative to the array base. Basic block T is the loop head, and V $\rightarrow$ T the back edge.

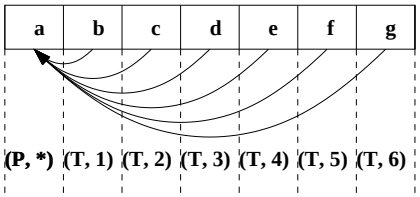

Figure 11. An example array accessed by the loop in Figure 10. The notation follows Figure 4.

3. if A is tagged either with L's parent's lid in its most recent iteration or L's lid, and if A is marked as a *probable* base (check the next bullet for that), try to add B to an array with base A; keep all results in the stack element associated with the loop head L,
4. if A is tagged with L's parent's lid in its most recent iteration, mark it as a *probable* array base.

When an array has been built over a few iterations already, we try to determine its *stride*. In the case of a regular access to an array of integers, we see a sequence $\{4, 8, 12, 16\}$ of offsets associated in the subsequent iterations with the array base. Here, the stride of 4 is an obvious answer. However, let us now assume that we deal with an array of structures of size 12, where each structure has three 4-byte fields, and some of them are accessed relative to the array base[3]. In this case we may face a following sequence of offsets: $\{16, 20, 24, 28, 36\}$. Here we apply a heuristic picking the *best fitting* stride.

[3] Keep in mind that structure fields can be accessed in two basic ways: (1) relative to the structure base: a_struct_pointer->field, or (2) relative to the array base: often array[i].field is compiled to *(array+i*size+offset(field)).

Finally, we again explicitly check for extending the array (this phase is similar to the other array access scheme).

Remarks. We conclude the section with a few remarks on the algorithm.

We introduced the notion of the *probable* array base (rule 4.) to avoid treating structure accesses as possible arrays. Now, the only possibility to trick DDE to start considering a structure as an array is as follows: the base of the structure is accessed in the first iteration of a loop, fields `fdA` and `fdB` are accessed only later, and additionally in different iterations. Moreover, the heuristic which decides on array stride would need a number of *sensible* offsets. However, a structure with four 4-byte fields, `f1`, `f2`, `f3`, and `f4`, which get accessed for the first time in subsequent loop iterations will be classified as an array of integers, and there is nothing we could do about it.

Observe that we currently require that the base of an array is accessed in the loop before the back edge is taken (or just before the loop – so that it is tagged with the parent's `lid`). It is very often the case in practice that an array is iterated over from the first element, and so for all our experiments we obtained accurate results with the current implementation. However, if necessary we should make a distinction between two facts: "`A` was used as a base in a loop with `lid L`" and "`A` was accessed in a loop with `lid L`", which would solve the problem. It would also significantly increase the amount of memory used by the analysis though, as we would have to keep an additional set of tags in shadow memory in the processor emulator.

It is worth mentioning that our heuristic calculating stride from a set of offsets does not require input from consecutive iterations. For instance, if a loop accesses odd array elements in odd iterations only, the resulting stride is the distance between these two odd elements.

Finally, let's discuss the issue of multiple loops accessing consecutive parts of one array (refer to Figure 12). Here the problem is more complex than in Section 3.6.1. Indeed, we do not track how index `i` is calculated, and so we do not have means of stating that the second loop is a continuation of the first one. DDE can often ascertain the opposite situation though - if a structure contains two arrays, located next to each other in memory, DDE can find out that the second array was always accessed with a constant and immediate offset from the base of the structure, as opposite to index `i` being a result of arithmetic operations. Consequently, such arrays are separate, and should not be merged. That information helps deciding on the actual data structure - but detailed discussion is beyond the scope of this document.

```
int array[256], i = 0;

while(condition1){
  access array[i]; i++;
}

while(condition2) {
  access array[i]; i++;
}
```

Figure 12. An example of an array accessed in two subsequent loops.

3.6.4 `elem=array[i]` in an outer loop

Problems posed by this case are very similar to the ones presented in Section 3.6.2. Again, if elements of an outer array are accessed in the first inner loop, the detection mechanism boils down to the inner-most case (Section 3.6.3). Otherwise, the solution is analogical to Section 3.6.2.

3.7 Final mapping

Having detected arrays and the most appropriate base pointers, DDE finally maps the analyzed memory into meaningful data structures.

First, we briefly discuss when during the program runtime the final mapping takes places (Section 3.7.1), and afterwards we sketch the basic mapping mechanism (Section 3.7.2).

3.7.1 Time of mapping

For a memory chunk, the mapping starts at a root pointer and reaches up to the most distant memory location still based (directly or indirectly) at this root. In the case of the stack, we extend the region for analysis in both directions: the part below the beginning of a function frame contains local variables, and the part above - function arguments (if any).

For static memory, the mapping is performed at the end of the program execution. Memory allocated with `malloc` is mapped when it is released using `free`, while local variables and function arguments on the stack are mapped when a function returns.

As we mentioned in Section 3.1, if a function is reached using a `jump`, and not a `call` instruction, we do not notice that a new function has been invoked. Therefore we do not determine possible beginnings of its function frame, nor can we mark its `root`. Indeed, we do not have the means of analyzing its local variables. More importantly, we need to avoid merging the callee's and caller's function frames, so that we do not map the whole region as if it represented local variables of a single function. Observe however, that this issue does not require any additional measures. When the callee builds its new function frame, its start address is relative to `ESP`, and tagged as `dummy` (by setting `bp_memtag`). Con-

sequently, the entire callee's frame has `bp_memtag`'s set to `dummy`, and thus we do not even try to map it.

3.7.2 Mechanism

Mapping a memory region without arrays is straightforward. Essentially, memory locations which share a base pointer form fields of a data structure rooted at this pointer, and on the stack, memory locations rooted at the beginning of a function frame represent local variables and function arguments.

When a potential array is detected, we check if it matches the data structure pattern derived from the base pointers. If not, the array hypothesis is discarded. For example, if base pointers hint at a structure with variable length fields, while the presumed array has fields of 4B, DDE assumes the accesses are due to functions like `memset`.

The analysis may find multiple interleaving arrays. (each corresponding to its own data structure access pattern indicated by base pointers). If such arrays are not included in one another, we merge them. Otherwise, we examine the base pointers further to see if the arrays are nested.

DDE has also enough information to speculate on the actual type of data stored in data structures. For instance, we can distinguish pointers, strings or integers. We leave further research in that area as a future work.

4. Results

DDE can analyse any application that runs on the Linux guest. To verify its accuracy, we compare the results to the actual data structures in the programs. One approach we are considering is to do so automatically by compiling binaries with debugging symbols and then comparing the DDE output to the data structures obtained from the debug symbol table. However, this works only up to a point, as aggressive compiler optimisations may change the binary significantly ("what you see is not what you execute" [2]). Moreover, our current parser for the debug symbols is incomplete (e.g., it does not handle heap structures) and we have not yet coupled DDE to code coverage techniques. Both are needed to run such verification tests automatically. For now, therefore, our evaluation is purely manual.

To verify the accuracy of the analysis, we manually inspect the source *and binary* of several Linux programs and see whether the data structures correspond to those extracted by DDE. We applied manual inspection to a host of handwritten programs as well as to the UNIX `fortune` program. All data structures are handled well, with the exception of the first two cases listed below.

We also analyse DDE's performance for other, more complex binaries, but for these we do not manually inspect the full program. Rather, we sample a number of functions, with a focus on the complex functions. These binaries include the Linux loader `ld-2.9.so`, the `Apache` web server, `wget`, `glines`, and `gnometris`.

Limitations. DDE is not flawless. While evaluating it with `fortune`, `ld.so` and the other binaries we identified the following limitations.

1. DDE cannot recognise nested structs if the inner struct is never accessed separately. In that case, DDE just returns a single large structure. As the result is equivalent, we do not consider this a problem.
2. Without code coverage, we may misclassify array accesses using `strncpy` and similar functions. For instance, `fortune` has a buffer of 512 bytes containing a string of n bytes that is copied using `strncpy`. n bytes are accessed one byte at a time, while all remaining bytes 4 bytes at a time (to zero the buffer). Thus, DDE classifies it as two arrays. Coupling DDE to code coverage should mostly solve it. We are not too concerned about this, as coupling DDE to code coverage should mostly solve it.
3. DDE does not detect arrays if a loop is executed once or twice (in that case it is classified as a structure).
4. The current implementation does not take all alignment issues into account. For instance, if a one byte field takes up 4 bytes in memory, DDE will classify it as a 4 byte field. Fixing this is possible, but requires inspecting the contents of memory accesses.
5. DDE cannot classify fields that the program never accesses. Again, code coverage will help here.
6. Highly irregular array accesses (e.g., `array[rand()]`) will not be recognised as arrays.
7. DDE cannot currently deal with custom memory allocators. If the program allocates a pool which serves various data structures, and is reused in the runtime, DDE does not handle that correctly. DDE does not detect the custom allocation routine, and thus it gets confused with the interleaving data structures.
8. DDE does not analyze local variables of functions reached using a `jump`, and not a `call` instruction.

Note that even if DDE cannot classify an array or structure correctly in one particular loop or function, it may still get it right eventually. Often data structures are accessed in more than one function, yielding multiple loops to analyse the layout.

5. Related work

Most existing approaches to decompilation build on static analysis of binaries. The most advanced techniques in this field include value set analysis (VSA) [3], and aggregate data structure inspection (ASI) [12], and combinations of VSA and ASI [4]. Unfortunately, none

of these static techniques can handle some of the most common data structures, like arrays, properly. Nor can they handle other common programming cases. For instance, if a C struct is copied using a function like memcpy, it will be misclassified as having many fields of 4 bytes, because the stride in `memcpy` on 32 bit machine is 4). Similarly, they cannot deal with functions like 'alloca'. Finally, the combination of VSA and ASI in [4] is context-sensitive, which leads to state space explosion. The reported results show that even the most trivial programs take an exceedingly long time to analyse.

In contrast, our analysis is based on dynamic analysis. We use static analysis only as a simple optimisation to detect loops in functions without indirect jumps.

Laika [8] uses dynamic analysis for data structure recovery. It employs Bayesian unsupervised learning to detect data structures. However, its detection is very imprecise and limits itself to aggregates. For instance, it may observe chunks of bytes in what looks like a list, but it does not know about fields in structures. For debugging and reverse engineering, this is wholly insufficient. The authors are aware of this and use Laika only to estimate, in an approximate manner, the similarity of viruses.

Some projects are specifally related to DDE's loop detection method. LoopProf [11] also uses dynamic analysis to detect loops. However, it is much weaker and cannot detect nested loops.

6. Conclusions

We have described a new technique, known as DDE, for extracting data structures from binaries dynamically without access to source code or symbol tables, by observing how program access memory during execution. As until now data structure extraction for C binaries was not possible, we expect DDE to be valuable for the fields of debugging, reverse engineering, and security.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition*. Pearson Education, Addison Wesley, 2007.

[2] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. Wysinwyx: What you see is not what you execute. In *IN VSTTE*, page 1603, 2005.

[3] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 binary executables. In *Proc. Conf. on Compiler Construction (CC)*, April 2004.

[4] Gogul Balakrishnan and Thomas Reps. DIVINE: Discovering variables in executables. In *Proc. Conf. on Verification Model Checking and Abstract Interpretation (VMCAI)*, January 2007.

[5] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proc. of the USENIX Annual Technical Conference*, 2005.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, 2008.

[7] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with revnic. In *Proceedings of EUROSYS*, Paris, France, April 2010.

[8] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *OSDI*, pages 255–266, San Diego, CA, December 2008.

[9] DataRescue. High level constructs width IDA Pro. `http://www.hex-rays.com/idapro/datastruct/datastruct.pdf`, 2005.

[10] Mike Van Emmerik and Trent Waddington. Using a decompiler for real-world source recovery. *Reverse Engineering, Working Conference on*, 0:27–36, 2004.

[11] Tipp Moseley, Vasanth Tovinkere, Ram Ramanujan, Daniel A. Connors, and Dirk Grunwald. Loopprof: Dynamic techniques for loop detection and profiling. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

[12] 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*, 1999.

[13] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009.