Howard: a dynamic excavator for reverse engineering data structures

Asia Slowinska Traian Stancescu Herbert Bos
Vrije Universiteit Amsterdam, The Netherlands
{asia,tsu500,herbertb}@few.vu.nl

Abstract

Even the most advanced reverse engineering techniques and products are weak in recovering data structures in stripped binaries—binaries without symbol tables. Unfortunately, forensics and reverse engineering without data structures is exceedingly hard. We present a new solution, known as Howard*, to extract data structures from C binaries without any need for symbol tables. Our results are significantly more accurate than those of previous methods — sufficiently so to allow us to generate our own (partial) symbol tables without access to source code. Thus, debugging such binaries becomes feasible and reverse engineering becomes simpler. Also, we show that we can protect existing binaries from popular memory corruption attacks, without access to source code. Unlike most existing tools, our system uses dynamic analysis (on a QEMU-based emulator) and detects data structures by tracking how a program uses memory.*

1. Introduction

State of the art disassemblers are indispensable for reverse engineering and forensics. The most advanced ones, like IDA Pro [24] and OllyDbg [2], offer a variety of techniques to help elevate low-level assembly instructions to higher level code. For instance, they recognize known library functions in the binary and translate all calls to these functions to the corresponding symbolic names in the presentation to the user. Some are sufficiently powerful to handle even binaries that are statically linked and 'stripped' so that they do not contain a symbol table.

However, they are typically weak in reverse engineering data structures. Since real programs tend to revolve around their data structures, ignorance of these structures makes the already complex task of reverse engineering even slower and more painful.

The research community has been aware of the importance of data structures in reverse engineering for several years now, but even so, no adequate solution emerged. The most common approaches are based on static analysis techniques like value set analysis [8], aggregate structure identification [38] and combinations thereof [39]. Some, like CodeSurfer/x86, are available as experimental plug-ins for IDA Pro [24]. Unfortunately, the power of static analysis is quite limited and none of the techniques mentioned above can adequately handle even some of the most common data structures – like arrays.

Some recent projects have therefore resorted to dynamic analysis. Again, success has been limited. The best known examples are Laika [22] and Rewards [31]. Laika's detection is both imprecise and limited to aggregates structures (i.e., it lumps together all fields in a structure). This is not a problem for Laika's application domain – estimating the similarity of different samples of malware by looking at the approximate similarity of their data structures. However, for forensics and reverse engineering this is wholly inefficient.

Rewards [31], presented in last year's NDSS, builds on a technique originally pioneered by Ramilingam et al. in aggregate structure identification (ASI) [38]. The idea is simple: whenever the program makes a call to a well-known function (like a system call), we know the types of all the arguments – so we label these memory locations accordingly. Next, we propagate this type information backwards and forwards through the execution of the program. For instance, whenever labeled data is copied, the label is also assigned to the destination. Rewards differs from ASI in that it applies this technique to dynamic rather than static analysis.

Either way, by definition the technique only recovers those data structures that appear, directly or indirectly, in the arguments of system calls (or the well-known library functions). This is only a very small portion of all data structures in a program. All internal variables and data structures in the program remain invisible.

In this paper, we describe a new technique known as

Howard[1] that greatly improves on these existing techniques. It is complementary to Rewards, but much more powerful as it also finds internal variables. Like Rewards and Laika, Howard is based on dynamic analysis

The main goal of Howard is to furnish existing disassemblers and debuggers with information about data structures and types to ease reverse engineering. For this purpose, it automatically generates debug symbols that can be used by all common tools. We will demonstrate this with a real analysis example using `gdb`. This is our first application.

In addition, however, we show that the data structures allow us to retrofit security onto existing binaries. Specifically, we show that we can protect legacy binaries against buffer overflows. This is our second application.

High-level overview. Precise data structure recovery is difficult, because the compiler translates all explicitly structured data in the source to chunks of anonymous bytes in the binary. Data structure recovery is the art of mapping them back into meaningful data structures. To our knowledge, no existing work can do this. The problem becomes even more complicated in the face of common compiler optimizations (like loop unrolling, inlining, and elimination of dead code and unused variables) which radically transform the binary.

Howard builds on dynamic rather than static analysis, following the simple intuition that memory access patterns reveal much about the layout of the data structures. Something is a structure, if it is accessed like a structure, and an array, if it is accessed like an array. And so on.

Like all dynamic analysis, Howard's results depend on the code that is covered at runtime – it will not find data structures in code that never executes. This paper is not about code coverage techniques. Rather, as shown in Figure 1, we use existing code coverage tools (like KLEE) and test suites to cover as much of the application as possible, and then execute the application to extract the data structures.

In summary, Howard is able to recover most data structures in arbitrary (gcc-generated) binaries with a high degree of precision. While it is too early to claim that the problem of data structure identification is solved, Howard 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.

We implemented all dynamic analysis techniques in an instrumented processor emulator based on Qemu [10]. Since single process emulation is available only for Linux, the implementation is also for Linux. However, the approach is not specific to any particular OS.

[1]The system was named after Howard Carter, the archaeologist who excavated the tomb of Tutankhamun.

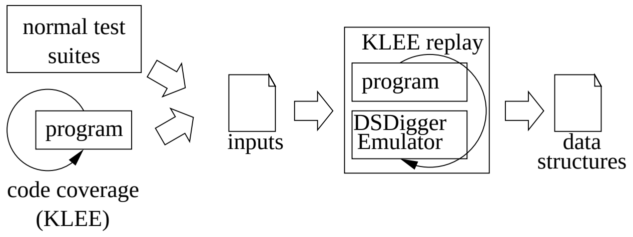


Figure 1. The main stages in Howard.

Outline The remainder of this paper is organized as follows:

2.	Related work	6.	Limitations
3.	Challenges	7.	Code transformation
4.	Design and implementation	8.	Evaluation
5.	Applications	9.	Conclusions

2 Existing work and what we took from it

Recovery of data structures is most obviously relevant to the fields of debugging and reverse engineering. Still, even the most advanced tools (like IDA Pro [24], and CodeSurfer [8]), are weak at identifying data structures. The limited support for data structure recovery they provide comes exclusively in the form of static analysis. These techniques are inadequate, as we shall see below. As far as we know, Howard is very different from any existing approach. Nevertheless, we have been influenced by certain projects. In this section, we summarize them and their relation to Howard.

Static analysis. A first stab at recovering data without executing the program is to try to identify all locations that look like variables and estimate the sets of values that they may hold [8]. Known as Value Set Analysis (VSA), this approach uses abstract interpretation to find (over-approximations of) the sets of possible values.

Of course, accurately pinpointing the locations that hold variables in a program is not simple, but studying the way in which the program accesses memory helps. Incidentally, this idea originates in efforts to deal with the Y2K problem in old COBOL programs and is known as Abstract Structure Identification (ASI) [38]. Translated into C terms, ASI attempts to partition memory chunks statically in `structs` of arrays and variables, depending on accesses. For instance, if a stack frame holds 40 bytes for local variables, and the program reads the 4 bytes at offset 8 in the range, ASI classifies the 40 bytes as a struct with one 4-byte variable wedged between 2 arrays. As more addresses are referenced, ASI eventually obtains an approximate mapping of variable-like locations.

ASI has another clever trick to identify data structures and types, and that is to use the type information from system calls and well-known library functions. As the argument types of these calls are known, at every such call, ASI tags the arguments with the corresponding types and propagates these tags through the (static) analysis.

The culmination of these static techniques is a combination of VSA and ASI by Balakrishnan et al. [39]. This powerful static analysis method is also available as an experimental plug-in for IDA Pro, known as CodeSurfer/x86 [7].

At this point, however, we have to mention that all of these static techniques have problems handling even the most basic aggregate data structures, like arrays. Nor can they handle some common programming cases. For instance, if a C `struct` is copied using a function like `memcpy`, VSA/ASI will misclassify it as having many fields of 4 bytes, simply because `memcpy` accesses the memory with a stride of 4 (on a 32 bit machine). Also, they cannot deal with functions like '`alloca`'. In contrast, Howard does not depend on static analysis at all.

In a more constrained setting, Christodorescu et al. show how static analysis of x86 executables can help recover string values [18]. Their technique detects C-style strings modified by the `libc` string functions.

Dynamic analysis. Eschewing static analysis, Laika recovers data structures during execution in a novel way [22]. First, Laika identifies potential pointers in the memory dump –based on whether the contents of 4 byte words look like a valid pointer– and then uses them to estimate object positions and sizes. Initially, it assumes an object to start at the address pointed to and to end at the next object in memory. It then converts the objects from raw bytes to sequences of block types (e.g., a value that points into the heap is probably a pointer, a null terminated sequence of ASCII characters is probably a string, and so on). Finally, it detects similar objects by clustering objects with similar sequences of block types. In this way, Laika detects lists and other abstract data types.

On the other hand, Laika's detection is both imprecise and limited to aggregates. For instance, it may observe chunks of bytes in what looks like a list, but it does not detect the fields in the structures. For debugging, reverse engineering, and protection against overflows, this is wholly insufficient. The authors are aware of this and use Laika instead to estimate the similarity of malware.

Originally, Howard borrowed from Laika the idea of dynamically classifying blocks of null-terminated printable characters as "probable string blocks" to improve the speed of string detection. However, as we improved our default string detection method, the additional accuracy provided by the Laika method was very small and we therefore removed it.

Rewards [31] builds on the part of ASI that propagates type information from known parameter types (of system calls and library functions). Unlike ASI, however, it does so dynamically, during program execution. All data structures that are used in, or derived from, system calls or known templates are correctly identified. However, Rewards is (fundamentally) not capable of detecting data structures that are internal to the program.

Howard emphatically does not need any known type to recover data structures, but whenever such information is available, it takes advantage of it to recover *semantics*. For instance, it may help to recognize a structure as a `sock_addr` structure, a file descriptor, or an IP address.

One of the more complicated features of Howard is its loop detector, which we use to detect array accesses. LoopProf [33] also detects loops dynamically. However, it is weaker than Howard and cannot detect nested loops.

Dynamic protocol format reverse engineering. Different in nature, but still related is the problem of automatic protocol reverse engineering. Systems like Polyglot [12], [44], AutoFormat [30], Tupni [23], ReFormat [43], and Prospex [20], aim to analyze how applications parse and handle messages to understand a protocol's format. They typically do so dynamically, although some are supplemented by static analysis. While different in various regards, these systems track data coming from the network and by observing the applications' behavior try to detect constant protocol fields, length fields, delimiters, and so on. The most advanced ones also cluster different types of messages to recover the protocol's state machine. One of Howard's array detection methods was influenced by that of Polyglot [12]. Still, it is considerably more advanced. In addition, we add a second, completely new array detection technique.

Security. The security application in Section 5 is inspired by WIT [4] and similar approaches [5, 45]. WIT is a compiler extension that analyzes C source code to detect which instructions can write which objects. It then generates instrumented object code to prevent instructions from writing to different objects, thus eliminating memory error exploits. With Howard, we do something similar, except that we do not require the source code. Thus, we can apply our technique to legacy binaries (at the cost of some additional overhead).

3 Recovery by access patterns: challenges

Howard aims to answer questions like: "What are the variables, `structs`, and arrays in the program and what are their layouts?" As explained in Section 2, for a subset of these variables (for which we have type sinks), we are also able to recover semantics, so that we can answer questions like: "Is the 4-byte field in this `struct` an IP address, a pointer, or an integer?"

Howard 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 the 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. As we shall see, distinguishing between these three scenarios is one of the challenges we need to address.

It is not the only issue. In the remainder of this section, we discuss the main obstacles that we had to remove to make Howard possible. Some are relatively straightforward and for these we discuss immediately how we solved them. Others are not, and we postpone their solution to Section 4, where we discuss our approach in full.

Even though we discuss many details, due to space limitations, we are not able to discuss *everything* in great depth. We realize that some readers are interested in all the details, and for this reason we made available a technical report that contains enough information to allow one to reproduce our system [42]. From time to time, we will refer readers interested in details to the report.

Memory allocation context Our work analyzes a program's use of memory, which includes local function variables allocated on the stack, memory allocated on the heap, and static variables. Static memory is not reused, so it can be uniquely identified with just its address. However, both the runtime stack and heap are reused constantly, and so a description of their data structures 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. While Howard handles these cases correctly, the details are tedious and beyond the scope of this paper (see [42]). For now, it suffices to think of function start addresses as the frame identifiers.

For heap memory, such simple pointers will not do. Consider a `my_malloc` wrapper function which invokes `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` with `my_malloc` itself, but rather with its caller. As we do not know the number of such `malloc` wrappers in advance, we associate heap memory with a call *stack*. We discuss call stacks in detail in Section 4.1.

Pointer identification To analyze memory access patterns, we need to identify pointers in the running program. Moreover, for a given address `B=A+4`, we need to know `A`, the *base* pointer from which `B` was derived (e.g., to find nested structures). 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. Therefore, Howard must track how new pointers are derived from existing ones. We discuss our solution in Section 4.2.

Missing base pointers As mentioned earlier, Howard detects new structure fields when they are referenced from the structure base. However, programs sometimes use fields without reference to a base pointer, resulting in misclassifications. Figure 2 illustrates the problem. 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. Without it, we would characterize this memory region as composed of 3 separate variables: `pelem`, `x`, and `y` (but since the program here does not actually use the connection between the fields `x` and `y`, this partially inaccurate result would be innocuous). A missing base pointer is of course a *fundamental* limitation, as we cannot recognize what is not there. In practice, however, it does not cause many problems (see also Section 8).

Multiple base pointers Conversely, memory locations can be accessed through *multiple* base pointers, which means that we need to decide on the most appropriate one. Observe that field `elem.y` from Figure 2 is already referred to using two different base pointers, the frame pointer `EBP` and `pelem` (`EAX`). While this particular case is 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. This is a serious problem for all approaches to date, even the most advanced ones [39].

```
typedef struct {      <fun>:

 int x;              [1] push %ebp
 int y;              [2] mov %esp, %ebp
} elem_t;            [3] sub $0x10, %esp
                     [4] mov $0x1,
                     -0xc(%ebp)
void fun() {         [5] mov $0x2,
                     -0x8(%ebp)
 elem_t elem,        [6] mov -0x4(%ebp),
*pelem;              %eax
 elem.x = 1;         [7] mov $0x3,
                     0x4(%eax)
 elem.y = 2;         [8] leave
 pelem = &elem;      [9] ret
 pelem->y = 3;
}
```

Figure 2. 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.

One (bad) way to handle such functions is to blacklist them, so their accesses do not count in the analysis. The problem with blacklisting is that it can only cope with known functions, but not with similar ones that are part of the application itself. Instead, we will see that Howard uses a heuristic that selects the "less common" layout. For instance, it favors data structures with different fields over an array of integers.

Code coverage As Howard uses dynamic analysis, its accuracy increases if we execute more of the program's code. Code coverage techniques (using symbolic execution and constraint solving) force a program to execute most of its code. For Howard, the problem is actually easier, as we do not need all code paths, as long as we see all data structures. Thus, it is often sufficient to execute a function once, without any need to execute it in all possible contexts. In our work, we use KLEE [13]. Recent work at EPFL (kindly provided to us) allows it to be used on binaries [15]. Figure 1 illustrates the big picture. In reality, of course, KLEE is not perfect, and there are applications where coverage is poor. For those applications, we can sometimes use existing test suites.

4 Howard design and implementation

We now discuss the excavation procedure in detail. In the process, we solve the remaining issues of Section 3.

4.1 Function call stack

As a first step in the analysis, Howard keeps track of the function call stack. As Howard runs the program in a 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 only as part of a `call/pop` sequence to read the value of the instruction pointer. 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[2].

Whenever we see a function call, we push this information to a Howard internal stack, which we refer to as DSD-Stack.

4.2 Pointer tracking

Howard identifies 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. Howard identifies different root pointers for statically allocated memory (globals and static variables in C functions), heap and stack.

For pointer tracking, we extended the processor emulator so that each memory location has a *tag*, MBase(`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, RBase(`reg`), identifies its base pointer.

We first present tag propagation rules, and only afterward explain how root pointers are determined.

When Howard encounters a new root pointer A, it sets MBase(A) to a constant value `root` to mark that A has been accessed, but does not derive from any other pointer. When a pointer A (root or not) is loaded from memory to a register `reg`, we set RBase(`reg`) to A.

The program may manipulate the pointer using pointer arithmetic (`add`, `sub`, or `and`). To simplify the explanation, we assume the common case, where the program manipulates pointers completely *before* it stores them to memory, i.e., it keeps the intermediate results of pointer arithmetic operations in registers. 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 RBase(`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

[2]In rare cases, functions are reached by a jump. Howard merges these functions with the caller. We discuss the impact on the analysis in [42].

kept in RBase(`reg`). In other words, we set MBase(B) to A. This way we ensure that base pointers always indicate valid application pointers, and not intermediate results of pointer arithmetic operations.

Extracting root pointers We distinguish between 3 types of root pointers: (a) those that point to statically allocated memory, (b) those that point to newly allocated dynamic memory, and (c) the start of a function frame which serves as a pseudo root for the local variables.

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 Howard analyzes each memory region as a single entity, we need to retrieve their base addresses and sizes. Howard uses the emulator to intercept both. Intercepting the system calls is easy - we need only inspect the number of each call made. For `libc` routines, we determine the offsets of the relevant functions in the library, and interpose on the corresponding instructions once the library is loaded.

Statically allocated memory. Statically allocated memory includes both static variables in C functions and the program's global variables. 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 the data. 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, MBase(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, MBase(A) gets adjusted.

Stack memory. Function frames contain arguments, local variables, and possibly intermediate 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 intermediate results on the stack, 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 [42]. 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, Howard does not rely on

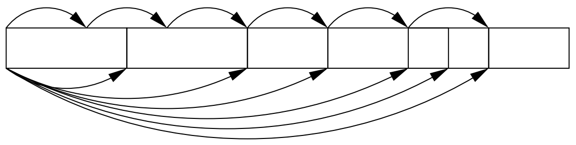


Figure 3. Example of a memory area accessed using multiple base pointers. The arrows on top, illustrate a function like `memset` that accesses all fields with a stride of 4 bytes, while the 'real' access patterns, below, show accesses to the individual fields.

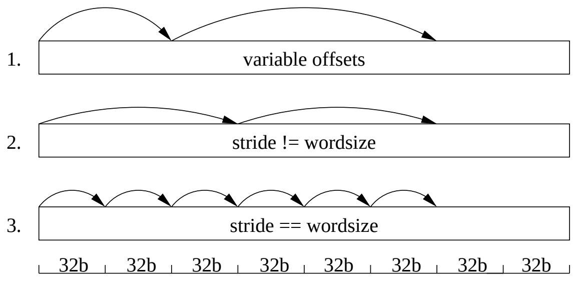


Figure 4. Different memory access patterns. When Howard observes different access patterns to the same object, it prefers pattern 1 over patterns 2 and 3, and 2 over 3.

the actual usage of the EBP register. If, due to optimizations, EBP does not point to the beginning of the frame, nothing bad happens.

4.3 Multiple base pointers

As a program often accesses a memory location A 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, as shown in Figure 3, functions like `memset` and `memcpy` often process composite data structures. These functions are completely unaware of the actual structure and 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 categories of loop accesses (see Figure 4): (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. Next, we discuss arrays and loops in detail.

4.4 Array detection

Array detection is both difficult and important – especially for security. Howard recovers arrays when the program accesses them in loops. Fortunately, this is true for the vast majority of arrays. In the simplest case, the program would access an array in an inner loop, with one array element in each loop iteration. However, a general solution must also handle the following scenarios: (a) multiple loops accessing the same array in sequence, (b) multiple nested loops accessing the same array, (c) loop unrolling, resulting in multiple array elements accessed in one loop iteration, (d) inner loops and outer loops not iterating over the same array, and (e) boundary array elements handled *outside* the loop. Howard uses several complementary array detection methods to deal with all these cases. We divide these methods in two major classes depending on the way array elements are accessed by the program.

Loops in real code implement one of two generic schemes for deriving array element addresses: (1) relative to the previous element, realized in instructions like `elem=*(prev++)`, and (2) relative to the base of an array, `elem=array[i]`. As we shall see, Howard handles both cases. We discuss limitations of our array detection method in Section 6.

4.4.1 Accesses relative to previous elements

To handle loop accesses in a buffer where each element address is relative to a previous one, Howard is set up to track *chained* sequences of memory locations. For instance, if `elem = *(pprev++)`, then the pointer to `elem` is derived from `pprev`. A few of these scenarios are illustrated in Figure 4.

No loop unrolling. We explain what happens for the non-optimized case first and worry about loop-unrolling later. Howard identifies each loop with a timestamp-like id (`lid`) which it assigns to the loop head at runtime when the back edge is taken for the first time. See Figure 5. At this point this loop head is pushed on DSDStack. So, if a loop executes just once and never branches back for a second iteration, it does not get a new `lid`. Howard assigns the top `lid` as a tag to each memory location `A` the code accesses: `MLid (A):=lid`. Thus, memory accesses in the first iteration of a loop get the parent `lid`. Tags are kept similarly to MBase, in the emulator. If there are no loops on the call stack, pushed functions are assigned new `lid`. Otherwise, new functions inherit the top loop `lid`.

Writing $\texttt{B} \leftarrow \texttt{A}$ to denote that pointer `B` is derived from pointer `A`, conceptually Howard detects arrays as follows. When pointer `B`, with $\texttt{B} \leftarrow \texttt{A}$, is dereferenced in iteration i, while `A` was dereferenced in a previous iteration, Howard treats `A` as a likely array element[3]. It stores information about the array in the loop head on DSDStack. The more iterations executed, the more array elements Howard discovers.

Loop unrolling. The algorithm is simple and intuitive and sketches the main idea fairly accurately, but it is a (slight) simplification of Howard's real array detection algorithm. In reality, we cannot just look at loop iterations, as common optimizations like loop unrolling force us to look at these patterns within a single iteration also. For completeness, we briefly explain the details.

Assume that the loop id of the current loop is $\texttt{Lid}_\texttt{T}$, while the previous element on the call stack has id $\texttt{Lid}_\texttt{S}$. Also assume that the pointer `C` is dereferenced for the first time in the loop `T`, where $\texttt{C} \leftarrow \texttt{B}$ and $\texttt{B} \leftarrow \texttt{A}$.

Howard now treats `B` as a likely arrays element if the following conditions hold:

1. $\texttt{MLid (B)} \geq \texttt{Lid}_\texttt{T}$, which means that `B` was accessed in the current loop (regardless of the iteration),

2. $\texttt{MLid (A)} \geq \texttt{Lid}_\texttt{S}$, which means that `A` was accessed either in the current loop or just before it.

It stores information about the array in `T`, the top loop head on DSDStack. Whenever a new piece of information is added, Howard tries to extend arrays already discovered in the top loop head. We will discuss shortly how this works when multiple loops access the same array. First, we look at boundary elements.

[3]There is a subtle reason why we do not classify `B` as an array element (yet): if the array consists of `struct`s, `B` may well point to a field in a `struct`, rather than an array element.

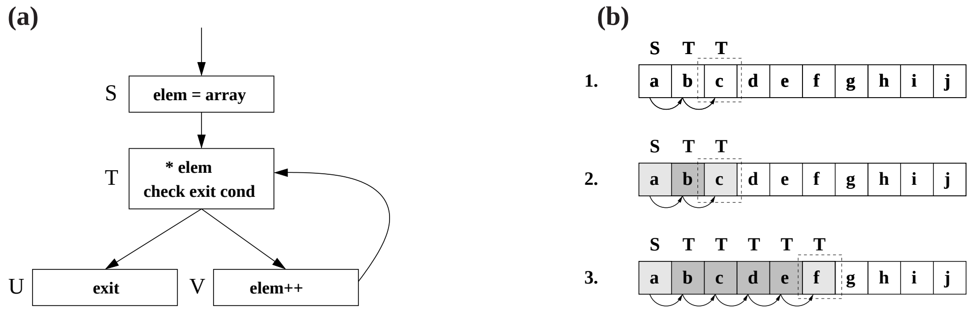


Figure 5. **(a)** Control flow graph of a loop that derives array element addresses relative to the previous element. Basic block T is the loop head, and V→ T the back edge. **(b)** An example ten-element array accessed by the loop in (a). Three diagrams present the information about the array gathered by Howard in different points of the loop execution. The dashed squares indicate the most recently accessed array element. Arrows represent base pointers as determined by the loop. S and T above the array elements are assigned during the loop execution and indicate loop head ids, MLids. In step 1, Howard checks that a, b, and c were accessed in the current loop (or just before), and decides (step 2) to store information about a new array [b]. It notes that the array should possibly be extended later to contain a and c. As more iterations of the loop are executed (step 3), the array contains elements [b-e] with a and f marked as potential extensions.

Boundary elements Because the first element of an array is often accessed before a new id is assigned to the loop head (remember, a new loop id is assigned only once the back edge is taken), Howard explicitly checks for extending the array. It looks for earlier memory accesses at the base pointers used to recursively derive the first element of the array. Before we continue with the algorithm, see Figure 5 for an example array detection scenario.

Multiple loops To handle arrays accessed in multiple loops or functions, arrays that are found in inner loops are passed to the previous element on DSDStack, whether it be a function or an outer loop node. Howard must decide if the findings of the inner exiting loop ought to be merged with the outer loop results or whether they represent internal arrays of nested structures and should be kept separately. Intuitively, Howard waits to determine whether (1) the array detected in the internal loop will be further extended in the outer loop – hinting at the existence of an iterator which prompts Howard to merge the results, or (2) whether the array detected in the internal loop is not extended further, and kept independent from the outer loop – prompting Howard to classify it as a nested structure.

4.4.2 Accesses relative to the base

The above technique helps us detect sequentially accessed arrays where each next address is relative to the previous one. Sometimes, however, accesses are relative to the base pointer. These accesses may or may not be sequential. For instance, randomly accessed arrays like hash tables fall in this category.

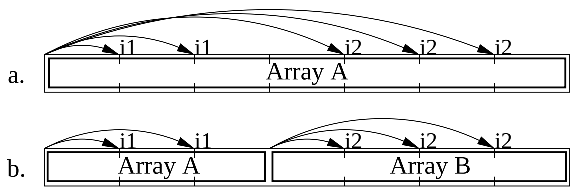


Figure 6. Array accesses from the base pointer. The labels denote whether the accesses are by instruction i1 or i2. In (a) we see a single array accessed by multiple instructions in a single loop, while in (b), the access patterns are similar except that we now have two arrays. Howard distinguishes the two cases by looking for a shared base pointer.

For this type of access Howard uses a second method that bears a superficial resemblance to the array detection method in Polyglot [12], but is considerably more powerful. Essentially, Howard tracks all instructions in a loop that access a set of addresses that can be mapped to a linear space ($stride * x + offset$) and that all share the same base pointer (See Figure 6.a). If the accesses share a base pointer, they almost certainly belong to the same array, even if the accesses came from different instructions in the loop. Moreover, we can easily extend the array to include boundary elements (e.g., a last element that is accessed outside the loop), because it will share the same base pointer. If the accesses do not share the base pointer (Figure 6.b), Howard classifies them as different arrays.

Existing methods like those used in Polyglot [12], and also Rewards [31], only check whether single instructions in a loop access an area that can be mapped to a linear space

($stride * x + offset$). Therefore, they can handle only the simplest of cases. They fail when the program: (1) accesses arrays in multiple loops/functions, (2) accesses boundary elements outside the loop, (3) has multiple instructions that access the same array within one loop (very common with unrolled loops or loops containing conditional statements like `if` or `switch`), and (4) allocates the arrays on the stack or statically. These are all common cases.

4.4.3 Are both methods necessary?

The two array detection methods in Howard are complementary. *Both* are necessary. The first will not detect accesses relative to the base pointer (like hash tables), while the second does not detect accesses where each next address is relative to the previous. In contrast, the combination of our two techniques works quite well in practice. Howard is able to detect nested arrays, hash tables and many other complex cases.

4.5 Final mapping

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

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. E.g., if base pointers hint at a structure with variable length fields, while the presumed array has fields of 4 bytes, Howard assumes the accesses are due to functions like `memset`. The analysis may find multiple interleaving arrays. 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.

4.6 Partial recovery of semantics

As mentioned earlier, type sinks are functions and system calls with well-known prototypes. Whenever Howard observes a call to one of these functions, it knows the type of the arguments, so it can attach this label to the data structure. Howard propagates these type labels, e.g., when a labeled structure is copied. In addition, it also propagates the type information to the pointers that point into the memory.

In our experience, the recovery works only for a limited set of all data structures, but some of these are important for debugging and forensics. For instance, it may be interesting to see which data structures contain IP addresses. We have implemented type sinks for `libc` functions and for system calls, as these are used by practically all applications. In general, Howard does not depend on type sinks, but it is capable of using them when they are available. All results in the remainder of this paper were obtained without turning on type sinking at all.

5 Applications

To demonstrate the usefulness of Howard, we describe two new applications: binary analysis with reconstructed symbol tables, and retrofitting security to legacy binaries not designed with security in mind.

5.1 Binary analysis with reconstructed symbols

To aid forensics and reverse engineering, Howard automatically generates new debug symbol tables for stripped binaries. In this example, we focus primarily on the new techniques for detection of data structures. However, we also want to show how we complement Rewards' method of recognizing well-known functions (known as *type sinks*) and propagating the types of their arguments [31]. For this reason, we also add a minimal set of type sinks.

The symbol tables Howard generates are generic and we can use them with any common UNIX tool. Of course, they are not entirely complete. For instance, we only have exact semantics for the subset of data structures that derive from type sinks. Also, we cannot generate out of thin air the correct names of fields and variables. Nevertheless, the recovered symbol table allows us to analyze the data structures of applications that were practically impossible to analyze otherwise.

Figure 7 shows a screenshot of a real `gdb` session with reconstructed symbols. Suppose we have purchased a program that crashes occasionally. We want to analyze the program and perhaps reverse engineer parts of it. Without Howard, common disassemblers (like those of `gdb` or IDA Pro) help to analyze the instructions, but not the data structures. In this example, we show that we can now analyze the data also. For demonstration purposes, we use a stripped version of `wget` as our demo binary, and show that we can analyze it with the `gdb` debugger.

To show both the unadorned data structure recovery *and* recovery of semantics, this example uses a truly minimal set of type sinks. Specifically, our type sinks consist (only) of `inet_addr()` and `gethostbyname()`.

The scenario is that we have witnessed a crash. Thus, we start our analysis with an instruction prior to the crash (`0x805adb0`) ① and try to find information about the variables in the scope of the current function (`info scope` ② and then `print variables` ③). Where the stripped binary normally does not have *any* information about the variables, we see that Howard reconstructed most of them (`struct`s, pointers, and strings) and even recovered partial semantics. For instance, we find pointers to a `struct hostent` and an `inetaddr_string`[4].

We could print out the contents of the `struct hostent`, but really we are interested in the data structures pointed to by the various pointers – for instance the pointer to a `struct` identified by `pointer_struct_1_0`. Unfortunately, it currently has the value NULL, so there is nothing interesting to see and we have to wait until it changes. We do this by setting a watch point on the pointer④. In our opinion, the ability to inspect data structures and set watch points clearly shows the usefulness of the data structures recovered by Howard.

Once the pointer changes, and receives its new value `0x80b2678`, we can analyze the corresponding memory area. We see that it points to a structure containing: an integer with value 3, a pointer to a `struct`, a one-byte field, and a four-byte field ⑤. If we follow the pointer, we see that it points to a struct with two fields, one of four bytes and one of type `in_addr_t` ⑥. We can even examine the value of the IP address and see that it corresponds to 74.125.77.147 ⑦.

Moreover, we see that our `struct` is in an array of size 3 by dividing the amount of memory allocated (`malloc_usable_size`) by the size of an element ⑧. Thus, we can make an educated guess that the integer value of 3 we found earlier denotes the number of IP addresses in the array.

To complete the admittedly simple example, we also print out these IP addresses ⑨. For reasons of space, we stop the scenario here. The purpose of the example is merely to show that we *can* debug and analyze stripped binaries in the same way as when debugging and analyzing binaries with debugging symbols.

We emphasize that by means of type sinks alone (e.g., using Rewards [31]), we would have obtained hardly any data structures as almost all of them are internal to `wget`. Eventually, the IP addresses are used, so they would appear, but no `struct` or array would have been found. In addition, connecting the IP addresses to other structures from our entry point into the code (the break point) would have been completely impossible. In contrast, with Howard we can progressively dig deeper and trivially find the connections between the data structures.

[4]Names are generated by Howard. Prefixes like `field_`, `pointer_` are for convenience and are not important for this paper.

5.2 Protection against memory corruption

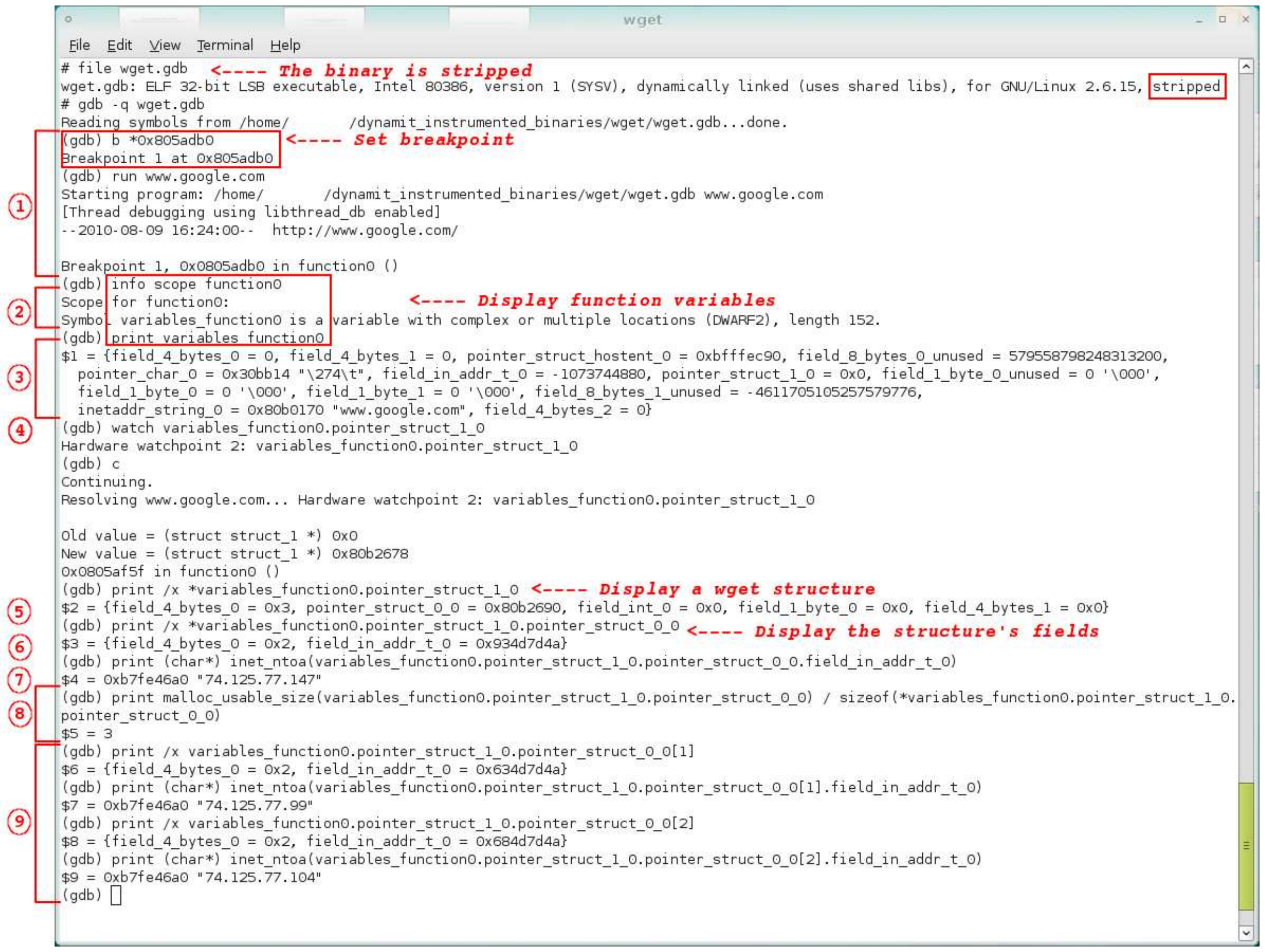
Memory corruption errors such as buffer overflows are responsible for a large share of the remote attacks on the Internet. An overflow occurs when a C program reads more bytes than fit in the buffer, so that important data in the memory above the buffer is overwritten. The overwritten data can be either control data such as function pointers [36], or non-control data such as clearance levels [14].

Protection of legacy binaries is very hard. Existing solutions tend to be too slow (like taint analysis [35]) or incomplete (ASLR [11], StackGuard [21], NX/DEP/W⊕X support [25]). For instance, ASLR, StackGuard, and NX/DEP/W⊕X offer no protection against attacks against non-control data. In addition, a recent report shows that both DEP and ASLR in Windows third party applications are typically either improperly implemented or completely overlooked [40]. Finally, even if configured correctly, the protection offered by ASLR is quite limited [41].

To demonstrate the problem and our solution, we use an example based on a real vulnerability in `nullhttpd` [1] as shown in Figure (8.a). Given a CGI command, the Web server calls `ProcessCGIRequest` with as arguments the message it received from the network and its size. The program copies the message in the `cgiCommand` buffer and calls `ExecuteRequest` to execute the command. As the copy does not check for bounds violations, the buffer may overflow and overwrite the heap.

Protection Many papers have looked at patching programs (source or binary) *after* a vulnerability was detected [34], and recent work looked at extending the compiler framework to protect against new memory corruption attacks given the source [4]. So far, no one has shown how one can really protect (i) existing binaries, (ii) in production environments, (iii) against such attacks (on both control and non-control data), and (iv) *pro-actively*, without access to vulnerabilities, source code, or even symbol tables. This means that all legacy binaries with unknown vulnerabilities are left at the mercy of attackers. We now describe a promising new way to protect this important class of applications against memory corruption.

Howard has no problem recovering buffers such as `cgiCommand`. However, since we use code coverage to excavate the array, one might think that Howard would discover an array that is larger than 1024 bytes. After all, since code coverage can feed the function with arbitrarily long messages, wouldn't Howard 'discover' an arbitrarily long

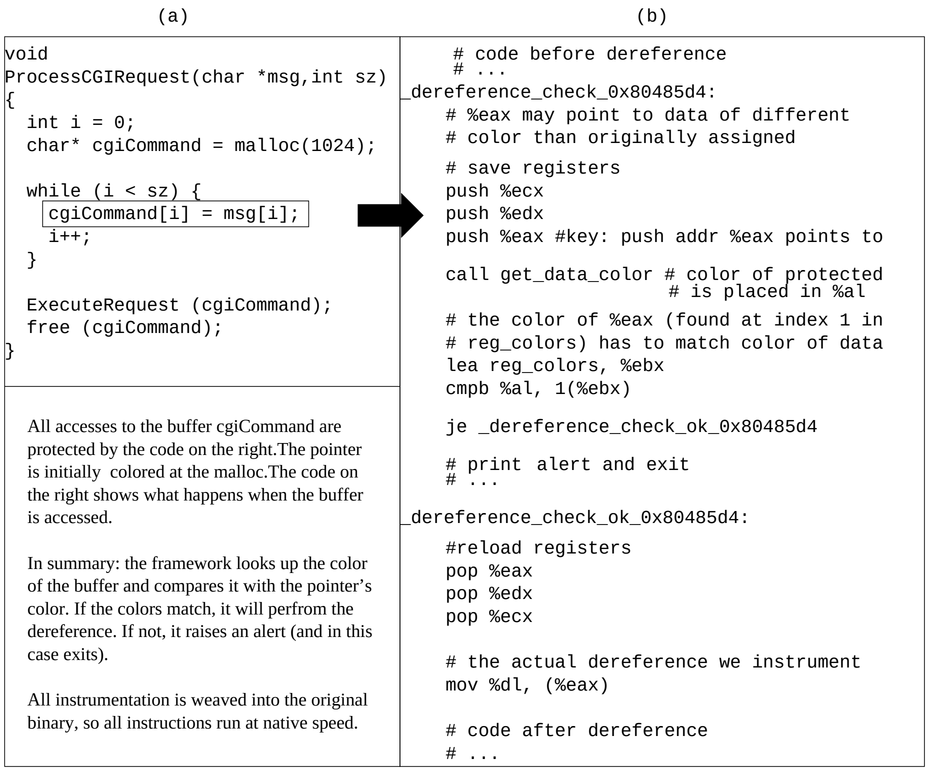

wget
File Edit View Terminal Help
<---- The binary is stripped
<---- Set breakpoint
<---- Display function variables
<---- Display a wget structure
<---- Display the structure's fields
1
2
3
4
5
6
7
8
9

```
# file wget.gdb
wget.gdb: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.15, stripped
# gdb -q wget.gdb
Reading symbols from /home/       /dynamit_instrumented_binaries/wget/wget.gdb...done.
(gdb) b *0x805adb0
Breakpoint 1 at 0x805adb0
(gdb) run www.google.com
Starting program: /home/       /dynamit_instrumented_binaries/wget/wget.gdb www.google.com
[Thread debugging using libthread_db enabled]
--2010-08-09 16:24:00--  http://www.google.com/

Breakpoint 1, 0x0805adb0 in function0 ()
(gdb) info scope function0
Scope for function0:
Symbol variables_function0 is a variable with complex or multiple locations (DWARF2), length 152.
(gdb) print variables_function0
$1 = {field_4_bytes_0 = 0, field_4_bytes_1 = 0, pointer_struct_hostent_0 = 0xbfffec90, field_8_bytes_0_unused = 579558798248313200,
  pointer_char_0 = 0x30bb14 "\274\t", field_in_addr_t_0 = -1073744880, pointer_struct_1_0 = 0x0, field_1_byte_0_unused = 0 '\000',
  field_1_byte_0 = 0 '\000', field_1_byte_1 = 0 '\000', field_8_bytes_1_unused = -4611705105257579776,
  inetaddr_string_0 = 0x80b0170 "www.google.com", field_4_bytes_2 = 0}
(gdb) watch variables_function0.pointer_struct_1_0
Hardware watchpoint 2: variables_function0.pointer_struct_1_0
(gdb) c
Continuing.
Resolving www.google.com... Hardware watchpoint 2: variables_function0.pointer_struct_1_0

Old value = (struct struct_1 *) 0x0
New value = (struct struct_1 *) 0x80b2678
0x0805af5f in function0 ()
(gdb) print /x *variables_function0.pointer_struct_1_0
$2 = {field_4_bytes_0 = 0x3, pointer_struct_0_0 = 0x80b2690, field_int_0 = 0x0, field_1_byte_0 = 0x0, field_4_bytes_1 = 0x0}
(gdb) print /x *variables_function0.pointer_struct_1_0.pointer_struct_0_0
$3 = {field_4_bytes_0 = 0x2, field_in_addr_t_0 = 0x934d7d4a}
(gdb) print (char*) inet_ntoa(variables_function0.pointer_struct_1_0.pointer_struct_0_0.field_in_addr_t_0)
$4 = 0xb7fe46a0 "74.125.77.147"
(gdb) print malloc_usable_size(variables_function0.pointer_struct_1_0.pointer_struct_0_0) / sizeof(*variables_function0.pointer_struct_1_0.
pointer_struct_0_0)
$5 = 3
(gdb) print /x variables_function0.pointer_struct_1_0.pointer_struct_0_0[1]
$6 = {field_4_bytes_0 = 0x2, field_in_addr_t_0 = 0x634d7d4a}
(gdb) print (char*) inet_ntoa(variables_function0.pointer_struct_1_0.pointer_struct_0_0[1].field_in_addr_t_0)
$7 = 0xb7fe46a0 "74.125.77.99"
(gdb) print /x variables_function0.pointer_struct_1_0.pointer_struct_0_0[2]
$8 = {field_4_bytes_0 = 0x2, field_in_addr_t_0 = 0x684d7d4a}
(gdb) print (char*) inet_ntoa(variables_function0.pointer_struct_1_0.pointer_struct_0_0[2].field_in_addr_t_0)
$9 = 0xb7fe46a0 "74.125.77.104"
(gdb)
```

Figure 7. Analysis and disassembly with recovered symbols



```
void
ProcessCGIRequest(char *msg,int sz)
{
  int i = 0;
  char* cgiCommand = malloc(1024);

  while (i < sz) {
    cgiCommand[i] = msg[i];
    i++;
  }

  ExecuteRequest (cgiCommand);
  free (cgiCommand);
}
```

All accesses to the buffer cgiCommand are protected by the code on the right.The pointer is initially colored at the malloc.The code on the right shows what happens when the buffer is accessed.

In summary: the framework looks up the color of the buffer and compares it with the pointer's color. If the colors match, it will perfrom the dereference. If not, it raises an alert (and in this case exits).

All instrumentation is weaved into the original binary, so all instructions run at native speed.

```
    # code before dereference
    # ...
_dereference_check_0x80485d4:
    # %eax may point to data of different
    # color than originally assigned
    # save registers
    push %ecx
    push %edx
    push %eax #key: push addr %eax points to

    call get_data_color # color of protected
                        # is placed in %al

    # the color of %eax (found at index 1 in
    # reg_colors) has to match color of data
    lea reg_colors, %ebx
    cmpb %al, 1(%ebx)

    je _dereference_check_ok_0x80485d4

    # print alert and exit
    # ...
_dereference_check_ok_0x80485d4:
    #reload registers
    pop %eax
    pop %edx
    pop %ecx

    # the actual dereference we instrument
    mov %dl, (%eax)

    # code after dereference
    # ...
```

Figure 8. (a) Buffer overflow vulnerability in a simplified Web server, (b) Instrumentation

buffer? Fortunately, the answer is no and the reason is that the bytes just beyond the buffer (in this case memory management meta data) are accessed and classified differently. As a result, Howard sees two conflicting classifications and picks the safest one, ending the buffer at the right boundary.

In our approach, we protect all buffers which were classified as arrays, and all buffers which were followed by unused memory regions. The latter ones might be either arrays or structures. Since Howard not always has enough information to choose the correct data structure (also refer to "missed short arrays" in Section 8), to err on the safe side we have decided to protect these memory regions as well.

After identifying all potentially vulnerable data structures (mostly arrays), we harden the binary as follows. First we give each vulnerable data structure a unique color. For the rest of memory the color is set to 0. The colors are kept in a memory map. Next, when we initialize a pointer to, say, buffer B, we give the pointer B's color. The initialization varies, depending on the type of memory. For global data, this happens when the program starts, for stack variables when the function is called, and for heap data at the time of the allocation.

Third, we instrument those (and only those) instructions that dereference the pointer to check whether it is still pointing to memory of the right color B[5]. Fourth, because we do not wish to maintain also a large memory map of colors for all pointers, we ensure that only pointers in registers have a color. Whenever a pointer is stored to memory, we check whether it is still pointing to the right area. If so, everything is fine and when the pointer is reloaded to a register, we give it the color of the byte to which it is pointing. If, however, the pointer does not point to an area of the right color, we explicitly tag the pointer with a single bit to indicate that it needs to obtain its color from a table. As this situation is rare, the overhead is minimal.

To add the instrumentation, we rewrite the binary using rewriting techniques similar to Dynamos [32]. We emphasize again that the instrumentation is applied only to exactly those instructions that need to be checked. The rest of the code is not touched. Figure (8.b) shows how we hardened the memory dereference in the web server.

To test the practicality of our approach, we applied the Howard protection to the main array of `lighttpd`, the popular high-performance web server that, at the time of writing, is used by such popular sites as YouTube, SourceForge, Wikimedia, Meebo, and ThePirateBay. The protected array is used to store strings, and in our experiment setting

[5]So it is fine if the pointer points beyond the buffer, as long as it is not dereferenced.

accounts for 80% of the whole heap memory usage. The runtime overhead of the current implementation is almost a factor of 2. Although the overhead of instrumentation is not negligible, we believe that the approach is practical. First, there are many ways in which we can optimize the instrumentation. Second, there are many legacy applications that are not extremely time critical, but that should not be compromised.

Discussion. Currently, while the instrumentation framework works, it is still an early prototype. For instance, when instrumenting a binary, some steps are still manual. This will be remedied shortly. In addition, we have not looked at all performance optimization yet. Finally, while we have taken care to avoid possible false positives, more extensive evaluation is needed across a wide range of applications. Nevertheless, the important message, in our opinion, is that with Howard we have a chance to protect arbitrary legacy C applications against the most common form of exploit without any access to source code or even symbol table, and without knowing in advance about the presence of vulnerabilities in the code. For instance, the Web server of Figure 8 now raises an alert whenever it reads too many bytes in the `cgiCommand` buffer.

6 Comments and limitations

Obviously, Howard is not flawless. In this section, we discuss some generic limitations we have identified.

Limitations in array detection We discuss both the limitations of our array detection techniques and their impact on our applications – especially on our second application which uses the data structures to protect legacy binaries. It is crucial that possible misclassifications do not cause false positives.

- *At least four accesses.* To make array detection more accurate, Howard will not recognize an array if fewer than four of its elements are accessed in a loop. In that case, it will be classified as a structure. This misclassification can only cause false negatives, but never false positives. Indeed, if an array remains undetected, Howard does not try to protect instructions accessing it.

- *Merging with unused bytes.* We merge detected arrays with any unused bytes that follow the array. Doing so prevents identification of buffers that are too small (and thus false positives in the case of protection against memory corruption). In general, the Howard protection scheme, was designed to err on the safe side. Observe that even if we would add unused bytes that do not belong to the same buffer, we still protect the next *used* field or variable from buffer overflows.

- *Incorrect merges.* If a structure consists of one field followed by an array, and the field and array are accessed in sequence, it is impossible to classify it correctly solely based on memory access patterns. As Howard always extends array to include the first element unless it finds evidence that it does not match (e.g., if because size or type are different), it could lead to overestimation of the array length. Again, this can never cause false positives.

- *Separate last element accesses.* It may happen that all but the last elements of an array form a *chain*, while the last element is *always* accessed separately, e.g., when a string is first copied to a destination buffer, and then extended with `EOL` or `NULL`. Howard misclassifies the memory as a structure containing an array and a separate field. Even though the last element is not attached to the array, this does not cause false positives for our application. Indeed, if the program never exploits the connection between the last element and the remaining part of the array, it is not important for us to join them either. Also, if the size of the array varies across multiple runs of the program, Howard prefers to err on the safe side, merges the elements, and reports accurate results. In general, even if Howard 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 analyze the layout.

Other limitations

- Howard cannot recognize nested `structs` if the inner `struct` is never accessed separately. In that case, Howard returns a single large structure. As the result is equivalent, we do not consider this a problem.

- Unions might exhibit more than one memory access pattern. As a result Howard would report a data structure being a *merge* (or *intersection*) of the multiple structures included in the union. Howard might certainly report an incorrect (more detailed) interpretation of fields, but it does not lead to false positives.

- Howard gets confused by certain custom memory allocators. Specifically, it can handle slab-like allocators, but a generic, application-specific memory allocator (such as that of Apache) leads to misclassifications. It stems from the fact that a memory region serving the allocator exhibits mixed access patterns inherited from various different structures/arrays for which it was used. As a result, Howard would classify such buffer as either an array of (perhaps) 4-byte fields or a highly nested structure.

- Howard does not analyze local variables of functions reached using a `jump` rather than a `call`.

7 Code transformation: compiler optimization and obfuscation techniques

In production code, compilers apply optimizations to improve runtime performance. Such optimizations may change the code substantially. Likewise, obfuscation tools change the source code or binary to reduce the ability to understand or reverse engineer the program. In this section, we introduce popular optimizations and obfuscation techniques and discuss how they influence Howard's data structure detection results. We treat compiler optimizations in Section 7.1 and obfuscation techniques in Section 7.2. In both cases, we limit ourselves to the techniques that are relevant and that may affect data structure detection.

7.1 Compiler optimizations

Howard detects data structures in `gcc`-generated `x86 C` binaries. Even though our techniques were not designed with any specific compiler in mind and should work with other binaries also, we conducted all our experiments on `gcc-4.4` binaries. So we focus our discussion on `gcc`.

7.1.1 Data layout optimizations

Data layout optimizations [29, 28] adapt the layout of a data-structure to its access patterns in order to better utilize the cache by increasing spatial locality. They include structure splitting and field reordering transformations.

In general, Howard detects data structures at runtime, so the analysis results correspond to the optimized code and data–which may be different from what is specified in the source. This is known as WYSINWYX (What You See Is Not What You eXecute) [9] and while it complicates reverse engineering (for instance, some data structures may be reordered or transformed), analyzing the code that really executes is of course 'the right thing'. Without it, we would not be able to protect buffers from overflows, or perform proper forensics.

7.1.2 Loop optimizations

Loop transformations [3, 6] reduce loop overhead, increase instruction parallelism, and improve register, data cache or TLB locality. Popular transformations include: (1) *loop unrolling*, where the body of a loop is replicated multiple times in order to decrease both the number of loop condition tests

Loop code:

```
for(i = 0; i < 64; i++)  arr1[i] = i;
```

And two possible executions:

```
addr = arr1;
for(i = 0; i < 64; i++){
 *addr = i;
 addr += 1;
}
```

```
addr = arr1;
for(i = 0; i < 16; i++){
 *addr = i<<2;
 *(addr + 1) = i<<2+1;
 *(addr + 2) = i<<2+2;
 *(addr + 3) = i<<2+3;
 addr += 4;
}
```

Figure 9. An example loop and two possible ways in which the loop can be executed: the non-transformed one on the left hand side, and the unrolled one on the right hand side.

and the number of jumps, (2) *loop peeling*, where a loop is peeled, a small number of iterations are removed from the beginning or end of the loop and executed separately to remove dependencies created by the first of last few loop iterations, (3) *loop blocking*, where a loop is reorganized to iterate over blocks of data sized to fit in the cache.

As described in Section 4.4, Howard recovers arrays when the program accesses them in loops. To increase the accuracy of the analysis, Howard's algorithm allows for arrays accessed in multiple loops, and checks for array elements accessed before or after the loop body. Basically, when a transformed loop accesses an array, all its elements are *classified together*.

However, loop transformations may change not only the layout and number of loops, but also the memory access patterns. As an example, Figure 9 presents a simple loop accessing an array, and two possible ways in which this loop can be executed. Refer to Figure 10 to observe array `arr1` access patterns executions. We can see that depending on the execution, the array is either classified as an array of single fields (as desired) or as an array of 4-field structures. Even though `arr1` is possibly misclassified here, it might be used in other ways somewhere else in the program, and we might eventually get it right. In all similar cases, Howard cannot do anything about not entirely accurate classifications, as its analysis is based solely on memory access patterns.

7.1.3 Optimizations affecting function frame

There are numerous compiler optimization which affect the way we perceive a function's frame. First, functions can be inlined, what means that they are integrated into their callers. In this case the small inlined function is not called separately, but its function frame extends the caller's one. Second, (some of) the input function arguments might be passed through the registers and not through the stack. Also,

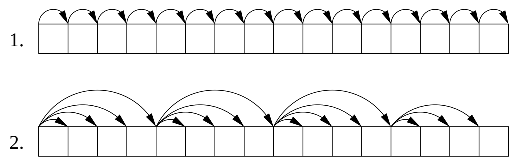


Figure 10. Memory access patterns realized in two executions of the loop in Figure 9. The top one represents the non-transformed one, and the bottom one - the unrolled one. As in the other figures, arrows represent base pointers as determined by the loop.

`gcc` might analyse the program to determine when values passed to functions are constants. These are optimized accordingly.

Howard is expected to reflect functions in the way they appear in the binary. In the case of inlined functions we get just one extended function frame. Likewise, since Howard analyzes memory access patterns only, it cannot spot function parameters passed through the registers or precomputed by the compiler. As this inaccuracy does not affect our applications, we did not worry about it.

7.2 Obfuscation techniques

Code obfuscation techniques aim to reduce the ability to understand or reverse engineer the program. Data transformations [19] that obscure data structures used in the source application are the most relevant to Howard, and we focus our discussion on them. We also briefly mention anti-dynamic analysis techniques (Section 7.2.3).

7.2.1 Obfuscating arrays

Numerous transformations can be devised for obscuring operations performed on arrays [16, 46, 26]. Popular techniques include: (1) *array splitting*, where an array is split into several subarrays, (2) *array merging*, where two or more arrays are merged into one array, (3) *array folding* and *flattening*, where the number of dimensions is increased or decreased, respectively.

As before, Howard is expected to reflect arrays in the way they appear and are accessed in the binary. In the case of split, folded or flattened arrays, Howard is supposed to report the new transformed data structures. When two or more arrays are merged to form a new array, the results of the analysis greatly depend on the merging method. For example, one could combine arrays `arr1[N1]` and `arr2[N2]` in such a way that the new array `arr3` contains `N1` elements of `arr1` followed by `N2` elements of `arr2`. In this case, it is very probable that Howard reports `arr3` as a structure containing two arrays: an `N1`- and an `N2`-element long. Another merging technique could interleave `arr1`'s elements with `arr2`'s elements. Here, we perhaps expect Howard to report one array of `[N1+N2]` elements.

Since Howard analyzes memory access patterns only, it cannot recognize that certain arrays are used in *similar* ways by *similar* functions, and containing *similar* elements. Thus, it cannot say that some arrays might share a higher level data type. Howard concerns itself with low level data structures only, and limits itself to recognizing sizes and basic types of data structures. Understanding contents and usage scenarios is part of our future work.

7.2.2 Obfuscating variables

There is a lot of work on hiding *values* of sensitive variables from static and dynamic analysis [17, 47]. For example, instead of using a variable, one can use a piece of code that generates the value dynamically. Also, like arrays, variables may be split or merged. None of our test applications contain such obfuscation, so we left this for future work.

7.2.3 Anti-dynamic analysis techniques

Howard observes the behavior of a program in runtime, and so it is crucial that the program does not refuse to run in the instrumented processor emulator. However, there exist techniques that could be used by programs running in virtualized environments to determine that they are running in a virtual machine rather than on native hardware [37, 27]. When this happens, Howard cannot perform its analysis.

8 Evaluation

Howard can analyze any application on the Linux guest on our (QEMU-based) emulator. For instance, we successfully applied Howard to games like `glines` and `gnometris`, but also to complex binaries like the Linux loader `ld-2.9.so`, and huge ones like Apache[6]. However, for good results we need code coverage and for code coverage we currently depend on what is supported by KLEE [13] and existing test suites.

Our experimental setup is shown in Figure 1. We conduct the code coverage runs offline, using KLEE on LLVM version 2.6 with home-grown modifications to handle networking. All applications run on a Linux-2.6.31-19 kernel. We then use `klee-replay` to replay the application on top of our Howard analyzer with the inputs generated by KLEE. If our coverage is low, we use normal test suites for the applications.

Starting out with small applications (fortune), we worked towards progressively larger applications. In the plots below, we include results for programs of several tens

[6]244.000 lines of code (LoC), as reported by David Wheeler's `sloccount` (`www.dwheeler.com/sloccount/`).

of thousands LoC, including the `wget` download program and the `lighttpd` high-performance web server. Table 1 shows the applications and the coverage we obtained with KLEE and the test suites. We also applied Howard to utilities in CoreUtils, but these tend to be so small (a few hundred lines, typically) that they are not representative of real applications, and we do not discuss them here.

Results To verify Howard's accuracy, we compare the results to the actual data structures in the programs. This is not entirely trivial. We cannot compare to the original source code since aggressive compiler optimizations may change the binary significantly ("what you see is not what you execute" [9]). Thus, all the results presented in this section were obtained for binaries for which we could also generate symbol tables to compare our results with. This way we were able to get ground truth for real world applications.

We will start with a few quick observations. First, Howard cannot discover variables that always remain unused, but this probably should not count as a 'missed' data structure. Second, all these results were obtained solely with Howard's 'core' data structure excavation techniques. In other words, we turned off *all* type sinking for these tests.

Figure 11 presents our results in terms of accuracy for both the stack and the heap. The accuracy is calculated both for the number of variables, and for the total number of allocated bytes. In the case of stack memory we evaluated the accuracy of Howard analysis for all functions used during our experiments. Thus we did not count the stack variables used by never invoked functions. In the case of heap, we simply considered all allocated memory regions. The plots show Howard results in five categories:

- **OK**: Howard identified the entire data structure correctly (i.e., a correctly identified structure field is not counted separately).
- **Flattened**: fields of a nested structure are counted as a normal field of the outer structure. On the stack these are usually structures accessed via EBP, as explained in Section 3.
- **Missed**: Howard misclassified the data structure.
- **Unused**: single fields, variables, or entire structures that were never accessed during our tests.
- **Unused array**: this is a separate category that counts all arrays not recognized because there were insufficient accesses to the array.

The general conclusion is that practically all memory is either detected correctly or unused. Moreover, for practically all arrays and `structs` the lengths were identified correctly.

The stack has more unused memory than the heap. This is expected. Whenever a function is called, the whole frame is allocated on the stack, regardless of the execution path taken in the function's code. Heap memory is usually allocated on demand. Much of the unused bytes are due to the limited coverage. As we never ran KLEE for a longer than a few hours, it may be that by running it longer[7] we obtain better coverage.

The stack also counts more structures that are flattened. Again, this is not surprising. As explained in Section 3, structure field addresses may be calculated relative to the beginning of the function frame. In that case, Howard has no means of classifying the region as a structure. On the heap this is less common. Indeed, if an application allocates memory for a structure, it refers to the structure's fields from the base of the structure, i.e., the beginning of the allocated region. However, Howard may still misclassify in the case of a nested structure. This happens for instance in `fortune`, where a 88-byte `FILEDESC` structure contains a nested 12-byte `STRFILE` structure.

The main source of *missed* are data structures occasionally classified as arrays of 4-byte fields. Assume that an application uses a simple structure with 4-byte fields based at the beginning of the structure (i.e., no inner structures). If none of the fields is 1-byte long, nor is a pointer, while a `memset` function is used for the structure initialization, we have no means for discarding the array hypothesis.

As the vast majority of `structs` and arrays reside on the heap, we zoom in on these results for more details. Figure 12 breaks down the overall results to show how well Howard discovers individual fields and different types of arrays. The two plots correspond to structures that are *separately allocated* on the heap and on arrays (possibly containing structures). We label the results as follows:

- **Structures**
  - **OK**: fields correctly identified.
  - **Missed**: fields incorrectly identified.
  - **Unused**: fields missed because the program did not use them.
  - **Flattened**: fields in nested structures that Howard placed at the wrong nesting level.
- **Arrays**
  - **OK**: arrays correctly/incorrectly identified.
  - **Missed**: arrays correctly/incorrectly identified.

[7]Or running it better. Driving KLEE down the right execution path is not always trivial.

Prog	LoC	Size	Funcs%	Vars%	How tested?	KLEE%
wget	46K	200 KB	298/576 (51%)	1620/2905 (56%)	KLEE + test suite	24%
fortune	2K	15 KB	20/28 (71%)	87/113 (77%)	test suite	N/A
grep	24K	100 KB	89/179 (50%)	609/1082 (56%)	KLEE	46%
gzip	21K	40 KB	74/105 (70%)	352/436 (81%)	KLEE	54%
lighttpd	21K	130 KB	199/360 (55%)	883/1418 (62%)	test suite	N/A

Table 1. Applications analyzed with Howard. LoC indicates the lines of code according to `sloccount`. Size is the approximate size of the text segment. Func% is the fraction of functions that the tests exercised (KLEE or test suite). Vars% is the fraction of variables used in the tests (KLEE or test suites), and KLEE% is the coverage offered by KLEE (if any).

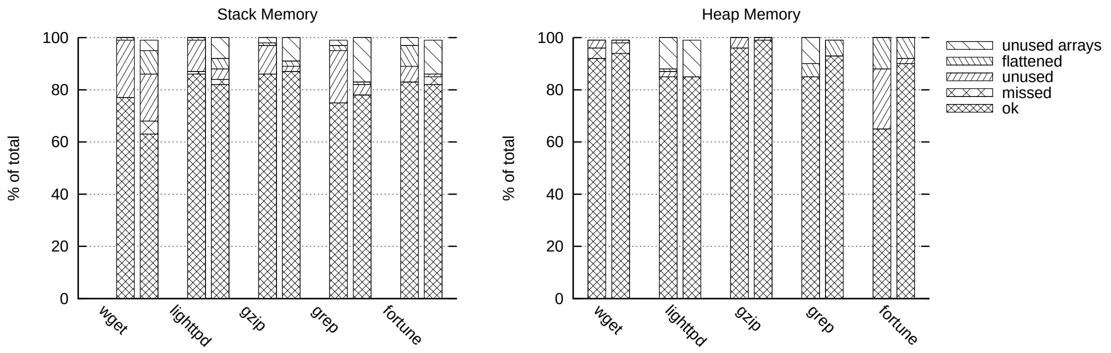


Figure 11. The accuracy of the analysis per application. For each application the first bar denotes the accuracy in number of variables, and the second in number of bytes.

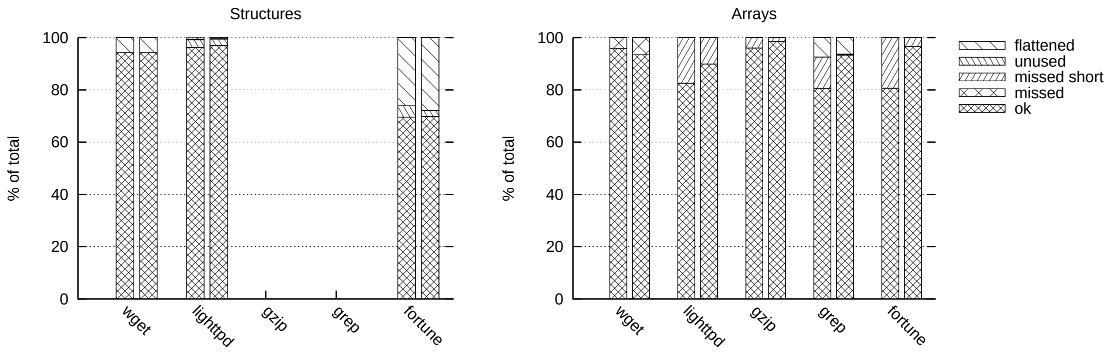


Figure 12. The accuracy of the analysis per application. For each application the first bar denotes the accuracy in number of variables, and the second in number of bytes.

- **Missed short**: arrays missed because they were accessed fewer than 4 times.
- **Unused**: arrays missed by Howard because they were not used at all.
- **Flattened**: fields of structs in arrays classified as separate elements.

If an array contains structs and any field of any struct is missed, we count it as a missed array. A flattened array means at least one of the structs was flattened. In other words, we analyze structs that were allocated individually separate from structs in arrays. The reason for doing so is that otherwise the arrays would completely dominate the analysis of structs. For instance, if all fields in an array of 1000 structures were classified correctly, this would artificially inflate the number of correctly identified fields.

In the plots in Figure 12 we see that provided they are used at all, Howard mostly detects fields and structures quite accurately. Most of the misclassifications are also minor. For instance, the flattened structure in fortune consists of an 88-byte FILEDESC structure that contains a nested 24-byte STRFILE structure, where the inner structure is never used separately. Examples of the arrays missed in wget are a char* array that is reallocated and then not used anymore. Howard classifies it as an int* array. There are also misclassifications when the last element is accessed outside the loop and relative to the base pointer, rather than the previous element. For grep, some arrays of structs were classified as arrays of individual fields. As explained above, we decided to count them in the flattened arrays, rather than in the structures. Also, for grep (and gzip) *no* structures appear on the heap except in arrays (hence the lack of data in the plot on the left).

Performance. Most of the overhead in Howard is due to code coverage. It takes several hours to obtain reasonable coverage of real-life applications. Howard is considerably cheaper. Still, since we either re-run the KLEE experiments or the test suites on a heavily instrumented binary, the analysis takes more time than running the applications natively. For instance, programs like gzip and grep took 5 and 15 minutes to analyze, respectively. Of all the applications we tried, grep took the longest. As Howard performs a once-only, offline analysis, these results show that even with the current unoptimized version, we can realistically extract data structures from real applications.

9 Conclusions

We have described a new technique, known as Howard, for extracting data structures from binaries dynamically without access to source code or symbol tables by observing how program access memory during execution. We have shown that the extracted data structures can be used for analyzing and reverse engineering of binaries that previously could not be analyzed before, and for protecting legacy binaries against memory corruption attacks. As until now data structure extraction for C binaries was not possible, we expect Howard to be valuable for the fields of debugging, reverse engineering, and security.

Future work Our focus throughout this project was on detecting low level data structures in stripped binaries. We have demonstrated that with Howard we have a chance to protect arbitrary legacy C binary against the most common form of exploit. We plan to make our current instrumentation framework fully automated, and to further evaluate our approach.

Future research also addresses the issue of lifting the current low level analysis to higher level data structures. We intend to observe connections between structures, and based on that information reason about pointer structures like linked lists, trees and hash tables.

10 Acknowledgments

This work has been supported by the European Commission through projects FP7-ICT-216026-WOMBAT and FP7-ICT-257007 SYSSEC. Also, with the support of the Prevention, Preparedness and Consequence Management of Terrorism and other Security-related Risks Programme European Commission - Directorate-General Home Affairs. This publication reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

References

[1] Null httpd remote heap overflow vulnerability. http://www.securityfocus.com/bid/5774l.

[2] Ollydbg. http://www.ollydbg.de/.

[3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition*. Pearson Education, Addison Wesley, 2007.

[4] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *IEEE Symposium on Security and Privacy*, 2008.

[5] P. Akritidis, M. Costa, M. Castro, , and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th Usenix Security Symposium*, August 2009.

[6] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26:345–420, 1994.

[7] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executables. In *Lecture Notes in Computer Science*, pages 250–254. Springer, 2005.

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

[9] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What you see is not what you execute. In *In Verified Software: Theories, Tools, Experiments*, page 1603, 2007.

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

[11] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2005. USENIX Association.

[12] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[13] C. Cadar, D. Dunbar, and D. 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.

[14] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, 2005.

[15] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, Paris, France, April 2010.

[16] S. Cho, H. Chang, and Y. Cho. Implementation of an obfuscation tool for C/C++ source code protection on the XScale architectures. In *Proceedings of the 6th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems*, 2008.

[17] S. Chow, P. A. Eisen, H. Johnson, and P. C. v. Oorschot. White-box cryptography and an aes implementation. In *Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography (SAC '02)*, 2002.

[18] M. Christodorescu, N. Kidd, and W.-H. Goh. String analysis for x86 binaries. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*, 2005.

[19] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Sciences, The University of Auckland, Auckland, New Zealand, 1997.

[20] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda. Prospex: Protocol specification extraction. In *SP '09: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, 2009.

[21] C. Cowan, C. Pu, D. Maier, H. Hintony, W. J., P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *7th USENIX Security Symposium*, 1998.

[22] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008*, pages 255–266, San Diego, CA, December 2008.

[23] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: automatic reverse engineering of input formats. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, 2008.

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

[25] T. DeRaadt. Advances in openbsd. CanSecWest, `http://www.openbsd.org/papers/csw03/index.html`, April 2003.

[26] S. Drape. Generalising the array split obfuscation. *Information Sciences*, 177:202–219, 2007.

[27] P. Ferrie. Attacks on virtual machine emulators. White paper, Symantec advanced threat research, 2007.

[28] O. Golovanevsky and A. Zaks. Struct-reorg: current status and future perspectives. In *Proceedings of the GCC Developers Summit*, 2007.

[29] M. Hagog and C. Tice. Cache aware data layout reorganization optimization in gcc. In *Proceedings of the GCC Developers Summit*, 2005.

[30] 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*, 2008.

[31] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, March 2010.

[32] K. Makris and K. D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *EuroSys*, pages 327–340, 2007.

[33] T. Moseley, V. Tovinkere, R. Ramanujan, D. A. Connors, and D. Grunwald. Loopprof: Dynamic techniques for loop detection and profiling. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

[34] J. Newsome, D. Brumley, and D. X. Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *Proceedings of the Network and Distributed System Security Symp osium, NDSS 2006, San Diego, California, USA*, February 2006.

[35] J. Newsome and D. Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2005.

[36] A. One. Smashing the stack for fun and profit. *Phrack*, 7(49), 1996.

[37] T. Raffetseder, C. Krügel, and E. Kirdahttp. Detecting system emulators. In *Information Security, 10th International Conference, ISC 2007*, 2007.

[38] 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.

[39] T. Reps and G. Balakrishnan. Improved memory-access analysis for x86 executables. In *CC'08/ETAPS'08: Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, 2008.

[40] Secunia. DEP/ASLR implementation progress in popular third-party windows applications. White paper, `http://secunia.com/gfx/pdf/DEPASLR2010paper.pdf`, June 2010.

[41] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security (CCS'04)*, 2004.

[42] A. Slowinska, T. Stancescu, and H. Bos. Precise data structure excavation. Technical Report IR-CS-55, Vrije Universiteit Amsterdam, February 2010.

[43] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: automatic reverse engineering of encrypted messages. In *ESORICS'09: Proceedings of the 14th European conference on Research in computer security*, 2009.

[44] G. Wondracek, P. Milani Comparetti, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[45] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. Paricheck: an efficient pointer arithmetic checker for c programs. In *ASIACCS '10: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 145–156, New York, NY, USA, 2010. ACM.

[46] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI '04)*, 2004.

[47] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In *Proceedings of the 8th international conference on Information security applications*, WISA '07, 2007.