# Digging For Data Structures

Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King
*University of Illinois at Urbana-Champaign*

## Abstract

Because writing computer programs is hard, computer programmers are taught to use encapsulation and modularity to hide complexity and reduce the potential for errors. Their programs will have a high-level, hierarchical structure that reflects their choice of internal abstractions. We designed and forged a system, Laika, that detects this structure in memory using Bayesian unsupervised learning. Because almost all programs use data structures, their memory images consist of many copies of a relatively small number of templates. Given a memory image, Laika can find both the data structures and their instantiations.

We then used Laika to detect three common polymorphic botnets by comparing their data structures. Because it avoids their code polymorphism entirely, Laika is extremely accurate. Finally, we argue that writing a data structure polymorphic virus is likely to be considerably harder than writing a code polymorphic virus.

## 1 Introduction

System designers use abstractions to make building complex systems easier. Fixed interfaces between components allow their designers to innovate separately, reduce errors, and construct the complex computer systems we use today. The best interfaces provide exactly the right amount of detail, while hiding most of the implementation complexity.

However, no interface is perfect. When system designers need additional information they are forced to bridge the gap between levels of abstraction. The easiest, but most brittle, method is to simply hard-code the mapping between the interface and the structure built on top of it. Hard-coded mappings enable virtual machine monitor based intrusion detection [13, 22] and discovery of kernel-based rootkits using a snapshot of the system memory image [29]. More complicated but potentially more robust techniques infer details by combining general knowledge of common implementations and runtime probes. These techniques allow detection of OS-level processes from a VMM using CPU-level events [18, 20], file-system-aware storage systems [19, 32], and storage-aware file systems [28]. Most of these techniques work because the interfaces they exploit can only be used in a very limited number of ways. For example, only an extremely creative engineer would use the CR3 register in an x86 processor for anything other than process page tables.

The key contribution of this paper is the observation that even more general interfaces are used often by programmers in standard ways. Because writing computer programs correctly is so difficult, there is a large assortment of software engineering techniques devoted to making this process easier and more efficient. Ultimately most of these techniques revolve around the same ideas of abstraction and divide-and-conquer as the original interfaces. Whether this is the only way to create complex systems remains to be seen, but in practice these ideas are pounded into prospective programmers by almost every text on computer science, from *The Art of Computer Programming* to the more bourgeoisie *Visual Basic for Dummies*.

We chose to exploit a small piece of this software engineering panoply, the compound data structure. Organizing data into objects is so critical for encapsulation and abstraction that even programmers who do not worship at the altar of object-oriented programming usually use a significant number of data structures, if only to implement abstract data types like trees and linked lists. Therefore we can expect the memory image of a process to consist of a large number of instantiations of a relatively small number of templates.

This paper describes the design and implementation of a system – which we named Laika in honor of the Russian space dog – for detecting those data structures given a memory image of the program. The two key

challenges are identifying the positions and sizes of objects, and determining which objects are similar based on their byte values. We identify object positions and sizes by using potential pointers in the image to estimate object positions and sizes. We determine object similarity by converting objects from sequences of raw bytes into sequences of semantically valued blocks: "probable pointer blocks" for values that point into the heap or the stack, "probable string blocks" for blocks that contain null-terminated ASCII strings, and so on. Then, we cluster objects with similar sequences of blocks together using Bayesian unsupervised learning.

Although conceptually simple, detecting data structures in practice is a difficult machine learning problem. Because we are attempting to detect data structures without prior knowledge, we must use unsupervised learning algorithms. These are much more computationally complex and less accurate than supervised learning algorithms that can rely on training data. Worse, the memory image of a process is fairly chaotic. Many malloc implementations store chunk information inside the chunks, blending it with the data of the program. The heap is also fairly noisy: a large fraction consists of effectively random bytes, either freed blocks, uninitialized structures, or malloc padding. Even the byte/block transformation is error-prone, since integers may have values that "point" into the heap. Despite these difficulties, Laika manages reasonable results in practice.

To demonstrate the utility of Laika, we built a virus checker on top of it. Current virus checkers are basically sophisticated versions of grep [2]. Each virus is identified with a fingerprint, usually a small sequence of instructions. When the virus checker finds that fingerprint in a program, it classifies it as a version of the corresponding virus. Because it is easy to modify the instruction stream of a program in provably correct ways, virus writers have created polymorphic engines that replace one set of instructions with another computationally equivalent one, obfuscating the fingerprints [25]. Most proposals to combat polymorphic viruses have focused on transforming candidate programs into various canonical formats in order to run fingerprint scanners [5, 8, 23].

Instead, our algorithm classifies programs based on their data structures: if an unknown program uses the same data structures as Agobot, it is likely to in fact be a copy of Agobot. Not only does this bypass all of the code polymorphism in current worms, but the data structures of a program are likely to be considerably more difficult to obfuscate than the executable code – roughly compiler-level transformations, rather than assembler-level ones. Our polymorphic virus detector based on Laika is over 99% accurate, while ClamAV, a leading open source virus detector, manages only 85%. Finally, by detecting programs based on completely different features our detector has a strong synergy with traditional code-based virus detectors.

Memory-based virus detection is especially effective now that malware writers are turning from pure mayhem to a greedier strategy of exploitation [1, 12]. A worm that merely replicates itself can be made very simple, to the point that it probably does not use a heap, but a botnet that runs on an infected computer and provides useful (at least to the botnet author) services like DoS or spam forwarding is more complex, and more complexity means more data structures.

## 2 Data Structure Detection

A classifier is an algorithm that groups unknown objects, represented by vectors of features, into semantic classes. Ideally, a classification algorithm is given both a set of correctly classified training data and a list of features. For example, to classify fruit the algorithm might be given the color, weight, and shape of a group of oranges, apples, watermelons, and bananas, and then asked whether a 0.1 kg red round fruit is an apple or a banana. This is called supervised learning; a simple example is the naive Bayes classifier, which learns a probability distribution for each feature for each class from the training data. It then computes the class membership probability for unknown objects as the product of the class feature probabilities and the prior probabilities, and places the object in the most likely class. When we do not have training data, we must fall back to unsupervised learning. In unsupervised learning, the algorithm is given a list of objects and features and directed to create its own classes. Given a basket of fruit, it might sort them into round orange things, round red things, big green things, and long yellow things. Designing a classifier involves selecting features that expose the differences between items and algorithms that mirror the structure of the problem.

### 2.1 Atoms and Block Types

The most important part of designing any classifier is usually selecting the features. Color, shape, and weight will work well for fruit regardless of what algorithm is used, but country of origin will not. This problem is particularly acute for data structure detection, because two objects from the same class may have completely different byte values. Our algorithm converts each machine word (4 on 32-bit machines, 8 bytes on 64-bit machines) into a *block type*. The basic block types are address (points into heap/stack), zero, string, and data (everything else). This converts objects from vectors of bytes

| Address | Value | Char Value | Block |
|---|---|---|---|
| 0x650000 | 0x20 | "!" | **D** |
| 0x650008 | 0x0 | "\0" | **0** |
| 0x650010 | 0x650028 | "\FS\0e" | **A** |
| 0x650018 | 0x650088 | "\^\0e" | **A** |
| 0x650020 | 0x20 | "!" | **D** |
| 0x650028 | 0x650008 | "\BS\0e" | **A** |
| 0x650030 | 0x650048 | "0\0e" | **A** |
| 0x650038 | 0x650068 | "h\0e" | **A** |
| 0x650040 | 0x20 | "!" | **D** |
| 0x650048 | 0x650028 | "\FS\0\e" | **A** |
| 0x650050 | 0x0 | "\0" | **0** |
| 0x650058 | 0x650068 | "h\0e" | **A** |
| 0x650060 | 0x20 | "!" | **D** |
| 0x650068 | 0x6873696620656E6F | "one fish" | **S** |
| 0x650070 | 0x6966206F7774202C | ", two fi" | **S** |
| 0x650078 | 0x20646572202C6873 | "sh, red " | **S** |
| 0x650080 | 0x20 | "!" | **D** |
| 0x650088 | 0x6C62202C68736966 | "fish, bl" | **S** |
| 0x650090 | 0x2E68736966206575 | "ue fish." | **S** |
| 0x650098 | 0x56700 | "\0g\ENQ" | **D** |
| 0x6500A0 | 0x40 | "A" | **D** |

**Class 1**

Composition

| Class 1* |
|---|
| Class 1* |
| Class 2* |
| Integer |

| Address | Array? | Blocks |
|---|---|---|
| 0x650008 | No | 0AAD |
| 0x650028 | No | AAAD |
| 0x650048 | No | A0AD |

**Class 2**

Composition

| String |
|---|

| Address | Array? | Blocks |
|---|---|---|
| 0x650068 | Yes; x3 | SSSD |
| 0x650088 | Yes; x2 | SSDD |

Figure 1: An example of data structure detection. On the left is a small segment of the heap, and on the right is Laika's output. Class 1, in rows 2-13, is a doubly linked list of C strings; the first two elements are pointers to other elements of Class 1. The fourth element is actually internal malloc data on the size of the next chunk. Laika estimates the start of the next object as the end of the first, which is 8 bytes too long. Class 2 contains two C null-terminated character arrays. A real heap sample is much noisier; in the programs we looked at, less than 50% of the heap was occupied by active objects; the rest was a mix of freed objects, malloc padding, and unallocated chunks.

into vectors of block types, and we can expect the block type vectors to be similar for objects from the same class.

Classes are represented as vectors of *atomic types*. Each atomic type roughly corresponds to one block type (e.g., pointer → address and integer → data), but there is some margin for error since the block type classification is not always accurate; some integer or string values may point into the heap, some pointers may be uninitialized, a programmer may have used a union, and so on. This leads to a probability array $p(blocktype|atomictype)$ where the largest terms are on the diagonal, but all elements are nonzero. While these probabilities will not be exactly identical for individual applications, in our experience they are similar enough that a single probability matrix suffices for most programs. In the evaluation we measured $p(blocktype|atomictype)$ for several programs; all of them had strongly diagonal matrices. There is also a random atomic type for blocks that lie between objects. Figure 1 shows an example of what memory looks like when mapped to block and atomic types.

Although the block type/atomic type system allows us to make sense of the otherwise mystifying bytes of a memory image, it does have some problems. It will miss unaligned pointers completely, since the two halves of the pointer are unlikely to point at anything useful, and it will also miss the difference between groups of integers, for example four 2-byte integers vs. one 8-byte integer. In our opinion these problems are relatively minor, since unaligned pointers are quite rare, and it would be extremely difficult to distinguish between four short ints and one long int anyway. Lastly, if the program occupies a significant fraction of its address space there will be many integer/pointer mismatches, but as almost all new CPUs are 64-bit, the increased size of a 64-bit address space will reduce the probability of false pointers.

## 2.2 Finding Data Structures

Unlike the idyllic basket of fruit, it is not immediately obvious where the objects lurk in an image, but we can estimate their positions using the targets of the pointers. Our algorithm scans through a memory image looking for all pointers, and then tentatively estimates the positions of objects as those addresses referenced in other places of the image.

Although pointers can mark the start of an object, they

seldom mark the end. Laika estimates the sizes of objects during clustering. Each object's size is bounded by the distance to the next object in the address space, and each class has a fixed size, which is no larger than its smallest object. If an object is larger than the size of its class, its remaining blocks are classified as random noise. For this purpose we introduce the random atomic type, which generates all block types more or less equally. In practice some objects might be split in two by internal pointers - pointers to the interior of malloc regions. At the moment we do not merge these objects.

## 2.3 Exploiting Malloc

It should be possible for Laika to find objects more accurately when armed with prior information about the details of the malloc library. For example, the Lea allocator, used in GNU libc, keeps the chunk size field inlined at the top of the chunk as shown in figure 1. Unfortunately there is sufficient variety in malloc implementations to make this approach tedious. Aside from separate malloc implementations for Windows and Linux, there are many custom allocators, both for performance and other motivations like debugging.

Early versions of Laika attempted to exploit something that most malloc implementations should share: address space locality. Most malloc implementations divide memory into chunks, and chunks of the same size often lie in contiguous regions. Since programs exhibit spatial locality as well, this means that an object will often be close to one or more objects of the same type in memory; in the applications we measured over 95% of objects had at least one object of the same class within their 10 closest neighbors. Despite this encouraging statistic, the malloc information did not significantly change Laika's classification accuracy. We believe this occurs because Laika already knows that nearby objects are similar due to similar size estimations.

## 2.4 Bayesian Model

Bayesian unsupervised learning algorithms compute a joint probability over the classes and the classification, and then select the most likely solution. We represent the memory image by $\boldsymbol{M}$, where $\boldsymbol{M}_l$ is the $l$th machine word of the memory image, $\boldsymbol{\omega}$ for the list of classes, where $\boldsymbol{\omega}_{jk}$ is the $k$th atomic type of class $j$, and $\boldsymbol{X}$ for the list of objects, where $\boldsymbol{X}_i$ is the position of the $i$th object. We do not store the lengths of objects, since they are implied by the classification. Our notation is summarized in Table 1.

We wish to estimate the most likely set of objects and classes given a particular memory image. With Bayes's rule, this gives us:

$$p(\boldsymbol{\omega}, \boldsymbol{X}|\boldsymbol{M}) = \frac{p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})p(\boldsymbol{X}|\boldsymbol{\omega})p(\boldsymbol{\omega})}{p(\boldsymbol{M})} \quad (1)$$

Since Laika is attempting to maximize $p(\boldsymbol{\omega}, \boldsymbol{X}|\boldsymbol{M})$, we can drop the normalizing term $p(\boldsymbol{M})$ , which does not affect the optimum solution, and focus on the numerator. $p(\boldsymbol{\omega})$ is the prior probability of the class structure. To simplify the mathematics, we assume independence both between and within classes, even though this assumption is not really accurate (classes can often contain arrays of basic types or even other objects). We let $\boldsymbol{\omega}_{jk}$ be the $k$th element of class $j$, which lets us simply multiply out over all such elements:

$$p(\boldsymbol{\omega}) = \prod_j \prod_k p(\boldsymbol{\omega}_{jk}) \quad (2)$$

$p(\boldsymbol{X}|\boldsymbol{\omega})$ is the probability of the locations and sizes of the list of objects, based on our class model and what we know about data structures. This term is 0 for illegal solutions where two objects overlap, and 1 otherwise. $p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})$ represents how well the model fits the data. When the class model and the instantiation list are merged, they predict a set of atomic data types (including the random "type") that cover the entire image. Since we know the real block type $\boldsymbol{M}_l$, we can compute the probability of each block given classified atomic type:

$$p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X}) = \prod_l p(\boldsymbol{M}_l|\boldsymbol{\omega}, \boldsymbol{X}) \quad (3)$$

When $p(\boldsymbol{\omega})$, $p(\boldsymbol{X}|\boldsymbol{\omega})$, and $p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})$ are multiplied together, we finally get a master equation which we can use to evaluate the likelihood of a given solution. Although the master equation is somewhat formidable, the intuition is very simple; $p(\boldsymbol{M}|\boldsymbol{\omega}, \boldsymbol{X})$ penalizes Laika whenever it places an object into an unlikely class, and ensuring that the solution reflects the particular memory image, while $p(\boldsymbol{\omega})$ enforces Occam's razor by penalizing Laika whenever it creates an additional class, thus causing it to prefer simpler solutions.

## 2.5 Typed Pointers

While the simple pointer/integer classification system already produces reasonable results, a key optimization is the introduction of typed pointers. If all of the instances of a class have a pointer at a certain offset, it is probable that the targets of those pointers are also in the same class. As the clustering proceeds and the algorithm becomes more confident of the correct clustering, it changes the address blocks to typed address blocks based on the class of their target. Typed pointers are especially important for small objects, because the class is smaller

| | |
|---|---|
| atomic type | A machine level type, like a pointer. |
| block type | Value of an atomic type |
| data structure | "struct 1". Compound type. |
| $\boldsymbol{X}_i$ | instantiation / object $i$ |
| $i$ | index of objects |
| $\boldsymbol{\omega}_j$ | class / compound data type $j$ |
| $j$ | index of classes |
| $k$ | block offset within a class /object |
| $\boldsymbol{M}$ | the memory image of our process |
| $l$ | index with a memory image |

Table 1: Terms and symbols

and inherently less descriptive, and objects that contain no pointers, which can sometimes be accurately grouped solely by their references. Since it is impossible to measure the prior and posterior probabilities for the classes and pointers of an unknown program, we simply measured the probability that a typed pointer referred to a correctly typed address or an incorrectly typed address. Typed pointers greatly increase the computational complexity of the equation, because the classification of individual objects is no longer independent if one contains a pointer to the other. Worse, when Laika makes a mistake, the typed pointers will cause this error to propagate. Since typed pointer mismatches are weighted very heavily in the master equation, Laika may split the classes that reference the poorly classified data structure as well.

### 2.6 Dynamically-Sized Arrays

The second small speed bump is the dynamically-sized array. In standard classification, all elements in a class have feature vectors of the same size. Obviously this is not true with data structures, with the most obvious and important being the ubiquitous C string. We handle a dynamic array by allowing objects to "wrap around" modulo the size of a class. In other words, we allow an object to be classified as a contiguous set of instantiations of a given class - an array.

## 3 Implementation

We implemented Laika in Lisp; the program and its testing tools total about 5000 lines, including whitespace and comments. The program attempts to find good solutions to the master equation when given program images.

Unfortunately our master equation is computationally messy. Usually, unsupervised learning is difficult, while supervised learning is simple: each item is compared against each class and placed in the class that gives the least error. But with our model, if $\boldsymbol{X}_1$ contains a pointer to $\boldsymbol{X}_2$, the type of $\boldsymbol{X}_2$ will affect the block type and therefore the classification error of $\boldsymbol{X}_1$, so even an exact supervised solution is difficult. Therefore our only choice is to rely on an approximation scheme. Laika computes $p(\boldsymbol{\omega}, \boldsymbol{X}|\boldsymbol{M})$ incrementally and uses heuristics to decide which classification changes (e.g., move $\boldsymbol{X}_{233}$ from $\boldsymbol{\omega}_{17}$ to $\boldsymbol{\omega}_{63}$). We leverage typed pointers to compute reasonable changes. For example, whenever an object is added to a class that contains a typed pointer, it tries to move the pointer targets of that object into the appropriate class.

## 4 Data Structure Detection

Measuring Laika's ability to successfully identify data structures proved surprisingly difficult. Because the programs we measured are not typesafe, there is no way to determine with perfect accuracy the types of individual bytes. The problems begin with unions, continue with strange pointer accesses, and climax with bugs and buffer overflows. Fortunately these are not too common, and we believe our ground truth results are mostly correct.

We used Gentoo Linux to build a complete set of applications and libraries compiled with debugging symbols and minimal optimizations. We then ran our test programs with a small wrapper for *malloc* which recorded the backtrace of each allocation, and used GDB to obtain the corresponding source lines and guesses at assignment variables at types. Because of the convoluted nature of C programs, we manually checked the results and cleaned up things like macros, typedefs, and parsing errors.

Our model proved mostly correct: less than 1% of pointers are unaligned, and only 1% of integers and 3% of strings point into the heap. About 80% of pointers point to the head of objects. Depressingly, the heap is extremely noisy: on average only 45% of a program's heap address space is occupied by active objects, with the rest being malloc padding and unused or uninitialized chunks. Even more depressingly, only 30% of objects contain a pointer. Since Laika relies on building "pointer fingerprints" to classify objects, this means that the remaining 70% of objects are classed almost entirely by the objects that point to them.

We also encountered a rather disturbing number of poor software engineering practices. Several key X Window data structures, as well as the Perl Compatible Regular Expressions library used by Privoxy, use the dreaded *tail accumulator array*. This archaic programming practice appends a dynamically sized array to a fixed structure; the last element is an array of size 0 and each call to malloc is padded with the size of the array. Although this saves a call to malloc, it makes the software much harder to maintain. This hampers our results on all of the X Window applications, because Laika assumes that

| Name | Objects | Classes | $p(real\|laika)$ | $p(laika\|real)$ | $p_{obj}(real\|laika)$ | $p_{obj}(laika\|real)$ |
|---|---|---|---|---|---|---|
| blackhack | 215 | 6 | 0.87 | 1.00 | 0.87 | 1.00 |
| xeyes | 680 | 17 | 0.66 | 0.68 | 0.74 | 0.93 |
| ctorrent | 295 | 19 | 0.61 | 0.67 | 0.60 | 0.70 |
| privoxy | 3881 | 32 | 0.90 | 0.71 | 0.93 | 0.82 |
| xclock | 2422 | 54 | 0.62 | 0.44 | 0.72 | 0.38 |
| xpdf | 16846 | 180 | 0.61 | 0.57 | 0.64 | 0.56 |
| xarchiver | 20993 | 315 | 0.52 | 0.49 | 0.60 | 0.60 |
| **Average** | **6476** | **89** | **0.68** | **0.65** | **0.73** | **0.71** |
| blackhack-wm | 201 | 8 | 0.96 | 1.00 | 0.96 | 1.00 |
| ctorrent-wm | 249 | 13 | 0.80 | 0.66 | 0.78 | 0.73 |
| xeyes-wm | 526 | 22 | 0.83 | 0.67 | 0.79 | 0.95 |
| privoxy-wm | 3615 | 32 | 0.92 | 0.71 | 0.90 | 0.88 |
| xclock-wm | 2197 | 43 | 0.72 | 0.58 | 0.79 | 0.56 |
| xarchiver-wm | 7501 | 89 | 0.77 | 0.62 | 0.80 | 0.66 |
| xpdf-wm | 12995 | 194 | 0.63 | 0.62 | 0.69 | 0.64 |
| **Average-wm** | **3898** | **57** | **0.80** | **0.70** | **0.82** | **0.77** |

Table 2: Data Structure Detection Accuracy. The first part of the table shows Laika's accuracy using only the memory image; the second part using the memory image and a list of the sizes and locations of objects. $p_{obj}(real|laika)$ and $p_{obj}(laika|real)$ are the accuracy when known atomic types like *int* or *char* are ignored.

all data structures have a fixed length. To express our appreciation, we sent the X Window developers a dirty sock.

We concentrated on the classification accuracy, the chance that Laika actually placed objects from the same real classes together, as opposed to the block accuracy, the chance that Laika generated correct compositions for its classes, because it is more relevant to correctly identifying viruses. There are two metrics: the probability that two objects from the same Laika class came from the same real class, $p(real|laika)$, and the probability that two objects from the same real class were grouped together, $p(laika|real)$. It is easy to see that the first metric could be satisfied by placing all elements in their own classes, while the second could be satisfied by placing all elements in the same class. Table 2 summarizes the results.

Laika is reasonably accurate but far from perfect, especially on larger programs. The first source of error is data objects (like strings or int arrays), which are difficult to classify without pointers; even some of the real objects contain no pointers. A more interesting problem arises from the variance in size: some classes contain many more objects than others. Generally speaking, Laika merges classes in order to increase the class prior probability $p(\boldsymbol{\omega})$ and splits them in order to increase the the image posterior probability $p(M|\boldsymbol{\omega}, \boldsymbol{X})$. Because the prior has the same weight as a single object, merging two classes that contain many objects will have a much larger effect on the posterior than the prior. For example, Laika may split a binary tree into an internal node class and a leaf node class, because the first would have two address blocks and the second two zero blocks. If there are some 10 objects, then the penalty for creating the additional class would outweigh the bonus for placing the leaf nodes in a more accurate class, but if there are 100 objects Laika is likely to use two classes.

This data also shows just how much Laika's results improve when the random data, freed chunks, and malloc information are removed from the heap. Not only does this result in the removal of 20% of the most random structures, but it also removes malloc padding, which can be uninitialized. Without size information, Laika can easily estimate the size of an object as eight or more times the correct value when there is no pointer to the successor chunk.

To avoid too much mucking around in Laika's dense output files, we can generate relational graphs of the data structures detected. Figure 2 shows a graph of the types discovered by Laika for the test application *Privoxy* without the aid of location information; Figure 3 shows the correct class relationships. Edges in the graph represent pointers; **s1** will have an edge to **s2** if **s1** contains at least one element of type **s2***. We filtered Figure 2 to remove unlikely classes and classes which had no pointers. We are able to easily identify the all the correct matching classes (shown shaded) from the given graph, as well as a few classes that were incorrectly merged by Laika

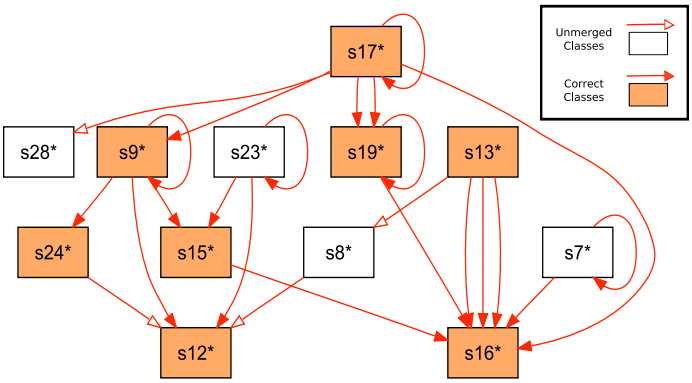


Figure 2: Laika generated type graph for Privoxy

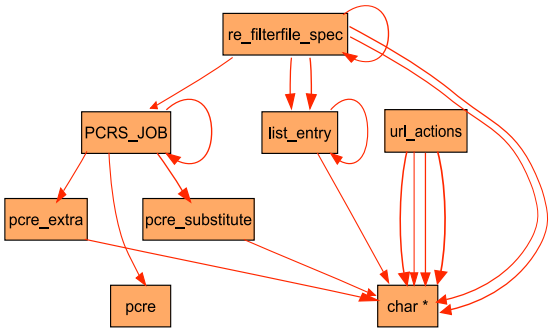


Figure 3: Correct type graph for Privoxy

(shown in white). For instance, type **s17** corresponds directly to the **re_filterfile_spec** structure, whereas types **s9** and **s23** are, in fact, of the same type and should be merged. In addition, by graphing the class relationship in this manner, visually identifying common data structures and patterns is quite simple. For example, a single self-loop denotes the presence of a singly linked list, as shown by the class **s19/list_entry**.

## 5 Program Detection

Because classifying programs by their data structures already removes all of their code polymorphism, we chose to keep the classifier itself simple. Our virus detector merely runs Laika on the memory images of two programs at the same time and measures how often objects from the different images are placed in the same classes; if many classes contain objects from both programs the programs are likely to be similar. Mathematically, we measured the mixture rate $P(image_i = image_j | class_i = class_j)$ for all object pairs $(i, j)$, which will be closer to 0.5 for similar programs and to 1.0 for dissimilar programs.

Aside from being easy to implement, this approach has several interesting properties. Because Laika is more accurate when given more samples of a class, it is able to discover patterns in the images together that it would miss separately. The mixture ratio also detects changes to the frequency with which the data structures are used. But most importantly, this approach leverages the same object similarity machinery that Laika uses to detect data structures. When Laika makes errors it will tend to make the same errors on the same data, and the mixture ratio focuses on the more numerous - and therefore more accurate - classes, which means that Laika can detect viruses quite accurately even when it does not correctly identify all of their data structures. To focus on the structures that we are more likely to identify correctly, we removed classes that contained no pointers and classes that had very low probability according to the master equation from the mixture ratio. The main disadvantage of the mixture ratio is that the same program can produce different data structures and different ratios when run with different inputs.

We obtained samples of three botnets from Offensive Computing. Agobot/Gaobot is a family of bots based on GPL source released in 2003. The bot is quite object oriented, and because it is open source there are several thousand variants in the wild today. These variants are often fairly dissimilar, as would-be spam lords select different modules, add code, use different compilers and so on. Agobot also contains some simple polymorphic routines. Storm is well known, and its authors have spent considerable effort making it difficult to detect. Kraken, also known as Bobax, has taken off in the spring of 2008. It is designed to be extremely stealthy and, according to some estimates, is considerably larger than Storm [15]. We ran the bots in QEMU to defeat their VM detection and took snapshots of their memory images using WinDbg.

Our biggest hurdle was getting the bots to activate. All of our viruses required direction from a command and control server before they would launch denial of service attacks or send spam emails. For Agobot this was not a problem, as Agobot allocates plenty of data structures on startup. Our Kraken samples were apparently slightly out of date and spent all their time trying to connect to a set of pseudo-random DNS addresses to ask for instructions; most of the data structures we detected are actually allocated by the Windows networking libraries. Our Storm samples did succeed in making contact with the command servers, but this was not an unmixed blessing as their spam emails brought unwanted attention from our obstreperous network administrators. To this day we

| Bot | $\mu_{virus}$ | $\sigma_{virus}$ | $\mu_{other}$ | $\sigma_{other}$ | Threshold | Samples | Errors | Est. Accuracy | ClamAV |
|---|---|---|---|---|---|---|---|---|---|
| Agobot | 0.64 | 0.038 | 0.89 | 0.053 | 0.75 | 19/27 | 0/0 | 99.4% | 83% |
| Kraken | 0.52 | 0.021 | 0.83 | 0.078 | 0.58 | 34/27 | 0/0 | 99.8% | 85% |
| Storm | 0.51 | 0.005 | 0.60 | 0.015 | 0.53 | 20/20 | 0/0 | 99.9% | 100% |

Table 3: Classification Results. For example, we used 17 of our 34 Kraken samples as training data. When the remaining 16 samples were compared against the signature, they produced an average mixture ratio of 0.52 with a standard deviation of 0.021. Of our 27 clean images, we used 13 as training data, and those produced an average mixture ratio with our Kraken sample of 0.83 with a standard deviation of 0.078. The resulting maximum likelihood classifier classifies anything with a mixture ratio of less than 0.58 as Kraken, and has an estimated accuracy of 99.8%; it classified the remaining 17 Kraken samples and 14 standard Windows applications without error. If a new sample was compared with the Kraken signature and produced a mixture rate of 0.56, it would be classified as Kraken, being $1.9\sigma$ from the average of the Kraken samples and $3.5\sigma$ from the average of the normal samples.

curse their names, especially those who are still alive.

These three botnets represent very different challenges for Laika. Agobot is not merely polymorphic; because it is a source toolkit there are many different versions with considerable differences, while Kraken is a single executable where the differences come almost entirely from the polymorphic engine. Agobot is written in a style reminiscent of MFC, with many classes and allocations on the heap. We think Kraken also has a considerable number of data structures, but in the Kraken images we analyzed the vast majority of the objects were from the Windows networking libraries. This means that the Agobot images were dissimilar to each other owing to the many versions, and also to regular Windows programs because of their large numbers of custom data structures, while the Kraken images were practically identical to each other, but also closer to the regular Windows programs. Neither was Laika's ideal target, which would be a heavily object oriented bot modified only by polymorphic routines. Storm was even worse; by infecting a known process its data structures blended with those of services.exe.

Our virus detector works on each botnet separately; a given program is classified as Kraken or not, then Agobot or not, and so on. We used a simple maximum likelihood classifier, with the single parameter being the mixture ratio between the unknown program and a sample of the virus, which acts as the signature. In such a classifier, each class is represented by a probability distribution; we used a Gaussian distribution as the mixture ratio is an average of the individual class mixture ratios. An unknown sample is placed in the most likely class, i.e. the class whose probability distribution has the greatest value for the mixture ratio of that sample. For a Gaussian distribution, this can be thought of as the class that is closest to the sample, where the distance is normalized by standard deviation.

We used half of our samples as training data to estimate the virus and normal distributions. For normal programs we used 27 standard Windows applications including bittorrent, Skype, Notepad, Internet Explorer, and Firefox. To select the signature from the training set, we tried all of them and chose the one with the lowest predicted error rate. Table 3 summarizes the results. The detector takes from 3 seconds for small applications up to 20 minutes for large applications like Firefox.

Because Storm injects itself into a known process, we had the opportunity to treat it a little differently. We actually used a clean services.exe process as the "virus"; the decision process is reversed and any sample not close enough to the signature is declared to be Storm. This is considerably superior to using a Storm sample, because our Storm images were much less self-similar than the base services.exe images.

## 5.1 Discussion

The estimated accuracy numbers represent the self-confidence of the model, specifically the overlap of the probability distributions, not its actual tested performance. We included them to give a rough estimate of Laika's performance in lieu of testing several thousand samples. They do not reflect the uncertainty in the estimation of the mean and variance (from some 10-15 samples), which is slightly exacerbated by taking the most discriminatory sample, nor how well the data fit or do not fit our Gaussian model.

It is interesting to note that the accuracy numbers for Kraken and Agobot are roughly comparable despite Agobot containing many unique structures and Kraken using mainly the Windows networking libraries. This occurs because our Kraken samples were extremely similar to one another, allowing Laika could use a very low classification threshold. It is also worth noting that while 99% seems very accurate, a typical computer contains far more than 100 programs.

It would be fairly straightforward to improve our rude 50-line classifier, but even a more complicated version would compare favorably with ClamAV's tens of thousands of lines of code. ClamAV attempts to defeat polymorphic viruses by unpacking and decrypting the hidden executable; this requires a large team of reverse engineers to decipher the various polymorphic methods of thousands of viruses and a corresponding block of code for each.

## 5.2 Analysis

Since our techniques ignore the code polymorphism of current botnets, it is reasonable to ask whether new "memory polymorphic" viruses will surface if data structure detection becomes common. Because virus detection is theoretically undecidable, such malware is always possible, and the best white hats can do is place as many laborious obstacles as possible in the path of their evil counterparts. We believe that hiding data structures is qualitatively more difficult than fooling signature-based detectors, and in this section we will lay out some counter and counter-counter measures to Laika. Our argument runs in two parts: that high-level structure is harder to obfuscate than lower level structure, and that because high-level structure is so common in programs, we can be very suspicious of any program that lacks it.

Most of the simplest solutions to obfuscating data structures simply eliminate them. For example, if every byte was XORed with a constant, all of the data structures would disappear. While the classifier would have nothing to report, that negative report would itself be quite damning, although admittedly not all obfuscated programs are malware. Even if the objects themselves were obfuscated, perhaps by appending a large amount of random pointers and integers to each, the classifier would find many objects but no classes, which again would be quite suspicious. Slightly more advanced malware might encrypt half of the memory image, while creating fake data structures from a known good program in the other half. Defeating this might require examining the instruction stream and checking for pointer encryption, i.e. what fraction of pointers are used directly without modification.

To truly fool Laika, a data structure polymorphic virus would need to actually change the layout of its data structures as it spreads. It could do this, perhaps, by writing a compiler that shuffled the order of the fields of all the data structures, and then output code with the new offsets. It is obvious that this kind of polymorphism is much more complicated than the kind of simple instruction insertion engines we see today, requiring a larger payload and increasing the chance that the virus would be enervated by its own bugs. The other option would be to fill the memory image with random data structures and hope that the real program goes undetected in the noise. This increases the memory footprint and reduces the stealthiness of the bot, and has no guarantee of success, since the real data structures are still present. Detecting such viruses would probably require a more complicated descendant of Laika with a more intelligent classifier than the simple mixture ratio.

The greatest advantage of Laika as a virus detector is its orthogonality with existing code-based detectors. At worst, it provides valuable defense in depth by posing malware authors different challenges. At best, it can synergize with code analysis; inspecting the instruction stream may reveal whether a program is obfuscating its data structures.

There are two primary disadvantages to using high-level structure. First, a large class of malware has no high-level structure by the virtue of simplicity. A small kernel rootkit that overwrites a system call table or a bit of shellcode that executes primarily on the stack won't use enough data structures to be detected, and distinguishing a small piece of malware inside a large program like Apache or Linux is more difficult than detecting it in a separate process. However, we believe that there is an important difference between fun and profit. Turning zombie machines into hard currency means putting them to some purpose, be it DoS attacks, spam, serving ads, or other devilry, and that means running some sort of moderately complex process on the infected machines. It is this sort of malware that has been more common lately [1, 12], and it is this sort of malware that we aim to detect.

The second main disadvantage is the substantial increase in resources, in both memory and processor cycles, when compared to current virus scanners. Although a commercial implementation would no doubt be more highly optimized, we do not believe that order of magnitude improvements are likely given the computational complexity of the equations to be solved. Moreover, our data structure detection algorithms apply only to running processes. Worse, those processes will not exercise a reasonable set of their data structures unless they are allowed to perform their malicious actions, so a complete solution must provide a way to blunt any malicious effects prior to detection. This requires either speculatively executing all programs and only committing output after a data structure inspection, or recording all output and rolling back malicious activity. Although we believe that Moore's law will continue to provide more resources, these operations are still not cheap.

### 5.3 Scaling

Laika runs in linear time in the number of viruses, with moderately high constants. Clearly a commerical version of Laika would require some heuristical shortcuts to avoid stubbornly comparing a test image against some $10^5$ virus signature images. The straightforward approach would be to run Laika on the unknown program, record the data structures, compute a signature based on those data structures, use that signature to look up a small set of similar programs in a database, and verify the results with the mixture ratio.

It turns out this is not completely straightforward after all. Consider the most natural approach. The list of data structures may be canonicalized by assigning each structure a unique id. Typed pointer issues can be avoided by assigning a structure a pointer only after canonicalizing the structures to which it points. A program image could then be represented as a vector of structure counts and confidences, and the full mixture ratio computed only for the $k$ nearest neighbors under some distance metric. This approach is extremely brittle. Imagine we have a data structure where one field is changed from 'pointer' to 'zero'. This not only changes the id of that data structure, but also all structures that point to it. In addition, the vector would lie in the space of all known data structures, which would make it hundreds of thousands of elements long. We believe that these difficulties could be solved by a bit of clever engineering, but we have not actually tried to do so.

## 6 Related Work

Most of the work on the semantic gap so far has come from the security community, which is interested in detecting viruses and determining program behavior by "looking up" from the operating system or VMM towards the actual applications. While most of these problems are either extremely computationally difficult or undecidable in theory, there are techniques that work in practice.

### 6.1 The Unobfuscated Semantic Gap

Recently, many researchers have proposed running operating system services in separate virtual machines to increase security and reliability. These separated services must now confront the semantic gap between the raw bytes they see and the high-level primitives of the guest OS, and most exploit the fixed interfaces of the processor and operating system to obtain a higher level view, a technique known as virtual machine introspection. Antfarm [18] monitors the page directory register (CR3 in x86) to infer process switches and counts, while Geiger [19] monitors the the page table and through it can make inferences about the guest OS's buffer cache. Intrusion detectors benefit greatly from the protective isolation of a virtual machine [6, 13]; most rely on prior information on some combination of the OS data structures, filesystem format, and processor features. Polishchuk *et al.* [30] also attempt to determine heap types, but they do so from a supervised environment, where exact malloc locations and debug information are available, making the problem considerably easier.

### 6.2 The Obfuscated Semantic Gap

Randomization and obfuscation have been used by both attackers and defenders to make bridging the semantic gap more difficult. Address space randomization [4], now implemented in the Linux kernel, randomizes the location of functions and variables; buffer overflows no longer have deterministic side effects and usually cause the program to crash rather than exhibit the desired malicious behavior.

On the other side, virus writers attempt to obfuscate their programs to make them more difficult to disassemble [25]. Compiler-like code transformations such as nop or jump insertion, reording, or instruction substitution [21] are relatively straightforward to implement and theoretically extremely effective: universal virus classification is undecidable [9] and even detecting whether a program is an obfuscated instance of a polymorphic virus is NP-Complete [7]. Even when the virus writer does not explicitly attempt to obfuscate the program, new versions of existing viruses may prove effectively polymorphic [16].

### 6.3 The Deobfuscated Semantic Gap

Polymorphic virus detectors usually fall into two classes: code detectors and behavior detectors. For example, Christodorescu *et. al* [8] attempt to detect polymorphic viruses by defining patterns of instructions found in a polymorphic worm and searching for them in the unknown binary. Unlike a simple signature checker, these patterns are fairly general and their virus scanner uses a combination of nop libraries, theorem proving and randomized execution. In the end it is capable of detecting instruction (but not memory) reordering, register renaming, and garbage insertion. A recent survey by Singh and Lakhotia [31] gives a good summary of this type of classifier. In addition, Ma *et. al* [26] attempt to classify families of code injection exploits. They use emulated execution to decode shellcode samples and cluster results on executed instruction edit distance. Their results show success in classifying small shellcode samples, but they

rely entirely on prior knowledge of specific vulnerabilities to locate and extract these samples.

The second class of detectors concentrate on behavior rather than fingerprinting. These methods usually have either a group of heuristics for malicious behavior [24] or statistical thresholds [11]. Often they concentrate on semantically higher level features, like system calls or registry accesses rather than individual instructions. Because they are not specific to any particular virus, they can detect unknown viruses, but they often suffer from false positives since benign executables can be very similar to malicious ones.

### 6.4 Shape Analysis

Modern compilers spend a great deal of time trying to untangle the complicated web of pointers in C programs [33]. Many optimizations cannot be performed if two different pointers in fact refer to the same data structure, and answering this question for structures like trees or hash tables can be difficult. Shape analysis [10, 14] attempts to determine the high-level structure of a program: does a set of allocations form a list, binary tree, or other abstract data type? Although it can enable greater parallelism, shape analysis is very expensive on all but the most trivial programs. Our work attacks the problem of high-level structure from a different angle; although we do not have the source code of the target program, our task is simplified by considering only one memory image.

### 6.5 Reverse Engineering

Reverse engineering is the art of acquiring human meaning from system implementation. However, most of the work in this field is concentrated on building tools to aid humans discover structure from systems [17, 34], rather than using the information directly. Furthermore, a large amount of the reverse engineering literature [27, 35] is concerned with reverse engineering structure from source code to provide developers with high-level understanding of large software projects.

A more superficially similar technique is Value Set Analysis [3]. VSA can be thought of as pointer aliasing analysis for binary code; it tries to determine possible values for registers and memory at various points in the computation. It is especially useful for analyzing malware. Laika differs from VSA in that it is dynamic rather than static, and that Laika's output is directly used to identify viruses rather than aid reverse engineers.

## 7 Conclusions

In this paper we have discussed the design and implementation of Laika, a system that detects the data structures of a process given a memory image. The data structures generated by Laika proved surprisingly effective for virus detection. The vast majority of current polymorphic virus detectors work by generating low level fingerprints, but these fingerprints are easily obfuscated by malware writers. By moving the fingerprint to a higher level of abstraction, we increase the difficulty of obfuscation. Laika also provides valuable synnergy to existing code signature based detectors.

Laika exploits the common humanity of programmers; even very flexible fixed interfaces like Von Neumann machines are often used in standard ways. Because fixed interfaces dominate the landscape in systems - often the interface *is* the system - there should be many other such opportunities.

## 8 Acknowledgments

We would like to thank our shepherd, Geoffrey Voelker, and the anonymous reviewers for their insightful comments and suggestions. We would also like to thank Deepak Ramachandran and Eyal Amir for their help navigating the byzantine labyrinth of machine learning, and Craig Zilles, Matt Hicks, Lin Tan, Shuo Tang, and Chris Grier for reviewing early drafts of this paper. This work was funded by a grant from the Internet Services Research Center (ISRC) of Microsoft Research, and by NSF grant CT-0716768.

## References

[1] 2007 malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code. http://www.computereconomics.com/page.cfm?name=Malware%20Report.

[2] Clamav website. http://www.clamav.org.

[3] BALAKRISHNAN, G., AND REPS, T. Analyzing memory accesses in x86 executables. In *In Comp. Construct* (2004), Springer-Verlag, pp. 5–23.

[4] BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of the 14th USENIX Security Symposium* (Aug. 2005), USENIX.

[5] BRUSCHI, D., MARIGNONI, L., AND MONGA, M. Detecting self-mutating malware using control flow graph matching. Technical Report, Universitaa degli Studi di Milano, http://idea.sec.dico.unimi.it/ lorenzo/rt0906.pdf.

[6] CHEN, P. M., AND NOBLE, B. D. When virtual is better than real. In *HotOS* (2001), IEEE Computer Society, pp. 133–138.

[7] CHESS, D. M., AND WHITE, S. R. An undetectable computer virus. In *Proceedings of the 2000 Virus Bulletin Conference* (2000).

[8] CHRISTODORESCU, M., JHA, S., SESHIA, S. A., SONG, D., AND BRYANT, R. E. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)* (Oakland, CA, USA, may 2005), pp. 32–46.

[9] COHEN, F. Computer viruses: Theory and experiments. In *Computers and Security* (1987), pp. 22–35.

[10] CORBETT, J. C. Using shape analysis to reduce finite-state models of concurrent java programs. *ACM Trans. Softw. Eng. Methodol. 9*, 1 (2000), 51–93.

[11] DENNING, D. E. An intrusion-detection model. *IEEE Trans. Softw. Eng. 13*, 2 (February 1987), 222–232.

[12] FRANKLIN, J., PAXSON, V., PERRIG, A., AND SAVAGE, S. An inquiry into the nature and causes of the wealth of internet miscreants. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 375–388.

[13] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *NDSS* (2003), The Internet Society.

[14] GHIYA, R., AND HENDREN, L. J. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1996), ACM, pp. 1–15.

[15] GOODIN, D. Move over storm - there's a bigger, stealthier botnet in town. http://www.theregister.co.uk/2008/04/07/kraken_botnet_menace/.

[16] GORDON, J. Lessons from virus developers: The beagle worm history through april 24, 2005. In *SecurityFocus Guest Feature Forum* (2004).

[17] HSI, I., POTTS, C., AND MOORE, M. Ontological excavation: Unearthing the core concepts of applications. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)* (2003).

[18] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the 2006 USENIX Annual Technical Conference: May 30–June 3, 2006, Boston, MA, USA* (2006).

[19] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Geiger: monitoring the buffer cache in a virtual machine environment. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 2006), ACM Press, pp. 14–24.

[20] JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Vmm-based hidden process detection and identification using lycosid. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (New York, NY, USA, 2008), ACM, pp. 91–100.

[21] JORDAN, M. Dealing with metamorphism, October 2002.

[22] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 2005 Symposium on Operating Systems Principles (SOSP)* (October 2005), pp. 91–104.

[23] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Polymorphic worm detection using structural information of executables, 2005.

[24] KRUEGEL, C., ROBERTSON, W., AND VIGNA, G. Detecting Kernel-Level Rootkits Through Binary Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (Tucson, AZ, December 2004), pp. 91–100.

[25] LINN, C., AND DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly.

[26] MA, J., DUNAGAN, J., WANG, H. J., SAVAGE, S., AND VOELKER, G. M. Finding diversity in remote code injection exploits. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2006), ACM, pp. 53–64.

[27] MÜLLER, H. A., ORGUN, M. A., TILLEY, S. R., AND UHL, J. S. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice 5(4)* (December 1993), 181–204.

[28] NUGENT, J., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Controlling your place in the file system with gray-box techniques. In *Proceedings of the 2003 USENIX Annual Technical Conference* (June 2003).

[29] PETRONI, N. L., FRASER, T., MOLINA, J., AND ARBAUGH, W. A. Copilot–a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 2004 USENIX Security Symposium* (August 2004).

[30] POLISHCHUK, M., LIBLIT, B., AND SCHULZE, C. W. Dynamic heap type inference for program understanding and debugging. *SIGPLAN Not. 42*, 1 (2007), 39–46.

[31] SINGH, P. K., AND LAKHOTIA, A. Analysis and detection of computer viruses and worms: An annotated bibliography. In *ACM SIGPLAN Notices* (2002), pp. 29–35.

[32] SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F., DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)* (San Francisco, CA, March 2003), pp. 73–88.

[33] STEENSGAARD, B. Points-to analysis by type inference of programs with structures and union. In *Proceedings of the 6th International Conference on Compiler Construction* (1996), pp. 136–150.

[34] SUTTON, A., AND MALETIC, J. Mappings for accurately reverse engineering uml class models from c++. In *Proceedings of the 12th Working Conference on Reverse Engineering (WCRE)* (2005).

[35] WONG, K., TILLEY, S. R., MÜLLER, H. A., AND STOREY, M.-A. D. Structural redocumentation: A case study. *IEEE Software 12*, 1 (1995), 46–54.