An Experience Report on Extracting and Viewing Memory Events via Wireshark

Sarah Laing
University of Calgary

Michael E. Locasto
University of Calgary

John Aycock
University of Calgary

Abstract

Modern program analysis environments lack a principled method of monitoring low-level memory events. Such monitoring is of great value to activities like debugging, reverse engineering, vulnerability analysis, and security policy enforcement. Although current systems can be coerced to produce streams of memory events, most such techniques are inefficient or overly invasive and offer an unconstrained control over memory, which can subvert the reliability of such memory interposition as part of the attack engineering workflow.

Our system, Cage, is a kernel-level mechanism for monitoring the memory events of a process. Like several existing memory trapping systems, Cage modifies and uses the functionality of the Linux kernel memory page subsystem. Cage translates the memory activity of a process into a packet-like format, and these events are exported over a network device. The memory event packets can be captured and displayed using an existing network packet analyzer (Wireshark). At present, Cage can monitor the memory events for the data, stack, and heap of a process as well as arbitrarily cage any other memory region. We have caged a Gnome login session successfully and noticed no ill effects. We discuss several potential applications that arise from imposing this "network packet" metaphor on memory events.

1 Introduction

This paper is an experience report that discusses the pros and cons of building an in-kernel memory event interception and export mechanism. Intercepting, extracting, and analyzing memory events is a sequence of activity common to both offensive and defensive systems security (e.g., debugging, vulnerability analysis, process profiling, heap analysis and structure setup to prepare for exploit delivery). We built a system called Cage that traps user-selected types of memory events inside the kernel and then exports them to a network interface. We are able to filter these "packets" two ways: early with BPF [17], and later with Wireshark. Besides applications like finding and leaking portions of the address space, extensions to BPF permit us to rewrite the events dynamically. A primary contribution of this work is the definition of the memory event protocol format, which facilitates processing and both types of filtering.

The design of Cage builds on some existing memory interception primitives and techniques, most notably the Memalyze paper by skape, the ELFbac work from Dartmouth, and the Linux kernel's own kmemcheck. Although these techniques have illustrated the basics of this approach to memory interception (i.e., using PTE bits to mark certain memory pages), we highlight the difficulties of composing them into a coherent and general-purpose memory trapping mechanism that exposes a computationally-constrained API (i.e., that of a packet filter). So, while Cage's memory interception mechanism inherits a design from Memalyze, the system as a whole is an argument for a principled engineering approach [6] to modeling extracted data (rather than inserting arbitrary computation into the dynamic memory analysis).

1.1 Background Rationale

The disposition of a target process's address space is often of more interest to various forms of analysis and attack than the process's specific place in "control flow." We agree with the observation that a data-centric view of a target's behavior is often more revealing than a control-centric view. Cage is therefore focused on extracting a stream of memory events from a process's behavior rather than recovering a control flow graph.

Many academic (and some commercial) defensive techniques focus on making it difficult to intercept or modify control flow, but it is no surprise that such instrumentation is often either overly invasive or ultimately ineffective at recognizing arbitrary malicious computation. Instead, the most successful and widely deployed defensive mechanisms (i.e., ASLR, DEP) *cheaply* alter the structure or properties of the address space rather than attempting to compute and supervise the "correct" control flow of an application at any given point in execution. While they are not a panacea, such defensive weird machines are more successful when they focus on modifying *state* rather than controlling computation. Attack techniques can use ROP or various encodings if they really need to defeat protections that focus on control flow, but attacks still need to be aware of the state of the target if the exploit is to unfold successfully.

The poor fit of most control flow integrity solutions teaches us something about the relative merits of studying control flow versus studying memory properties. A data-centric view of a target process is often a more fruitful method of analysis than a control-flow centered view. Although understanding how a process has reached a certain state is not unimportant, it is primarily useful insofar as it indicates the dependencies between data artifacts at a given point in time.

Getting control of EIP/RIP is but one piece of the puzzle; the environment must be suitably friendly to the subsequent computation (whether or not the exploit is ephemeral or persistent). This often requires some amount of control over the layout of memory objects. Even post-exploitation, with an initial and carefully hidden foothold on a host, subsequent exploitation may require an analysis of a second target process or group of threads. We argue that viewing the memory behavior of a process through the restricted lens of a packet stream creates a well-understood API for manipulating the underlying data, and that attack or analysis tools are therefore not in the position of needing to write custom analysis code whose execution is unconstrained.

There are a great many options for intercepting execution and extracting information from various points in the system stack (see Section 2 for a partial list). While extracting from PHY would be great, for now we choose to work within the confines of ring 0. Most of the user-level mechanisms are unsatisfactory from a granularity perspective (e.g., library interception) or a performance and complexity perspective (DBI). The hardware facilities like segments and debug registers are attractive, but have fallen out of use (segments) or are too few in number (debug registers). In some sense, the unused PTE bits represent a sweet spot in the hobbled architectural features available in x86 for introspection and monitoring: a type of "firmware" version of a debug register – not as efficient, but plentiful and somewhat more flexible.

1.2 Weaving Functionality In: AOP vs. ROP

The interception of memory events as driven by a packet-filter like policy opens up the potential for temporal conditions or complex triggering expressions. In some ways, this is an aspect-oriented approach to exploit programming, where exploit functionality is invoked at particular program states; this complements the ROP approach of stitching together exploit functionality from existing program code.

The composition of temporal watchpoints and entropy extraction is an interesting application of Cage (i.e., wait for a non-specified period of time under a complex expression to witness the desired data, then extract it). It would also be interesting to answer the question: on average, how long does it take until an application reads or writes an "interesting" data structure? The benefit of Cage's approach is that a combination of pre and post filters (accomplished with BPF and Wireshark, respectively) can support post-hoc analysis of this type. For example, an analyst could examine how frequently sshd touches the memory where server keys are stored (or other important state involved in the key exchange and transport setup).

1.3 Objections and Limitations

While Cage interposes on memory accesses, its current implementation is only a passive observer most of the time – it does not supply the ability to statically read the process address space from byte zero onward. Cage observes only the dynamic memory accesses of a target process. We show later how to modify the dynamic stream of memory events (since we already intercept them).

The mechanism Cage uses is inherently inefficient at scale, and this is unsurprising as noted by previous work proposing this kind of mechanism. Its penalty, however, can be tuned by the user via two main features. First, our chmem(2) interface (and chmem(1) tool) allows the user to select a small number of pages (memory regions) to trap. Second, the analyst/user can write BPF filters to discard events that are not of interest (or simply count them without emitting them).

Of course, the actual slowdown incurred will depend on how often pages of interest are referenced by the target program. As we note above, efficient and full-featured interception and introspection are not well-supported on x86, and so most such efforts have to make due with whatever they manage to cobble together. We view Cage and systems like it mainly as an argument for the creation of a "super-MMU" that efficiently supports such labeling, state extraction, and event aggregation. Such a wish hearkens back to tagged architectures, but instead of dynamic data flow tracking, we see great potential for both offensive and defensive weird machines.

Finally, one might object that in order to make use of Cage, the box must already be exploited. Our implementation of Cage is built as part of the Linux kernel source; we do not demonstrate the insertion of Cage or Cage-like functionality as a rootkit. Cage is not an exploit technique or rootkit, but rather a principled way of intercepting memory events that can serve as part of a toolkit for continued analysis of a machine and its programs.

2 Related Work

Cage's main area of contribution is in practical program analysis. Dynamic memory analysis is of general interest, and can be accomplished in many different ways and at different levels of the system stack. There is a wide menu of ways to try to trap memory events, with varying implications for how much data is captured, the granularity of information, and the cost of interception and extraction. We focus on the x86 platform in our work.

2.1 Memory Trapping Alternatives

There are a number of approaches to trapping memory events; they vary according to their chosen interception and inspection techniques. In general, interception must rely on hardware features like exceptions, segmentation, interrupt instructions, traps arising from hardware watchpoints, or traps arising from page meta-data checks. Some work also relies on PHY-level snooping (e.g., CoPILOT [19], in this case, checking the integrity of kernel instructions) on the memory bus.

The Standard Approaches Most commodity platforms come with tools similar to gdb, strace, and the ability to perform library interception of malloc and cousins. System call tracing via strace(1) and similar tools examines only the system-call level memory events and is a read-only view of this event stream. Library interception offers a bit finer granularity and the possibility of rewriting the events, but still misses instruction-level memory events (this lack of fine-grained instruction–level details is the stated motivation behind many projects, including Fenris[1]). Basic debuggers like gdb offer fine-grained observation and control, but require some scripting (gdb offers Python bindings and its own native command set). Some other debuggers (e.g., IDA) offer facilities more specialized to reverse engineering, such as propagation of labels and tags to group common pieces of functionality. Generally, the ptrace(2) mechanism offers custom debuggers or program supervision, but means that arbitrary userspace C code must be implemented by an analyst.

Creative Reuse of Hardware Facilities Most interception has to rest on some physical reality within the system to guarantee consistency and fidelity. Hardware debug registers offer an efficient means of "watching" a small number of memory locations and can even be used for very stealthy supervision [11]. Overloading the PTE is a common approach for a variety of analysis tasks, both offensive and defensive [20]. Finally, despite its utility, few systems use hardware memory segmentation and DPL bits overloading and then mostly for defensive purposes [20, 5, 10].

Emulation, Supervision Environments, Embedded Debuggers "Layer-below" approaches like emulation and virtualization are popular solutions for providing an environment for data collection and analysis of program or guest behavior. Popular platforms for such work include Bochs [4] and QEMU because they offer a way to easily modify CPU behavior in novel ways. Supervision environments and virtualization also offer the ability to support time-travel debugging [14] because of snapshot facilities included in the platform.

Because emulation and virtualization can introduce significant performance limitations (among other problems [7]), another common approach to instrumenting programs is DBI; popular systems include Pin [16], Valgrind [18], and DynamoRio [9, 8]. A complement to dynamic rewriting or runtime recompilation is static rewriting or instrumentation of program binaries, such as that offered by the ERESI project's elfsh [23]. Custom embedded debuggers and reverse engineering frameworks like RADARE [1] are largely concerned with recovering, cataloging, and connecting high-level meta-data and code properties with low-level program behavior.

[1]`http://lcamtuf.coredump.cx/fenris/README`

2.2 Mechanisms Most Similar to Cage

Encoding interesting conditions in page and TLB entries has been tried before, notably with OllyBone [21], grsecurity's PAGEEXEC [22], and ELFbac [2]. Indeed, the Page Fault Weird Machine [3] demonstrates how much power lurks in the paging circuitry.

ELFbac [2] describes how to combine ELF section names with page table marking tricks to trap certain code-data ownership relationships. ELFbac focuses primarily on the access control and labeling scheme behind marking pages. Cage does not deal with such markings or access control schemes built on page labels.

The Fenris project [2] offers a comprehensive set of program analysis tools; like Fenris, Cage exports low-level events from a single dynamic program path. In particular, the Ragnarok output mode of Fenris offers the ability to organize information about the relationship of functions to buffers. Ragnorok's *buffer view* offers "a history of all modifications and I/O ops applied to a single buffer." Fenris is particularly focused on aiding the understanding of program code / execution; in contrast, Cage places relatively more emphases on memory event behavior. Although the debugger tool part of Fenris provides a facility for reporting on "seen" addresses and setting read and write watchpoints on memory ranges, but this is largely limited to trapping at function invocation (in contrast, Cage monitors and filters all memory behavior of a process kernel-side).

Bochspwn [13] instrumented Bochs in order to spot time-of-check-to-time-of-use (more generally, double-fetch) vulnerabilities in the Windows kernel. Interestingly, they mention the approach Cage uses as a potential design option but dismiss it in favor of using Bochs, the "*simplest to quickly implement*" [13, page 17]. Cage, running on native hardware as well as many VMs, does not suffer from Bochspwn's large performance hit as a result. Bochspwn also looks for memory access patterns[3], although currently seems to look for just the one pattern specific to the type of vulnerabilities they are seeking; looking for other patterns they leave as future work. By contrast, Cage is already a more general mechanism that permits arbitrary pre- and post-filtering of memory accesses made within a user process.

Finally, Cage dynamically traces real program execution. Cage is not intended to be a symbolic execution framework, although it is possible that traces from Cage could help drive or test other symbolic execution engines.

[2]`http://lcamtuf.coredump.cx/fenris/devel.shtml`

[3]`https://github.com/j00ru/kfetch-toolkit`

3 Motivation and Design Fundamentals

Cage's main objective is to trap memory events and expose them as a stream of "network packets." We do not attempt to shoehorn them into a pre-existing network protocol format, but rather expose them in such a way so that existing tools (like Wireshark or BPF or other domain-specific packet filtering languages) can offer the ability to manipulate this stream in a well-understood and constrained way.

It is important to note that although our description of Cage and its implementation focuses on its real-time/in-situ operation for trapping memory events, the data it captures and produces can support post hoc and offline analysis (indeed, we expect most users might adopt this scenario).

The Cage system contains two primary components: (1) the memory event interception mechanism and the (2) packet generation code. The memory interception mechanism is based on a hybrid of PAGEEXEC [22] and the design articulated by skape [20] without the use of mirrored page tables. Cage is available on Github.[4] Cage is implemented as a set of modifications to the Linux kernel 3.9.4 and is implemented to work on `x86_64`. Like previous mechanisms, Cage takes advantage of unused bits in the page table entries (see Figure 1).

3.1 Cage Workflow

Pages in the process address space can be marked as caged by a combination of several pieces of state. First, a process is marked as caged via a flag communicated in the clone(2) call or when our new system call chmem(2) is invoked. This state is stored in the task struct. Second, pages inherit their PTE from the memory region metadata structure, so we mark memory regions as caged or not. This marking includes an arbitrary 10-bit label which is exposed to our packet format and can help filter sets of events.

An access to a caged page triggers a page fault because the caged page is being accessed from user space, but is marked as a supervisor page. Normally, page faults are handled by the page fault logic and might (for example) swap in a page from disk, deal with copy-on-write, or produce a segmentation violation. In this case, we insert logic to check that this is indeed a caged page and then uncage the page to allow the requested access for the restarted instruction. However, leaving the page uncaged would eliminate future traps on access. In ad-

[4]Code at: `https://github.com/selaing/Cage.git`. We welcome technical feedback on this work-in-progress.

63	62...52	51...12	10...9	8...3	2	1	0
XD	ignored	physical frame address (high bits reserved)	ignored	status bits	U/S	R/W	1

Figure 1: PTE diagram, based on [12]. Cage reserves bits 52–62 as a "label". Like previous approaches to such trapping, it overloads the User/Supervisor bit (2). We also overload bit 9 (to help distinguish a "Cage" page from other abuses of bit 2), which is supposedly unused (but is actually used by the kernel).

dition, letting the instruction restart and execute would allow the virtual-to-physical mapping to be stored in the TLB. We need to allow the instruction to complete, but then mark/re-cage the page and eliminate its entry from the TLB so that future references to the page do not skip the trap. Thus, we have to flush an entry from the TLB each time we re-cage a page to remove the mapping for the uncaged page from the TLB. Before letting the uncaged instruction proceed, we enter single-step mode, which allows us to re-cage after the instruction executes once.

Before the instruction is restarted, we capture a variety of information associated with the event, including:

1. the instruction pointer ... *bytes 0–7*
2. the effective address (faulting address) ... *bytes 8–15*
3. the instruction label (Cage "label", if any, of the page containing the faulting instruction) ... *bytes 16–17, 11 bits*
4. the effective address label (Cage "label") ... *bytes 17–19, 11 bits*
5. the PTE meta-data for the effective address (first 10 bits of the PTE) ... *bytes 19–20, 10 bits*
6. type of access: r/w/x ... *byte 21*
7. the instruction at the instruction pointer ... *bytes 22–36*
8. the PID and UID ... *bytes 37–40 and 41–44*
9. the data at the effective address ... *bytes 45–52*

The italicized text above indicates where the information is placed in the memory event packet that is sent through to Wireshark; the structure is shown in Figure 2. In the next section, we discuss the coding details of Cage and how it operates on a running example instruction and memory access.

4 Implementation and Running Example

The information we extract during trapping the event is placed into an instance of our data type `struct memevent_packet` and then exported on a virtual network interface (see Figure 2). We wrote an LKM to create a network device. When the device is set to "up", it allocates memory for a ring buffer (10 packets long) and sets a pointer to that memory in the Cage code so that Cage functions can access and maintain this buffer.

If this pointer is set (not NULL), then packet creation will occur. If it is NULL the packet creation does not happen. Thus, unless the LKM is loaded and the device is set to "up" no packets are actually created. The Cage code creates a packet using the metadata (listed in Section 3) it has concerning the memory access. It then looks up the `net_device` struct associated with the network device created by the LKM and calls the transmit function for that network device. The transmit function dequeues the packet off of the buffer in Cage.

It then encapsulates this information in an Ethernet header (using an ether type that is listed as being unused and available for personal use). A struct `sk_buff` is created and the packet (with its Ethernet header) is placed in the `sk_buff`. The function `netif_rx` is then called to pass the `sk_buff` to the receive network path of the kernel.[5] This arrangement effectively turns this memory network device into a loopback device. Wireshark can then capture off of this device and (coupled with the dissector specification we wrote) can display the information in the memory event packets. The flexibility of this approach was driven home when, in order to create an event noting the creation or modification of a memory region (Linux `vm_area_struct`), we simply had to introduce a new packet subtype by reusing a few unused bits in our packet meta-data.

4.1 Workflow Example

To illustrate Cage's workflow, we use the instruction

```
mov rax, [rbx+0x60014c]
```

as an example. The memory location 0x60014c, a constant in the .data section, has the page table entry shown in Figure 3. Bit 9 in the page table entry is set indicating that this is a Cage page. The user/supervisor bit

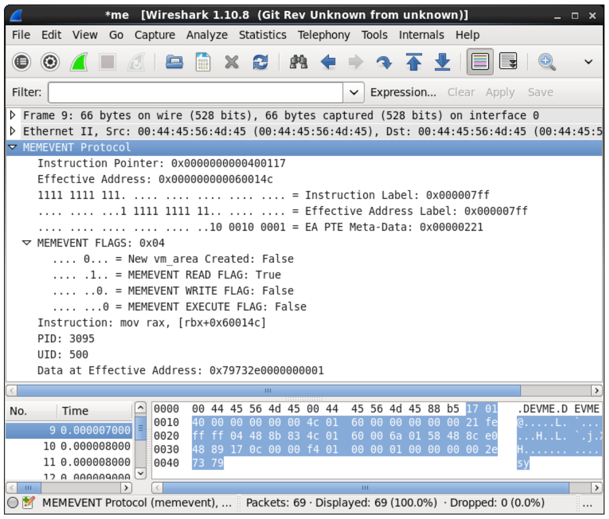
[5]It could easily be transmitted out to the network at this point.

```
struct memevent_packet {
        unsigned long src;
        unsigned long dest;
        unsigned int src_dest_pte;
        unsigned char rwx;
        unsigned char instruction[15];
        unsigned int pid;
        unsigned int uid;
        unsigned long data;
}
```

Figure 2: Structure of Memory Event Packet.

63	62…52	51…12	10…9	8…3	2	1	0
XD	label 11111111111	physical frame address	xx 1	status bits	0	0	1

Figure 3: PTE Contents for the Running Example.


*me [Wireshark 1.10.8 (Git Rev Unknown from unknown)]
File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help
Filter:
Expression... Clear Apply Save

```
Frame 9: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
Ethernet II, Src: 00:44:45:56:4d:45 (00:44:45:56:4d:45), Dst: 00:44:45:56:4d:45 (00:44:45:5
MEMEVENT Protocol
    Instruction Pointer: 0x0000000000400117
    Effective Address: 0x000000000060014c
    1111 1111 111. .... .... .... .... .... = Instruction Label: 0x000007ff
    .... .... ...1 1111 1111 11.. .... .... = Effective Address Label: 0x000007ff
    .... .... .... .... .... ..10 0010 0001 = EA PTE Meta-Data: 0x00000221
  MEMEVENT FLAGS: 0x04
      .... 0... = New vm_area Created: False
      .... .1.. = MEMEVENT READ FLAG: True
      .... ..0. = MEMEVENT WRITE FLAG: False
      .... ...0 = MEMEVENT EXECUTE FLAG: False
    Instruction: mov rax, [rbx+0x60014c]
    PID: 3095
    UID: 500
    Data at Effective Address: 0x79732e0000000001
```

```
No.  Time
  9 0.000007000
 10 0.000008000
 11 0.000008000
 12 0.000009000
```

```
0000  00 44 45 56 4d 45 00 44  45 56 4d 45 88 b5 17 01   .DEVME.D EVME
0010  40 00 00 00 00 00 4c 01  60 00 00 00 00 00 21 fe   @.....L. `...
0020  ff ff 04 48 8b 83 4c 01  60 00 6a 01 58 48 8c e0   ...H..L. `.j.
0030  48 89 17 0c 00 00 f4 01  00 00 01 00 00 00 00 2e   H....... ....
0040  73 79                                              sy
```

MEMEVENT Protocol (memevent), ... Packets: 69 · Displayed: 69 (100.0%) · Dropped: 0 (0.0%) ...

Figure 4: Running Example Event Viewed in Wireshark.

is cleared. This results in a user access to a supervisor page which generates a page fault. Once we have the page fault we recognize that it is due to a Cage page by the unique combination of PTE bits. Once in our Cage code we first run our BPF filter over this memory event. If there is no filter or if the memory event matches the filter we create a memory event packet. This packet is then sent to the network using the process described above. We then set the user/supervisor bit in the page table entry, place the processor in single step mode and return from the page fault handler. The execution of the instruction `mov rax, [rbx+0x60014c]` completes normally, and the beginning of the *next* instruction results in a debug trap. Here we clear the user/supervisor bit, take the processor out of single step mode and flush the single entry corresponding to the page table entry for the address 0x60014c out of the TLB. The resulting packet generated from this instruction can be seen in Figure 4. Note that if the text segment containing the instruction were also caged, then two events would be seen: one for the instruction's page, then one for the data page.

5 Evaluation

In this section, we discuss some of the testing we have performed with Cage to learn about its behavior. We also discuss some possible applications of this type of mechanism.

5.1 Testing

We ran Cage on a variety of platforms and combinations of VMs on both Intel and AMD. The purpose of this was validation and to see how fragile these kind of kernel modifications might be in the presence of virtualization or emulation.

5.2 Entropy Extraction

Cage produces a lot of data. One straightforward application of this kind of memory interception is leaking data. Even relatively simple applications produce large amounts of memory events: 'whoami' produces 288306 packets in 0.287 seconds; 'true' produces 3807 packets in 0.034 seconds.

We ran both ssh and sshd under Cage's observation (the entire process address space was caged) to see on average how much data would be exported. Figure 7 and Figure 8 represent the average of 10 runs each for 90

Architecture	Virtual Machine	Host System	Guest System	Dropped/Missing Packets	Other Issues
Intel	N/A	Centos 6.5	N/A	No	No
	Virtual Box 4.2.16	Mac OSX	Centos 6.5	Yes	No
		Centos 6.5	Centos 6.5	Yes	No
	Virtual Box 4.3.8	Centos 6.5	Centos 6.5	N/A	Trace/Breakpoint Trap
	VmWare Fusion 5.0.0	Mac OSX	Centos 6.5	No	No
	Parallels	Max OSX	Centos 6.5	No	No
	Xen (full-virt and para-virt)	Centos 6.5	Centos 6.4	N/A	Trace/Breakpoint Trap
	QEMU	Centos 6.5	Centos 6.5	No	No

Figure 5: Cage Running on Intel Hardware.

Architecture	Virtual Machine	Host System	Guest System	Dropped/Missing Packets	Other Issues
AMD	N/A	Centos 6.5	N/A	No	No
	Virtual Box 4.2.16	Kubuntu 12.04	Centos 6.5	No	No
		Centos 6.5	Centos 6.5	No	No
	Virtual Box 4.3.8	Centos 6.5	Centos 6.5	No	No
	VmWare Workstation	Kubuntu 12.04	Centos 6.5	No	No
	Xen (full-virt and para-virt)	Centos 6.5	Centos 6.5	N/A	Trace/Breakpoint Trap
	QEMU	Centos 6.5	Centos 6.5	No	No

Figure 6: Cage Running on AMD Hardware.

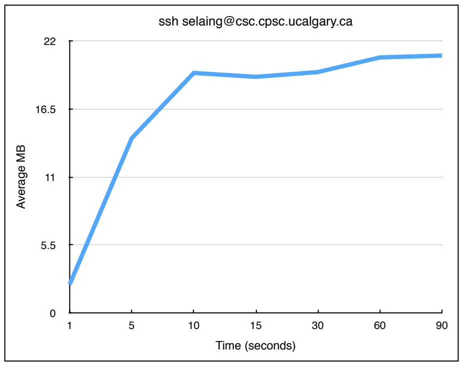


Figure 7: Cage on the ssh client.

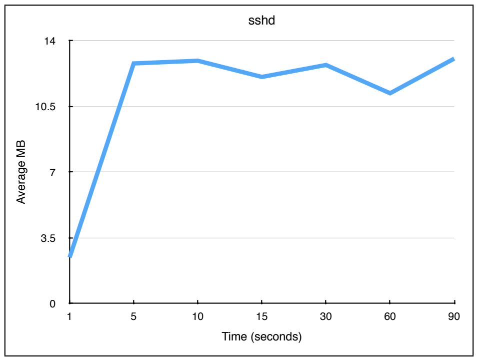


Figure 8: Cage on the sshd server.

seconds. The vertical axis displays the average amount of data (in MB) extracted by Cage.

We sampled the programs ssh and sshd. This was done at 100% caging. The graphs represent the average amount of MB of data exported from the program over time in seconds. Both graphs follow the same kind of curve. The amount of data levels off because there is no input to these applications after the initial startup phase. The startup phase would include checking the keys and passwords and other sensitive data that is probably in the data that is a good target for leaking somewhere. This data is based on the total number of packets that the interface saw, not the number of packets that Wireshark captured.

These kind of graphs show that a non-trivial portion of data is touched in the startup phase of an application. Relatively small amounts of new data are touched after the program has reached a certain point in execution (for sshd, likely the point where it sits in the accept() loop, and for ssh, the point where it has finished establishing the connection and remote shell and is waiting for user input). This plateau indicates that the programs have reached a particular stage in execution, and subsequent analysis can take it as a signal to begin analyzing the follow on execution or to start capturing in earnest (this bursty behavior can threaten to overwhelm buffers or initial data collection).

6 BPF Filtering

To explore some of our ideas related to filtering (both as a solution to performance (i.e., dealing with large bursts of events) and as a means of demonstrating the utility of analysis using a packet filtering language), we have implemented BPF-based filtering early on in Cage's workflow in the page fault handler. As the actual BPF programs themselves can appear inscrutable, we defer those to Appendix A and instead use BPF pseudocode in this section (the appendix maps the syntax back to each figure below).

It is important to point out that our work in this section is in support of the philosophical point from the introduction that "...the system [Cage] as a whole is an argument for a principled engineering approach to modeling extracted data"; the use of BPF for processing (and modifying) streams of memory events is a design alternative to the practice of inserting arbitrary C or x86 code into dynamic memory analysis. This kind of pattern has many precedents: consider the use of SystemTap built on top of kprobes. While inserting raw C or x86 provides a great deal of control, it also entails some risk (the inserted code may be buggy or hard to maintain). In contrast, BPF offers a still–powerful interface for analysis, but avoids some of the risks of inserting arbitrary computation into the instrumented memory event sequence. The gains in stability and reliability offer some justification for the utility of this approach in offensive operations.

6.1 Extensions and Implementation of BPF

BPF is used to filter memory events before packet creation occurs. It operates on each memory event and controls the creation of the memory event packets. This allows the ability to control the packets that are produced so that only packets for the relevant memory events, as decided by the BPF filter, are generated. In implementing this functionality we have had to extend BPF in two ways. First, we have added in a new BPF instruction. This instruction allows the ability to load information relating to a memory event into a BPF filter. In particular this gives us the ability to load the current effective address, the current instruction pointer, and the current value in rax. This was done so that BPF filters can be created which compare these values to a given constant or range of values, either effective addresses or instruction pointers, and emit packets whose memory events match. Adding in the ability to load the current value in rax allowed us to capture the return value of a function call. The usefulness of this became apparent when we wanted to capture the address returned by a call to malloc() and recover the packets that involved this buffer – the compiler could optimize the code such that the buffer address was never written to memory and was thus not viewable as a memory event.

Our new BPF instruction also gives us the ability to load in the values of pieces of state that we have stored such as the current number of events matching the BPF filter expression, and the stored effective address. This allows us the limited ability to store state across runs of the BPF filter, which are different memory events. This allows us greater flexibility in the types of BPF filters we can write. The second way in which we have extended BPF is to allow it to handle 64-bit values. The implementation of BPF in the Linux kernel is only able to 32-bit values. We modified this to allow the filtering function in the Linux kernel to handle the 64-bit values of the effective address and instruction pointer.

6.2 Filter Types

Currently we have implemented five different types of BPF filters; at present they are contained in a kernel module. Each of these filters is supported by code within Cage that allow it to either store state under certain conditions or retrieve state through use of our specialized load function.

6.2.1 Temporal Filtering

The purpose of this filter is to induce a type of rate limiting on the number of memory event packets that are generated. There are three different types of this filter. One that watches for only read accesses, one that watches for only write accesses and one that watches for either read or write accesses. This can be used to only produce a packet every n^{th} event. For example, if we know that the n^{th} write to an array is important we can ignore the first n-1 events. This filter watches for memory events that fall into a specified range of addresses. If the memory event falls within this range, the filter loads the current number of memory events that have been seen that fall within that range. If this stored value is equal to the number of events we are waiting for then a memory event packet is created. This filter is supported by code in Cage that maintains a piece of state storing how many events of this type have already occurred. Each time the specified number of events is reached, the stored number of events is set to zero. Figure 9 shows the logic of this BPF filter.

6.2.2 Overwriting Data at an Effective Address

The purpose of this filter is to overwrite the data contained at a specific effective address with a different spec-

```
L1:  A = current effective address
L2: (A == begin_address) ? goto L3 : goto L10
L3: (A > end_address) ? goto L10 : goto L4
L4:  A = A - end_address
L5: (A > end_address) ? goto L7: goto L6
L6: (A == end_address) ? goto L7: goto L10
L7:  A = current number of events
L8: (A == num) ? goto L9 : goto L11
L9:  return 1
L10: return 0
L11: return -1
```

Figure 9: Temporal filter. This filter will emit a memory event packet for every n^{th} packet as specified by num provided that memory event falls into a specific range of addresses specified by begin_address and end_address. Note: There is no functionality within BPF for jumping if A is less than some value.

```
L1:  A = current effective address
L2: (A == address) ? goto L3 : goto L4
L3:  return num
L4:  return 0
```

Figure 10: Data-overwriting filter. This filter will overwrite the data stored at the effective address specified by address with the value num.

ified value. This can be used to change the contents of the memory of a program at runtime. This filter watches for a memory event corresponding to a specific effective address. When this effective address is accessed during program execution the filter returns the value we wish to overwrite the data with. This is supported by code in Cage that watches for a non-zero return from the filter. Cage then takes the value returned by the filter and overwrites the data contained at the current effective address with this value. Figure 10 shows an example of the logic of this BPF filter.

6.2.3 Overwriting an Instruction

The purpose of this filter is to overwrite an instruction at a specific instruction pointer with a different specified instruction. This can be used to modify the executing instruction of a program at runtime. The new instruction must be the same size as the old instruction to prevent overwriting the next instruction. Overwriting an instruction in this way has the side effect of modifying the binary so that each subsequent run of the binary executes the new instruction. This filter watches for a memory event corresponding to a fetch of a particular instruction specified by the given instruction pointer. When this memory event occurs the filter returns the number of bytes contained in the new instruction. This is the number of bytes that will be written to the location of the current instruction pointer. The new instruction is passed to Cage through a filter structure that contains the BPF instructions along with the type of BPF filter we are executing. This is supported with code in Cage that recognizes the non-zero return value from the filter and overwrites the instruction contained at the current instruction pointer with the new instruction. This overwriting occurs during the memory event corresponding to the fetch of the instruction. Figure 10 shows an example of the logic of this BPF filter with the exception that we load the instruction pointer in the first step instead of the effective address.

6.2.4 Viewing a Buffer Allocated at Runtime

The purpose of this filter is to view all the memory events corresponding to a specific buffer that has been allocated at runtime. In this instance we cannot know the effective address to search for ahead of time. This filter gives us the ability to locate a dynamically allocated data structure of interest and recover all the memory events from the point of its creation on. This filter relies on the input of a specific instruction pointer. The value contained in rax at this instruction must be the address of a buffer. This filter watches for the memory event corresponding to the execution of a specific instruction pointer. When this memory event occurs, the value contained in rax at this time is loaded into the BPF filter and this value is returned. Supporting code in Cage then recognizes this return value and stores the returned value in a piece of state. The rest of the filter is executed any time the current instruction pointer does not match the specified instruction pointer. This part of the filter loads the stored rax value and checks to see if the current effective address is within the range of the stored rax value plus a specified value which is the length of the buffer to search for. If it is within this range then the memory event corresponds to the buffer we are looking at and a packet is emitted. Figure 11 shows an example of the logic of this BPF filter.

6.2.5 Capturing the Buffers In A Program

The purpose of this filter is to produce memory event packets corresponding to every buffer in a program that gets accessed sequentially. There are two types of this filter, one that looks for successive reads from a buffer and one that looks for successive writes to a buffer. This

```
L1:   A = current instruction pointer
L2:  (A == address) ? goto L3 : goto L5
L3:   A = current value of rax
L4:   return A
L5:   A = stored value of rax
L6:   X = A
L7:   A = current effective address
L8:  (A >= X) ? goto L9 : goto L15
L9:   A = X
L10:  A = A + num
L11:  X = A
L12:  A = current effective address
L13: (A > X) ? goto L15 : goto L14
L14:  return 1
L15:  return 0
```

Figure 11: Viewing memory events for a buffer allocated at runtime. This filter will produce memory event packets for every memory event corresponding to a dynamically allocated buffer.

```
L1: A = stored value of effective address
L2: X = A
L3: A = current value of effective address
L4: A = A - X
L5: (A == 0x8) ? goto L6 : goto L7
L6: return 1
L7: return 0
```

Figure 12: Filter to find buffers in a program. This filter will search for sequential read or write accesses to buffers within a program and emit memory event packets for these buffers.

filter compares the stored effective address with the current effective address and produces a packet if the two addresses are within eight bytes of each other. Eight bytes was chosen under the assumption that most sequential reads and writes to a buffer are optimized to read or write eight bytes at a time on a 64-bit system. This filter is supported by code in Cage that updates the value of the stored effective address each time the filter completes. Figure 12 shows an example of the logic of this BPF filter.

7 Future Work

This section discusses some areas for improvement that could benefit from the feedback of the workshop attendees. Even despite existing guidance and prior work, implementing this kind of mechanism is not straightforward. In some sense, this difficulty reinforces our belief in the need for a super-MMU to support this type of operation. We cover these issues from the mundane and solved to the weird and unsolved.

7.1 Implementation Issues (Addressed)

We had to add code into our fault handler mechanism to detect different types of page faults such as a page fault occurring because of a COW page. We had to detect these different types of page faults so that we can allow the normal page fault handler to handle these events before our code gets to them.

As mentioned, bit 9 in the PTE is the bit we chose to use to represent a caged PTE because it was "unused." However, it is actually used by the kernel to indicate a "special" PTE. In the function `vm_normal_page`, which returns the `struct page` given the `vm_area_struct`, an address, and a PTE, the special mapping is used to indicate that no `struct page` should be associated with that PTE. We had to recode the function to also check for our flag in the `task struct` that indicates whether or not the current process is caged. The function `vm_normal_page` is called in many different places (e.g., when a process is terminated and its memory is unmapped).

Related work to this approach (namely: kmemcheck and kmmiotrace) use per-cpu variables to store data across the single step and into the debug fault handler (such as the address to be flushed out of the TLB). When we tried to use per-cpu variables we found that we were occasionally missing re-caging a page. The hypothesis was that because the cpu variable is local to one cpu and we are using multiple cpu's, the cpu that handled the debug fault was different than the one that handled the page fault and so we did not have the correct data we needed to successfully re-cage the page after the page fault. We stopped using cpu variables and instead embedded the information in the task struct.

7.2 Implementation Issues (Sidestepped)

Different virtual machines have different reactions to the mechanism either with missing page faults or a trace/breakpoint trap on every instruction (other than the one we are expecting because of single step mode). This is likely due to the composition of certain VM implementation tricks with our kernel modifications. The tables in Figure 5 and Figure 6 list the combination of platforms, architectures, operating systems, and virtual machines we tested Cage on. The lesson here is that nothing works perfectly in practice, but targeting this kind of instrumentation into the kernel seems like a fairly portable means of deploying it (compared to emulation–based approaches).

7.3 Implementation Issues (Looming)

Debugging any caged process with gdb does not work. The mechanism will not work if the process is already being debugged.

During our attempts to build some automatic validation (to confirm that the mechanism would work on certain platform combinations), we encountered issues "diffing" two packet traces due to unexpected changes in the data that is at the effective address, even between subsequent runs of unmodified code on the same machine. These differences have been narrowed down to changes in the location of the environment variables. The differences occur when the data at the effective address contains the address of an environment variable. This raises the larger issue of building a reliable recognizer to model this kind of data stream language.

An additional problem that exacerbates diffing is the variability in the number of packets generated between machines. The number of packets generated on one machine will change any time the program is prelinked and across reboots. We found that turning prelinking off as well as turning ASLR off prevents the number of packets being generated from changing. The number of packets is also different between different machines even while running the same program. This is because there is code that checks information about the processor in the startup code of a program, therefore running the same program on different processors will result in a different number of packets.

7.4 BPF Improvements

The kernel networking infrastructure can handle the creation of many more packets than Wireshark can receive. Programs that are caged at 100% are likely to rapidly produce large amounts of data, and such data rates quickly overwhelm dumpcap's abilities (note that dumpcap is somewhat of an optional bottleneck at the very end of a memory event's journey). This slowdown can be mitigated by using BPF to limit the number of packets that Wireshark sees (for example, by narrowing in on a specific range of heap addresses rather than events for the entire PAS). The intended use is to create memory transaction dumps – for now, we view these with Wireshark, but other GUIs or analysis tools can be developed later. Our focus is on manipulating and analyzing a memory event stream via the metaphor of network packet traces.

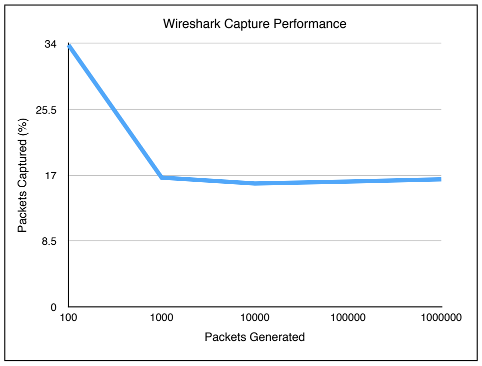


Figure 13: Wireshark's ability to handle high data rates quickly degrades.

The eventual goal of this type of monitoring is to scan for and categorize or model memory event patterns related to certain types of bugs (for example, correlated data structure updates [15], integer overflows, use-after-free, etc.) and then find other candidate instances of those bugs or flaws. This approach would complement code-pattern approaches like that of Yamaguchi et al. [24].

8 Conclusion

While some might contend that monitoring memory events is not inherently offensive, we suggest that program behavior analysis is the essential root of many offensive and defensive activities. This paper is part of a broader argument [6] that "offensive" does not mean *unprincipled* or *ad hoc*. The Cage system is aimed at demonstrating how a consistent model of memory interception can impose some structure on the task of analyzing a target address space.

Unfortunately, the recognition and extraction of streams of memory events is quite useful, but often poorly supported in the OS and ISA. We see most existing work that leverages page tables and the TLB as an argument for better hardware support for memory and data-centric processing. Being able to efficiently aggregate events and extract state would be useful to both offense and defense. In keeping with the theme of WOOT, it is precisely the variety of creative abuses of existing memory management circuitry that argue for a more sane and powerful hardware support for memory introspection on commodity architectures.

Acknowledgments

This work was funded in part by grants from the Natural Sciences and Engineering Research Council of Canada. We appreciate the guidance and advice of our paper shepherd, Julien Vanegue — the paper has undergone significant improvement due to his help. We owe thanks to Darcy Grant for some technical advice and support.

References

[1] Radare: The Reverse Engineering Framework. Project website. http://radare.org/.

[2] J. Bangert, S. Bratus, R. Shapiro, M. Locasto, J. Reeves, S. W. Smith, and A. Shubina. ELFbac: Using the Loader Format for Intent-Level Semantics and Fine-Grained Protection. In *Dartmouth College Computer Science Technical Report TR2013-727*, July 2013.

[3] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. The page-fault weird machine: Lessons in instruction-less computation. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, Berkeley, CA, 2013. USENIX.

[4] Bochs. `http://bochs.sourceforge.net`, 2004.

[5] S. Bratus, M. Locasto, and B. Schulte. SegSlice: Towards a New Class of Secure Programming Primitives for Trustworthy Platforms. In *International Conference on Trust and Trustworthy Computing (TRUST 10)*, pages 228–245, 2010.

[6] Sergey Bratus, Julian Bangert, Alexandar Gabrovsky, Anna Shubina, Michael E. Locasto, and Daniel Bilar. “Weird Machine” Patterns. In Clive Blackwell and Hong Zhu, editors, *Cyberpatterns*, pages 157–171. Springer International Publishing, 2014.

[7] Sergey Bratus, Michael E. Locasto, Ashwin Ramaswamy, and Sean W. Smith. VM-based Security Overkill: A Lament for Applied Systems Security Research. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW ’10, pages 51–60, New York, NY, USA, 2010. ACM.

[8] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 265–275, 2003.

[9] E. Duesterwald and S. P. Amarsinghe. On the Run – Building Dynamic Program Modifiers for Optimization, Introspection, and Security. In *Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[10] Bryan Ford and Russ Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC’08, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

[11] halfdead. Mistifying the debugger, ultimate stealthiness. In *Phrack 65:8*, 2008.

[12] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Number 325462-046US. March 2013.

[13] M. Jurczyk and G. Coldwind. Identifying and exploiting Windows kernel race conditions via memory access patterns. White paper, presentation at SyScan 2013, 2013.

[14] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-traveling Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC ’05, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[15] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP ’07, pages 103–116, New York, NY, USA, 2007. ACM.

[16] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of Programming Language Design and Implementation (PLDI)*, June 2005.

[17] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on*

USENIX Winter 1993 Conference Proceedings, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[18] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, 2003.

[19] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In 13^{th} *USENIX Security Symposium*, pages 179–194, 2004.

[20] Skape. Memalyze: Dynamic Analysis of Memory Access Behavior in Software. In `http://uninformed.org/?v=7&a=1&t=sumry`, 2007.

[21] Joe Stewart. OllyBone: Semi-Automatic Unpacking on IA-32. In *DEFCON 14*, 2006.

[22] PaX Team. PAGEEXEC. In `http://pax.grsecurity.net/docs/pageexec.old.txt`, 2003.

[23] The ERESI team. The ERESI Reverse Engineering Software Interface. Project website. http://www.eresi-project.org/wiki/.

[24] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 359–368, New York, NY, USA, 2012. ACM.

A Raw BPF Filter Code

Temporal Filter (cf. Figure 9)

```
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 1)
BPF_JUMP(BPF_JMP+BPF_JGE+BPF_K, begin_address,0,7)
BPF_JUMP(BPF_JMP+BPF_JGT+BPF_K, end_address,6,0)
BPF_STMT(BPF_ALU+BPF_SUB+BPF_K, end_address)
BPF_JUMP(BPF_JMP+BPF_JGT+BPF_K, end_address, 1, 0)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0, 0, 3)
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 2)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, num, 0, 2)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, 0)
BPF_STMT(BPF_RET+BPF_K, -1)
```

Data-Overwriting Filter (cf. Figure 10)

```
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 1)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, address, 1, 0)
BPF_STMT(BPF_RET+BPF_K, 0)
BPF_STMT(BPF_RET+BPF_K, num)
```

Instruction-Overwriting Filter (cf. Section 6.2.3)

```
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 3)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, address, 1, 0)
BPF_STMT(BPF_RET+BPF_K, 0)
BPF_STMT(BPF_RET+BPF_K, num)
```

Buffer-Viewing Filter (cf. Figure 11)

```
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 3)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, address, 0, 2)
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 6)
BPF_STMT(BPF_RET+BPF_A, 0)
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 5)
BPF_STMT(BPF_MISC+BPF_TAX, 0)
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 1)
BPF_JUMP(BPF_JMP+BPF_JGE+BPF_X, 0, 0, 6)
BPF_STMT(BPF_MISC+BPF_TXA, 0)
BPF_STMT(BPF_ALU+BPF_ADD+BPF_K, num)
BPF_STMT(BPF_MISC+BPF_TAX, 0)
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 1)
BPF_JUMP(BPF_JMP+BPF_JGT+BPF_X, 0, 1, 0)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, 0)
```

Buffer-Finding Filter (cf. Figure 12)

```
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 5)
BPF_STMT(BPF_MISC+BPF_TAX, 0)
BPF_STMT(BPF_LD+BPF_W+BPF_ABS, 1)
BPF_STMT(BPF_ALU+BPF_SUB+BPF_X, 0)
BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, 0x8, 0, 1)
BPF_STMT(BPF_RET+BPF_K, 1)
BPF_STMT(BPF_RET+BPF_K, 0)
```