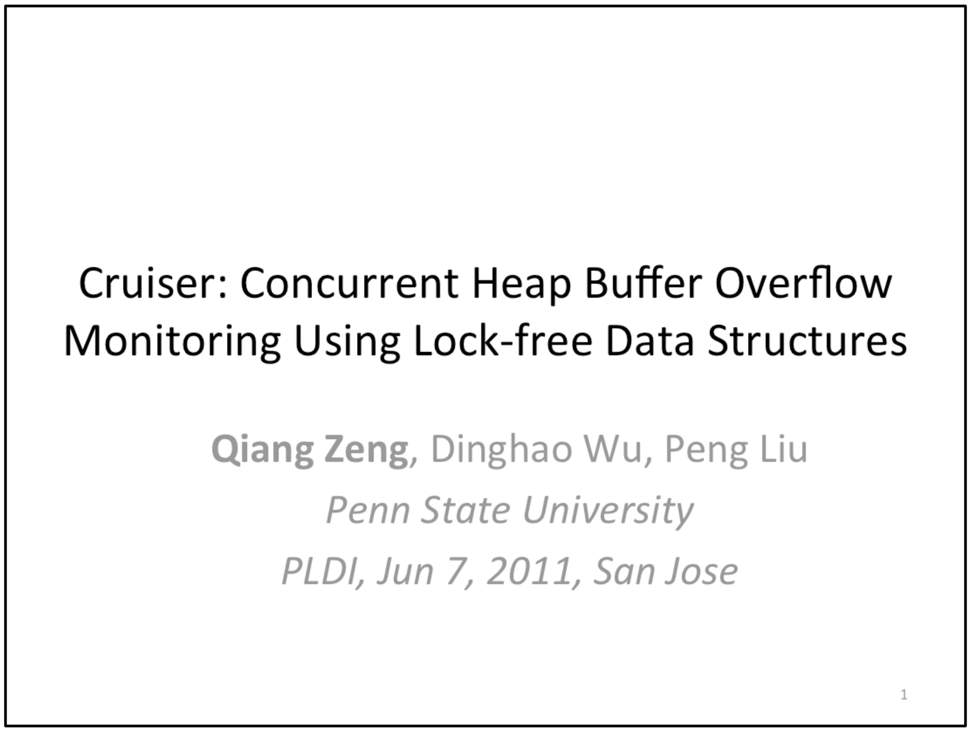
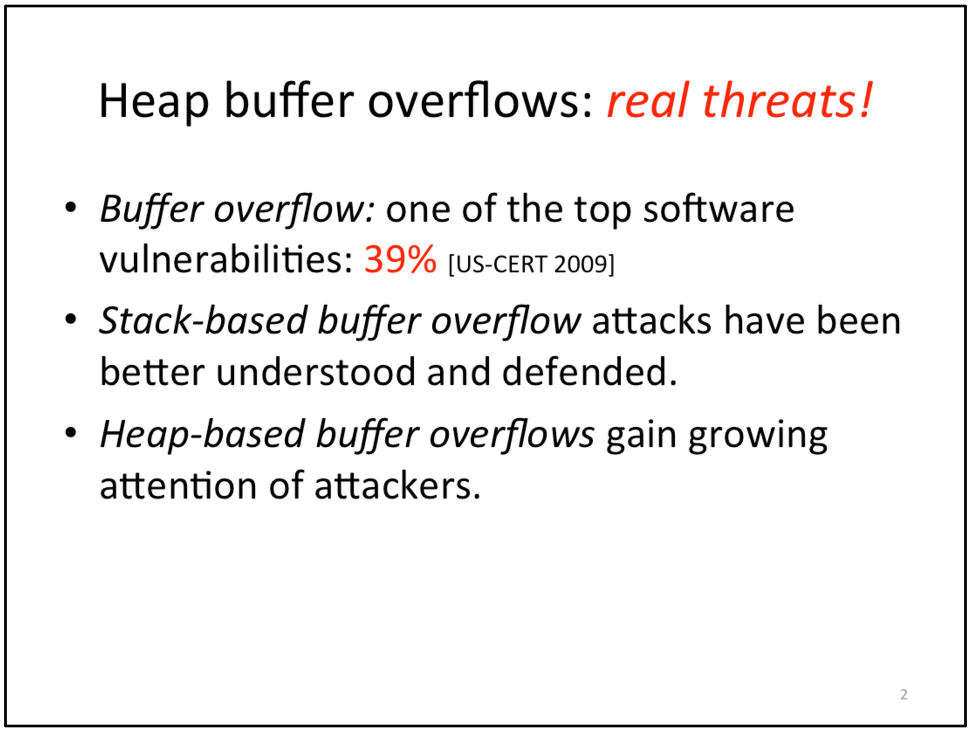


Cruiser: Concurrent Heap Buffer Overflow Monitoring Using Lock-free Data Structures

Qiang Zeng, Dinghao Wu, Peng Liu

Penn State University

PLDI, Jun 7, 2011, San Jose

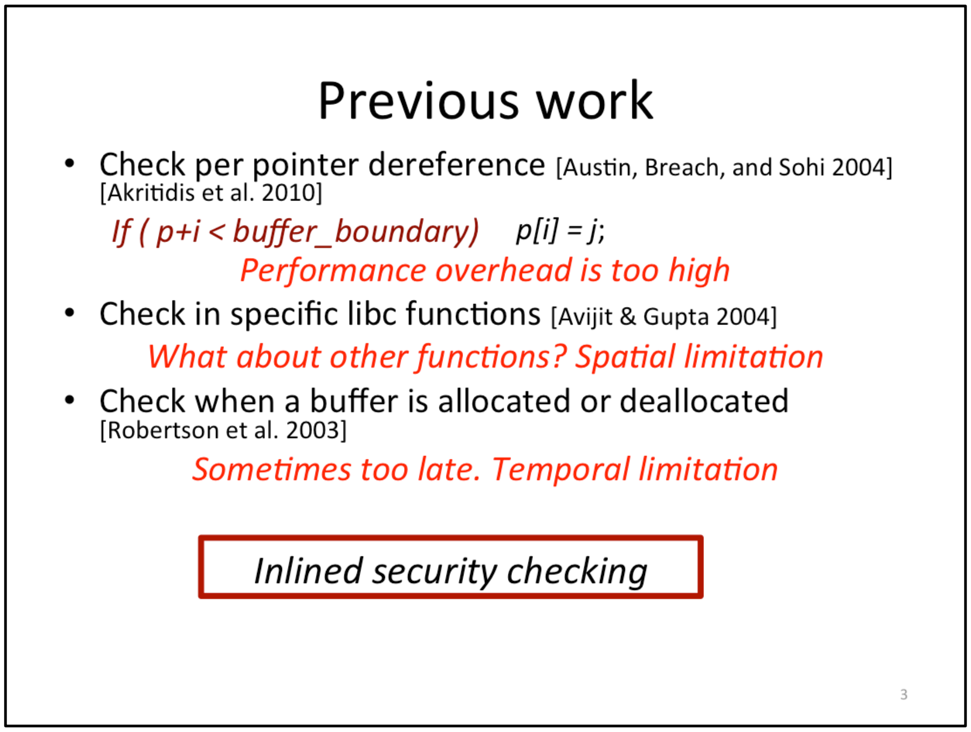


Heap buffer overflows: *real threats!*

- *Buffer overflow:* one of the top software vulnerabilities: 39% [US-CERT 2009]
- *Stack-based buffer overflow* attacks have been better understood and defended.
- *Heap-based buffer overflows* gain growing attention of attackers.

2

Despite extensive research over the past few decades, buffer overflows remain as one of the top software vulnerabilities. In 2009, 39% of the vulnerabilities published by US-CERT were related to buffer overflows. There are mainly two types of buffer overflows: stack-based and heap-based. Many effective countermeasures against stack-based buffer overflows have been devised; some of them have been widely deployed, such as StackGuard. As stack-based buffer overflows have been better understood and defended, heap-based buffer overflows gain growing attention of attackers. Related exploits have affected some well-known programs, such as Microsoft Data Access Component and the SQL Server.



Previous work

- Check per pointer dereference [Austin, Breach, and Sohi 2004] [Akritidis et al. 2010]

  If (p+i < buffer_boundary)   p[i] = j;

  Performance overhead is too high

- Check in specific libc functions [Avijit & Gupta 2004]

  What about other functions? Spatial limitation

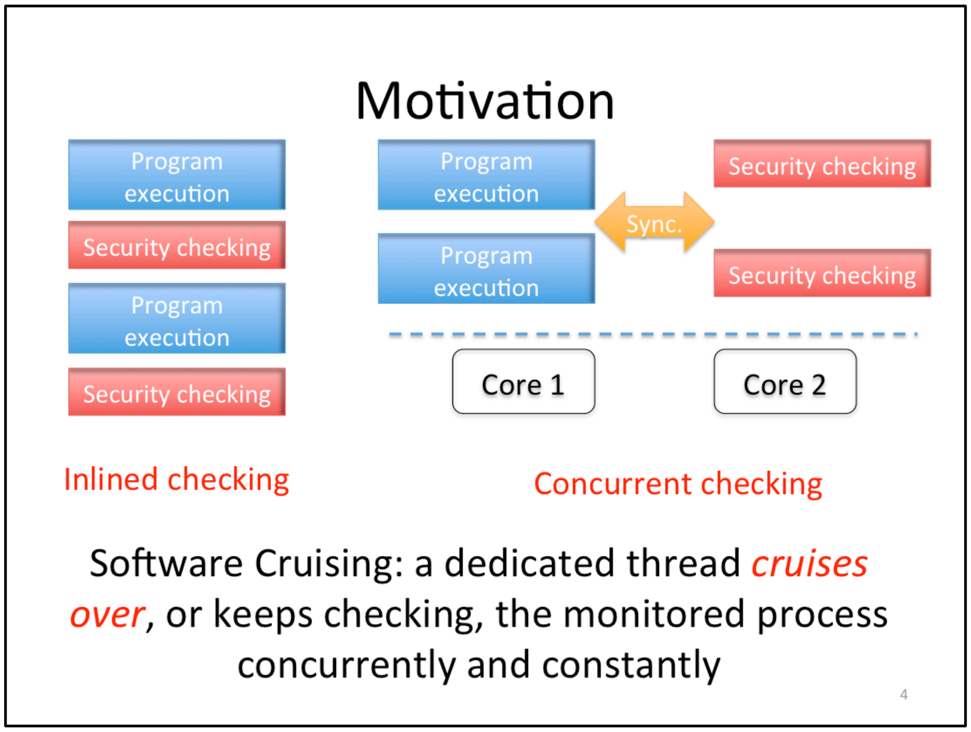
- Check when a buffer is allocated or deallocated [Robertson et al. 2003]

  Sometimes too late. Temporal limitation

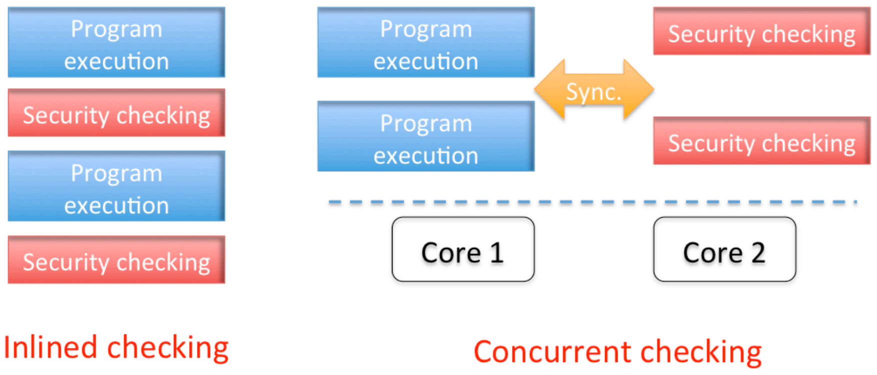
Inlined security checking

3

Existing dynamic heap buffer overflow detectors can be roughly divided into three categories according to the overflow checking occasions. The first category associates a buffer boundary lookup with each pointer dereference, which typically imposes times of performance overhead. The second category checks buffer overflows in specific libc functions, such as strcpy and gets. Buffer overflows existing in other functions are ignored, thus they have evident spatial limitations.
The third category checks buffer overflows when a buffer is allocated, deallocated or combined. However, the vulnerability may have been exploited much earlier than the checking. The common characteristic of these detectors is inlined security checking, which either delays program execution too much or has spatial or temporal limitations.



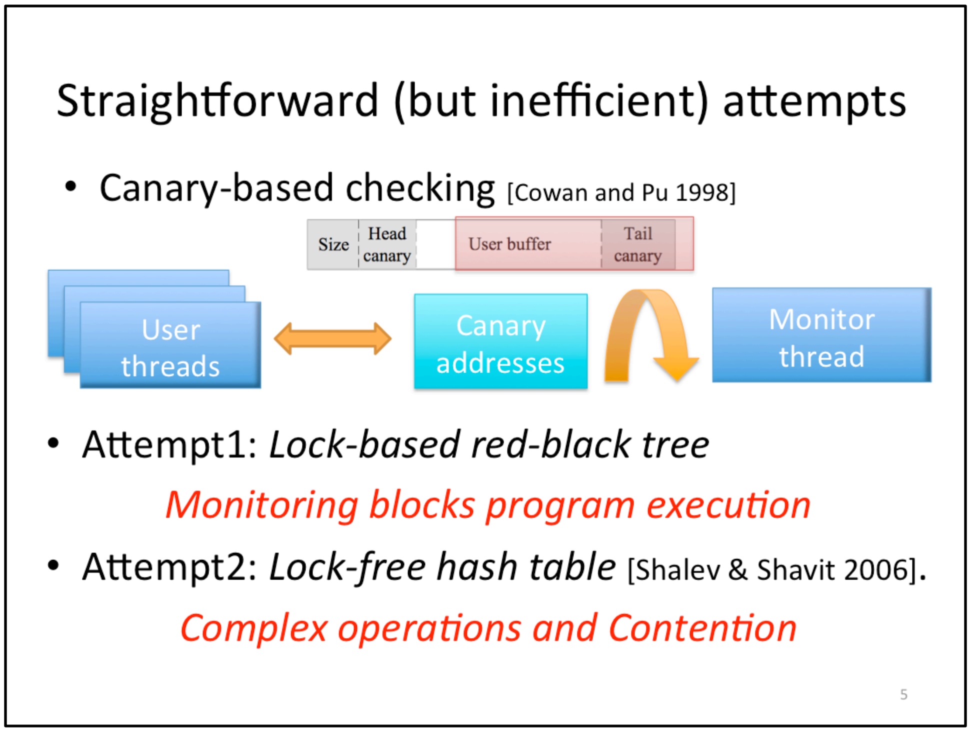
Motivation



Software Cruising: a dedicated thread *cruises over*, or keeps checking, the monitored process concurrently and constantly

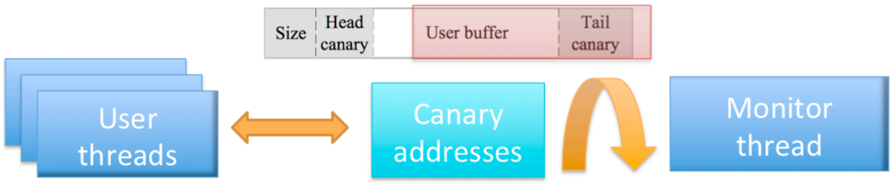
4

As we can see in this graph, the inlined security checking and the normal program execution are in a sequential order. Nowadays multicore architectures are more and more popular. We intended to move the inlined security checking from the program to a separate monitor thread, so that the delay due to security checking can be largely eliminated. In addition, the monitor thread can cruise over the process concurrently over and again at a high frequency throughout the process lifecycle. With multicore architectures, the additional cost for running a separate monitor thread is little. In addition, because the major operation of monitoring is read, the performance overhead can be very low, as long as the synchronization between user threads and the monitor thread is light-weight. On the other hand, if the synchronization overhead is high, the benefits gained from concurrent monitoring may be offset by the synchronization overhead, so it is critical also challenging to keep the synchronization overhead low. We call this concurrent monitoring technique with light-weight synchronization as Software Cruising.



Straightforward (but inefficient) attempts

- Canary-based checking [Cowan and Pu 1998]



- Attempt1: *Lock-based red-black tree*

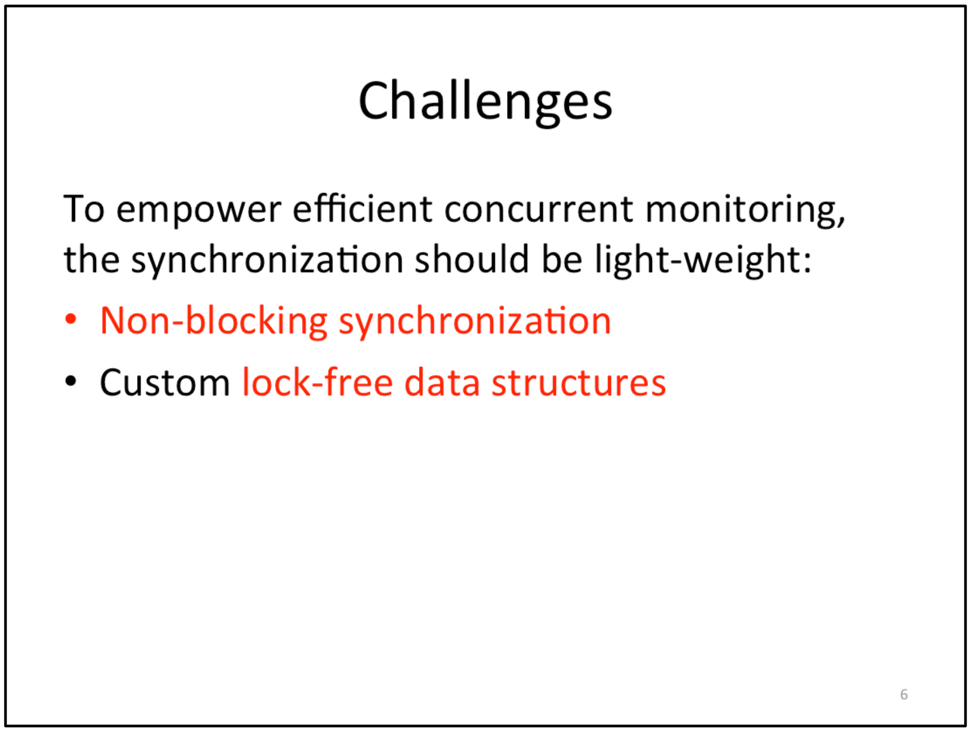
  Monitoring blocks program execution

- Attempt2: *Lock-free hash table* [Shalev & Shavit 2006].

  Complex operations and Contention

5

In our work, each heap buffer is enclosed by a pair of canaries; Whenever a buffer is overflowed, its canary is first corrupted. So once a canary is found tampered, a buffer overflow is detected. The classic canary-based buffer overflow checking was first proposed in Stackguard.
To the end of concurrent monitoring, one task is to collect canary addresses into some efficient data structures. Specifically the malloc/free calls in user threads are hooked to insert and delete canary addresses respectively, in the meanwhile the monitor thread traverses the data structure to locate and check canaries in an infinite loop. However there are a series of potential race conditions here, for example when the monitor thread checks a buffer, the buffer may have been deallocated, and the buffer memory may have been unmapped; Also the concurrent canary address insertion and removal are hazardous. Therefore, some concurrent data structures are needed here.
We first tried a lock-based red-black tree. The performance and scalability are not satisfactory, because user threads are blocked by the monitor thread frequently due to lock contention. So we turned to non-blocking synchronization and lock-free data structures.
In the second attempt, we utilized a state-of-the art lock-free hash table. In theory the insertion and removal operations can be finished in constant time and there is no blocking. However, our experiments show that the performance overhead is very high: the time for a pair of malloc and free calls is more than 5 times. One reason is the complex data structure operations; another reason is the contention between the user threads and the monitor thread. It shows that non-blocking is not magic; it usually requires the power of efficient, and sometimes specialized, lock-free data structures.



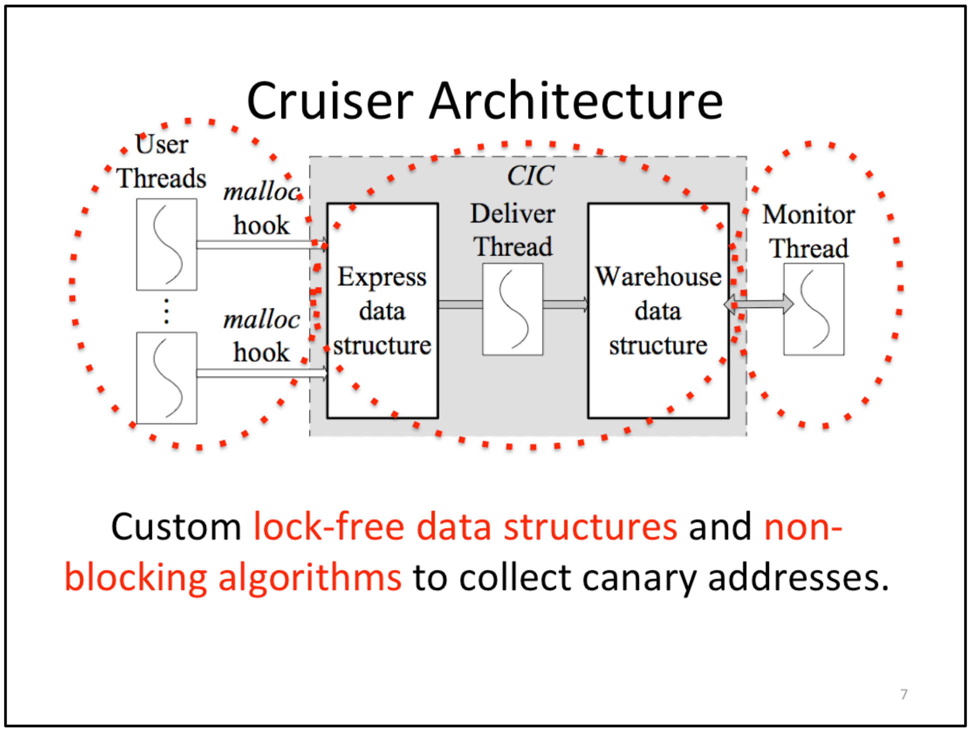
Challenges

To empower efficient concurrent monitoring, the synchronization should be light-weight:

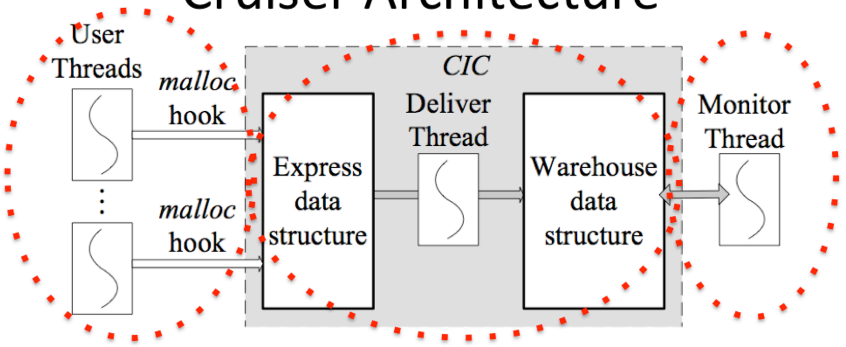
- Non-blocking synchronization
- Custom lock-free data structures

6

The major challenge for concurrent monitoring is synchronization overhead. According to the experience in the failed attempts, to achieve light-weight synchronization, user threads should not be blocked by the monitor thread; second, custom lock-free data structures should be designed for good performance and scalability.



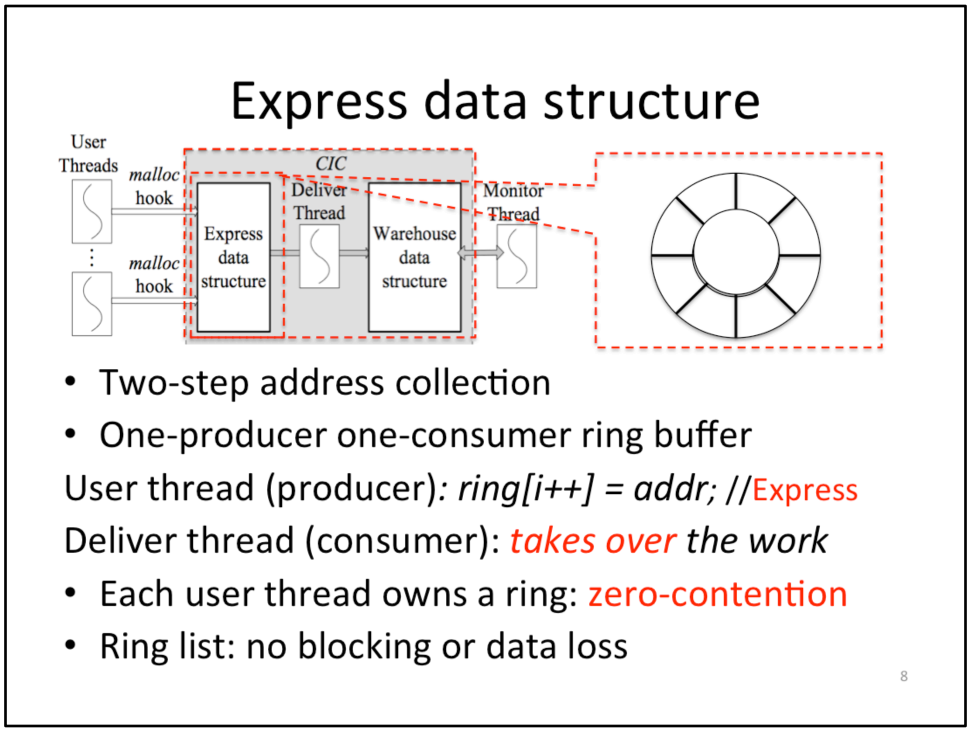
Cruiser Architecture



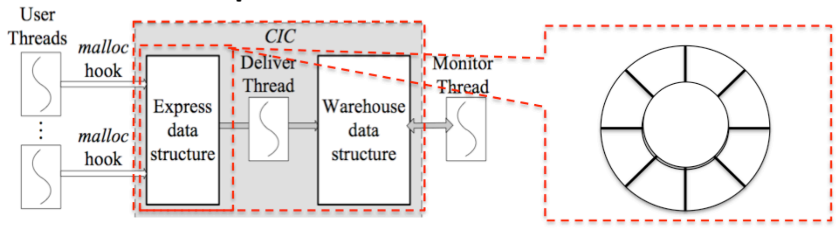
Custom lock-free data structures and non-blocking algorithms to collect canary addresses.

7

This is the architecture of Cruiser. The left part shows the interposition of the malloc function family, such that Cruiser is able to collect canary addresses. The middle part is the canary address collection component, which contains our custom lock-free data structures; the details will be presented later. The right part is the monitor thread, which traverses the lock-free data structure to perform concurrent buffer overflow checking. We will present the data structures and how the canary addresses are collected in a non-blocking and efficient fashion.



Express data structure



- Two-step address collection
- One-producer one-consumer ring buffer

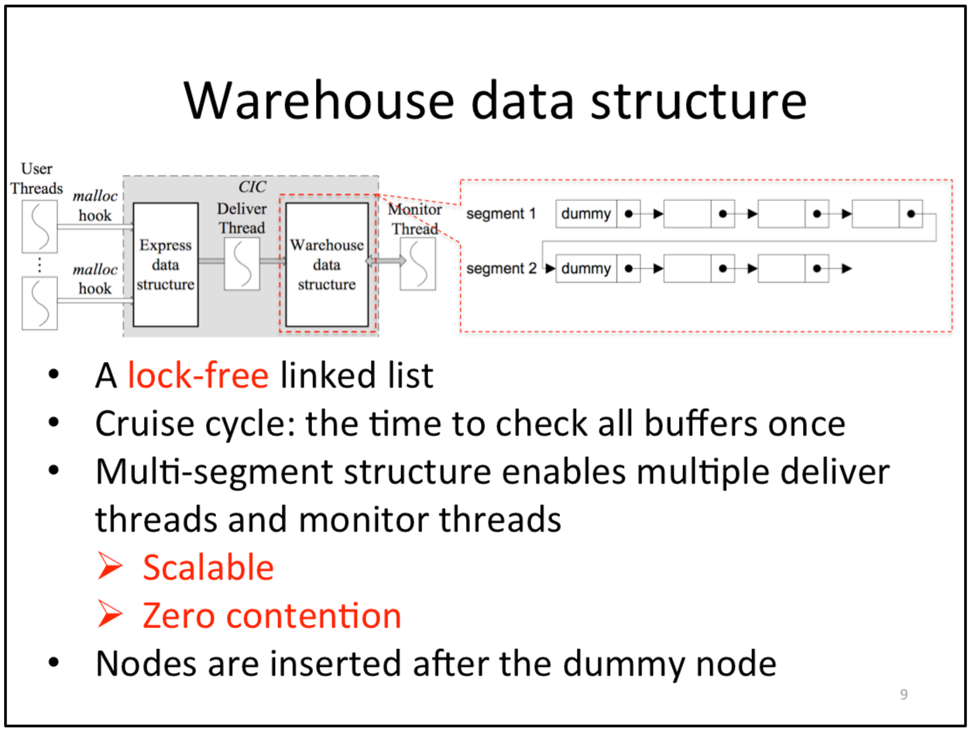
User thread (producer)*: ring[i++] = addr; //*Express

Deliver thread (consumer): *takes over the work*

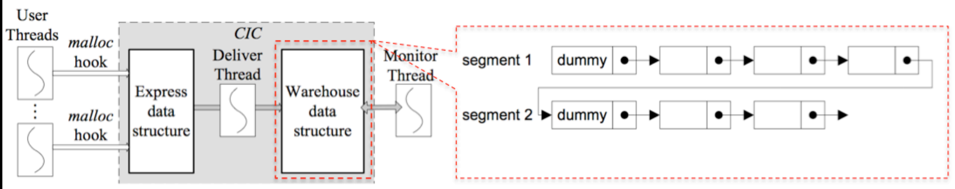
- Each user thread owns a ring: zero-contention
- Ring list: no blocking or data loss

8

The canary address collection component is composed of the express data structure, the deliver thread and the warehouse data structure. Canary addresses are collected in two steps: they are first put onto the Express data structure by the user threads, then moved to the Warehouse by the deliver thread. In this way, the effort of the user thread is minimized. We will discuss the component in detail.
Let's first take a close look at the Express data structure. The basic construct is a single-producer single-consumer ring buffer.
Base on the ring buffer, the hooked malloc call in the user thread simply copies the canary address onto the ring. The user thread only needs to do such a simple operation, and the rest work is taken over by the deliver thread. That is why we call it Express.
For multithreaded application, each user thread owns such as ring. Consequently, compared to accessing a shared data structure, the contention between user threads is eliminated.
Now consider what will happen if the ring is full: with the single ring buffer, the user thread has to either wait for empty entities or drop addresses. We extend the basic ring buffer to a ring list. Specifically when the current ring is full, the producer creates a bigger ring and puts addresses onto the new ring. All the rings created by a user thread are linked as list; the consumer retrieves canary addresses and deallocates ring buffers except for the last one. In this way, the producer doesn't need to wait or drop data, and the ring buffer will converge to a suitable size quickly. Actually, according to our experiments, this kind of ring list growth occurs quite infrequently.



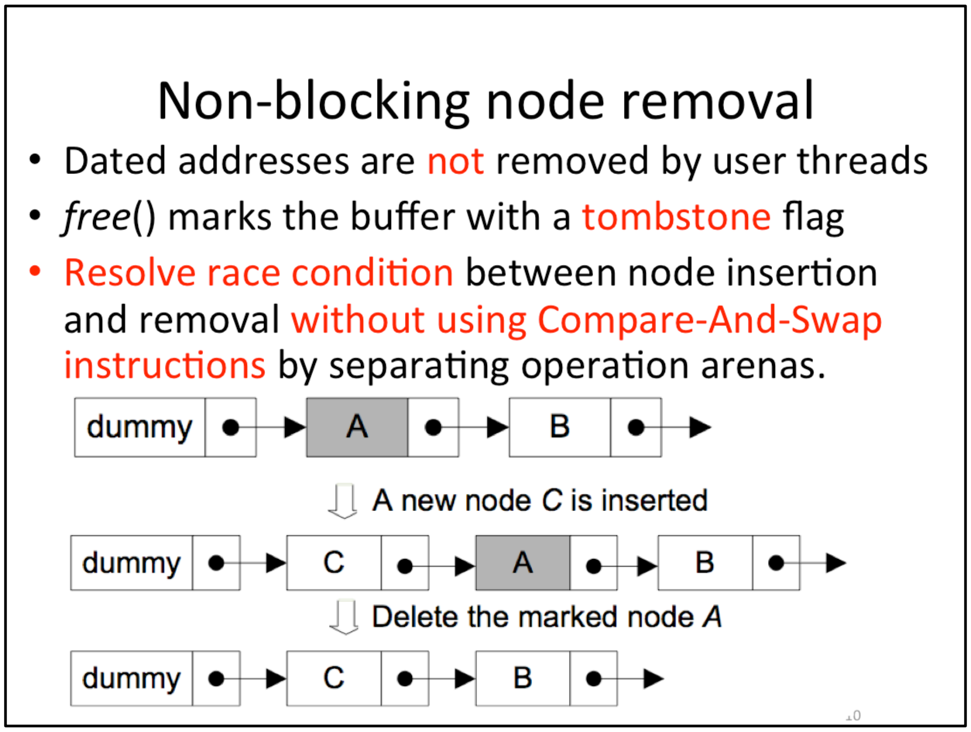
Warehouse data structure



- A lock-free linked list
- Cruise cycle: the time to check all buffers once
- Multi-segment structure enables multiple deliver threads and monitor threads
  - ➢ Scalable
  - ➢ Zero contention
- Nodes are inserted after the dummy node

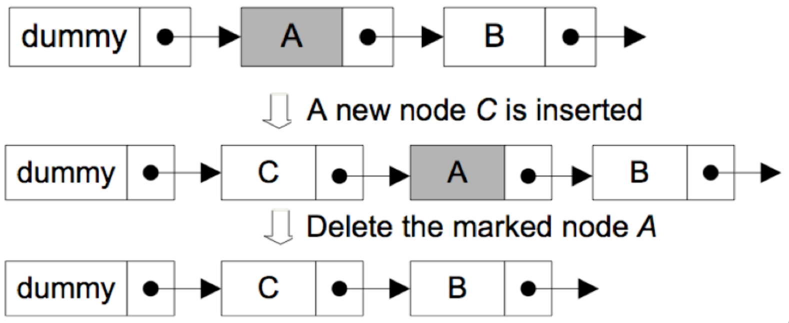
9

Although the ring is very efficient for canary address buffering, we still need a dynamic data structure to store a dynamic number of canary addresses. Warehouse was designed for this purpose, which is a lock-free linked list.
Recall that the monitor thread checks the buffers in an infinite loop. The time for the monitor thread to traverse through the list to check all the canaries is called cruise cycle. So any buffer is checked once per cruise cycle. Say, the cruise cycle is 1 microsecond, it means each buffer is checked 1000 times per second.
Warehouse is divided into segments. It's not a general-purpose linked list, as each segment allows for the access of one deliver thread and one monitor thread. Multiple segments are created for large-scale applications, which may want to deploy more than one deliver or monitor thread. With the multisegment structure, different deliver threads access different segments. Similarly, the access of different monitor threads doesn't overlap. In this way the contention is eliminated. The basic Cruiser has only one deliver thread and one monitor thread, so normally one segment is sufficient.
The segment itself is a linked list with a dummy node as the head. The dummy node is never removed and doesn't contain canary addresses. Nodes are inserted between the dummy node and the first genuine node by the deliver thread.



Non-blocking node removal

- Dated addresses are not removed by user threads
- *free*() marks the buffer with a tombstone flag
- Resolve race condition between node insertion and removal without using Compare-And-Swap instructions by separating operation arenas.



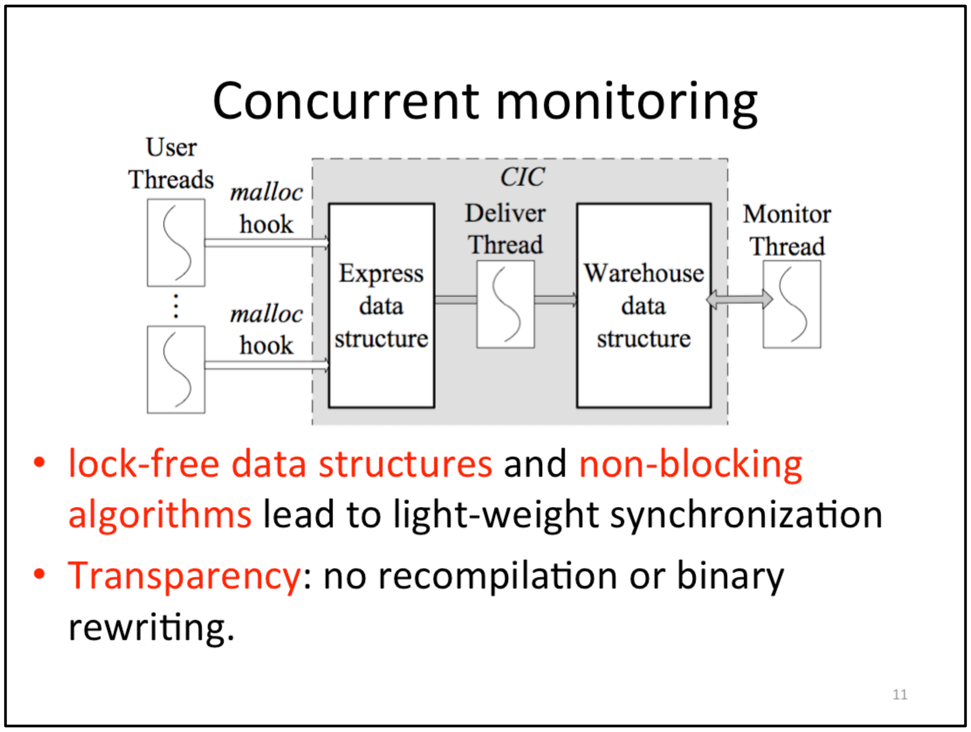
We have examined canary address insertion, now let's consider dated address removal. After heap buffers are deallocated, the corresponding canary addresses should be removed. Different from the aforementioned attempts which remove dated addresses inside free calls thus may heavily delay program execution, Cruiser makes the monitor thread do house-keeping. The free call only needs to mark the buffer with a tombstone flag; when the monitor thread scans the buffer and finds the flag, it will remove the corresponding addresses from Warehouse.

As aforementioned, a Warehouse segment is a linked list with a never-removed dummy node as its head. The deliver thread inserts the new node between the dummy node and the first genuine node, that is the node A, in the graph. However in the meanwhile the node A may be removed by the monitor thread. Without synchronization, the list may be corrupted.

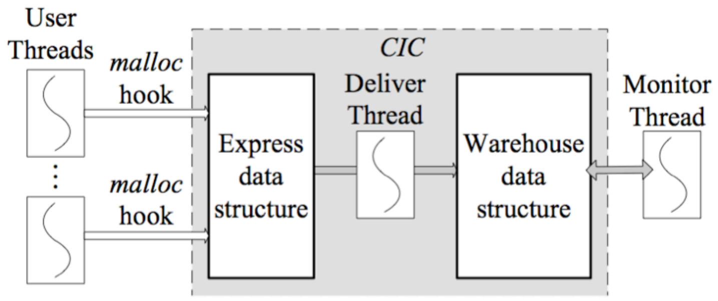
A conventional solution in non-blocking programming is to use Compare-And-Swap instructions to perform insertion and removal. However CAS instructions are relatively expensive and the contention between the deliver thread and the monitor thread introduces further overhead. We resolve the race conditions without the assistance of CAS instructions by guaranteeing that the monitor thread does not touch the link between the dummy node and the first genuine node.

Specifically, when the node A becomes dated, it is not removed directly. Instead, the monitor thread first marks it as to-be-removed and goes on with its traverse. In other words, the monitor thread does not remove the first genuine node for the time being.

When a new node, C, is to be inserted, it can always be inserted without the concern of race conditions. After node C is inserted, node A is not the first genuine node any longer, so it can be removed by the monitor thread as on a single-threaded list. The contention is completely eliminated as the deliver thread and the monitor thread operate in different arenas.



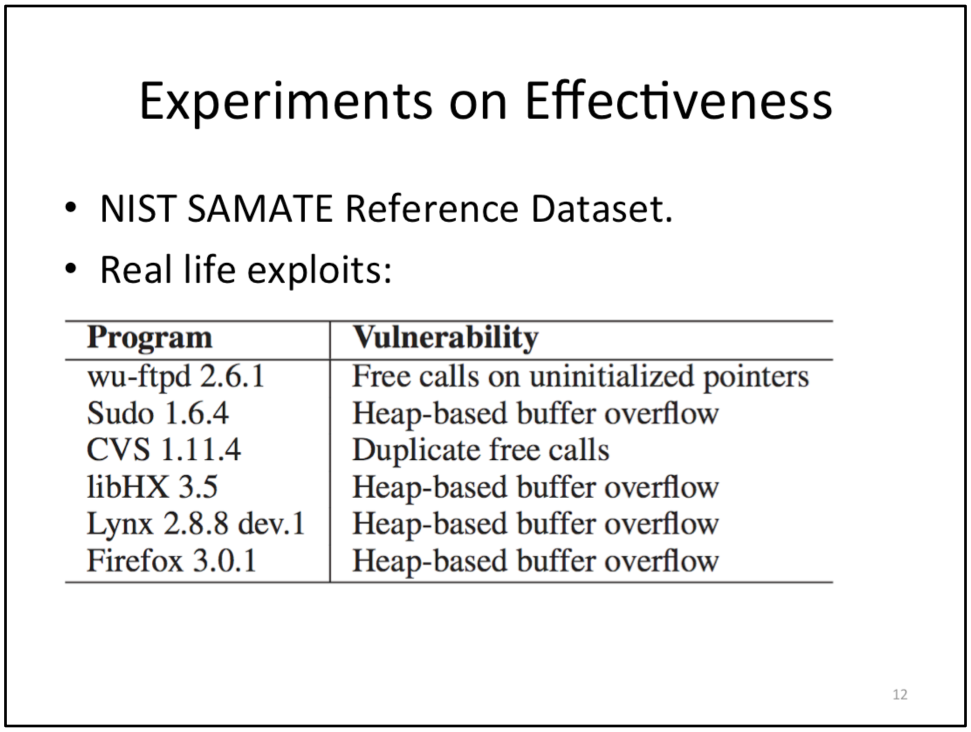
Concurrent monitoring



- lock-free data structures and non-blocking algorithms lead to light-weight synchronization
- Transparency: no recompilation or binary rewriting.

11

Based on the lock-free data structures and non-blocking algorithms, we achieved concurrent monitoring with high efficiency and scalability.
Cruiser is implemented as a dynamically linked library; it can be deployed to protect dynamically linked applications transparently without recompilation or binary rewriting.



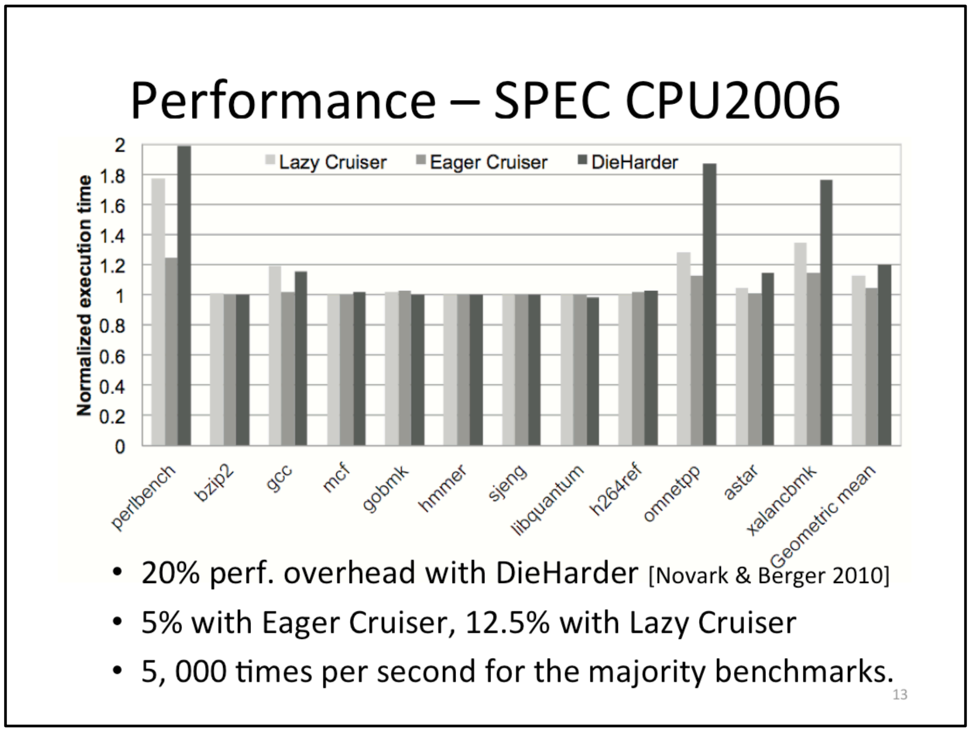
Experiments on Effectiveness

- NIST SAMATE Reference Dataset.
- Real life exploits:

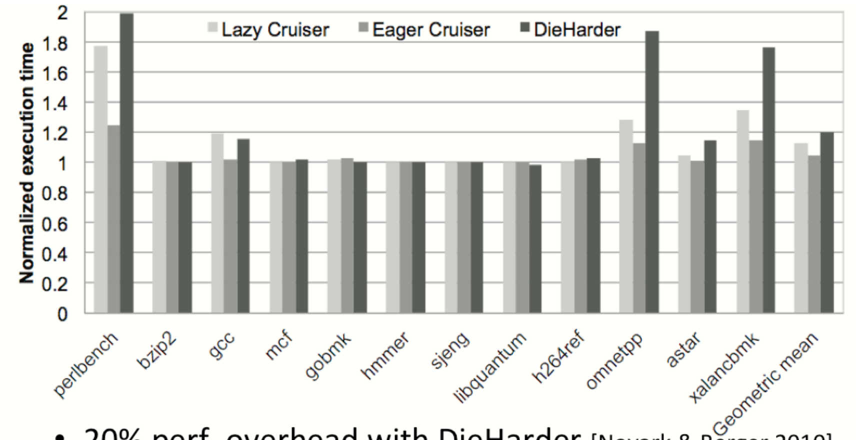
Program	**Vulnerability**
wu-ftpd 2.6.1	Free calls on uninitialized pointers
Sudo 1.6.4	Heap-based buffer overflow
CVS 1.11.4	Duplicate free calls
libHX 3.5	Heap-based buffer overflow
Lynx 2.8.8 dev.1	Heap-based buffer overflow
Firefox 3.0.1	Heap-based buffer overflow

12

We have tested Cruiser with the NIST SRD reference which contains a variety of heap buffer overflows, as well as some real life exploits. Cruiser detected all the buffer overflows and other heap vulnerabilities such as duplicate free calls. It can also detect heap memory leakage.



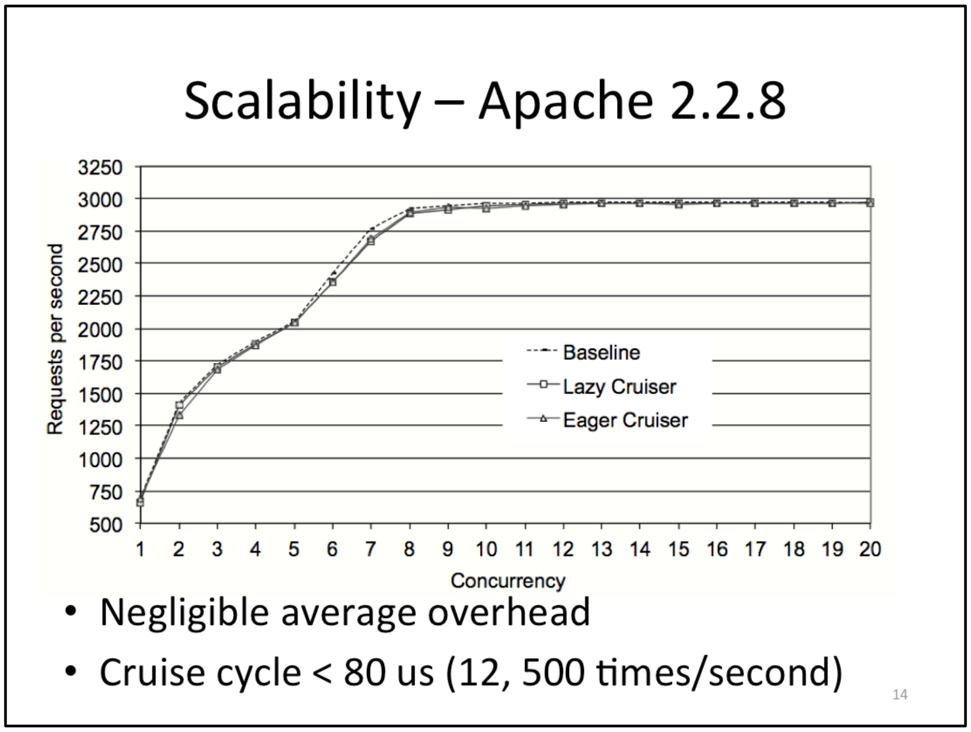
Performance – SPEC CPU2006



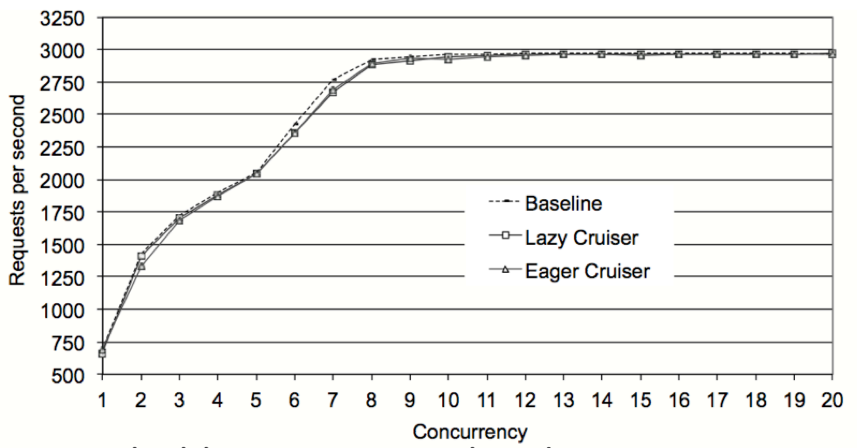
- 20% perf. overhead with DieHarder [Novark & Berger 2010]
- 5% with Eager Cruiser, 12.5% with Lazy Cruiser
- 5, 000 times per second for the majority benchmarks.

13

We ran SPEC CPU 2006 experiments on a eight core machine with 4GB RAM, and compare the results of DieHarder, which also enhances heap security.
Compared to 20% average overhead imposed by DieHarder, two variants of Cruiser imposes only 5% and less than 13% performance overhead, respectively. The maximum average cruise cycle is 120 microseconds. The majority of the 12 benchmarks have cruise cycles less than point two microsecond. In other words, Cruiser checks through the heap memory 5, 000 times per second.



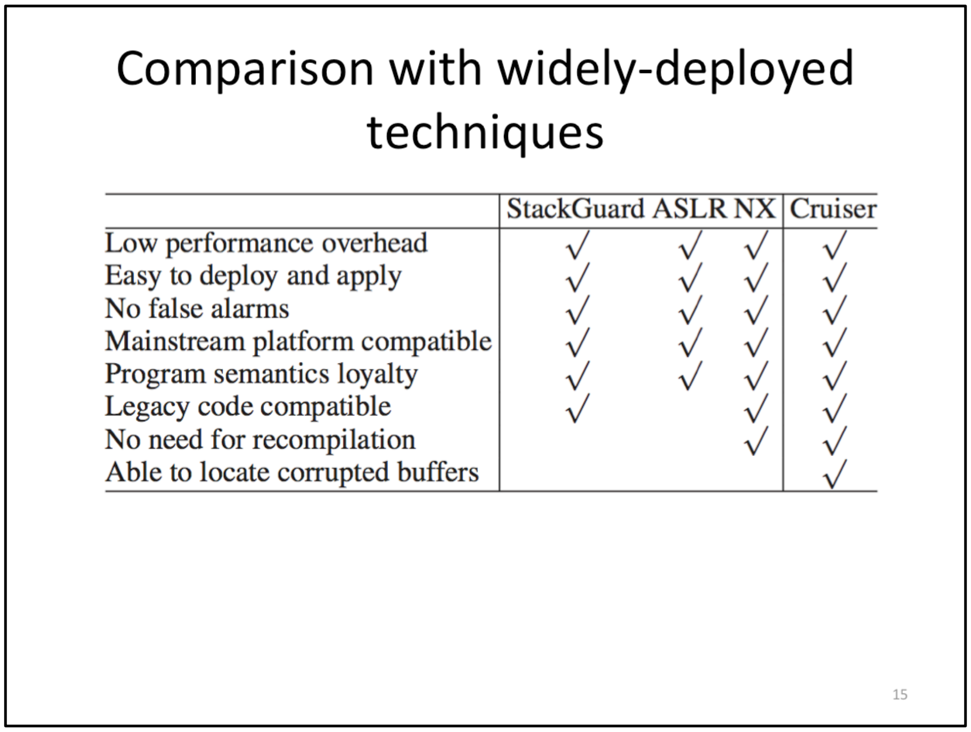
Scalability – Apache 2.2.8



- Negligible average overhead
- Cruise cycle < 80 us (12, 500 times/second)

14

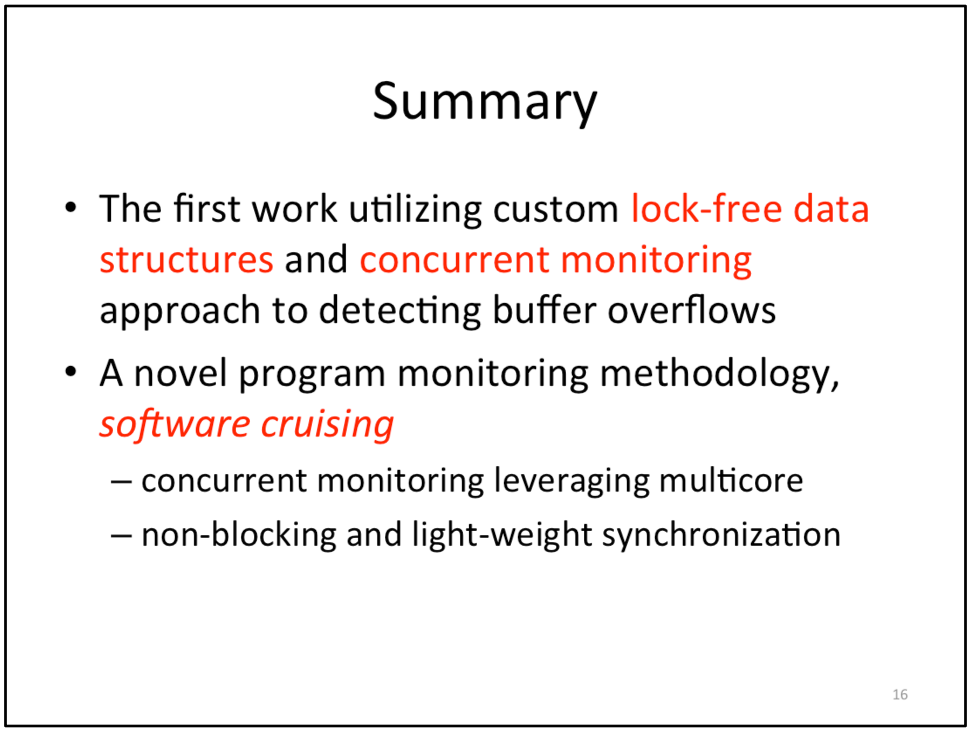
We also run cruiser with multithreaded Apache, In this graph, The X axis is the concurrent request number and the Y axis is the throughput of Apache.
As we can see the average overhead is negligible. The average cruise cycle is less than 80 milliseconds, that is any buffer is checked more than 12 thousand times per second.



Comparison with widely-deployed techniques

	StackGuard	ASLR	NX	Cruiser
Low performance overhead	√	√	√	√
Easy to deploy and apply	√	√	√	√
No false alarms	√	√	√	√
Mainstream platform compatible	√	√	√	√
Program semantics loyalty	√	√	√	√
Legacy code compatible	√		√	√
No need for recompilation			√	√
Able to locate corrupted buffers				√

We compared Cruiser with other widely-deployed buffer overflow countermeasures including StackGuard, ASLR and Non-executable heap, and it shows that Cruiser shares many good properties with them, such as low performance overhead and easiness to deploy and apply.

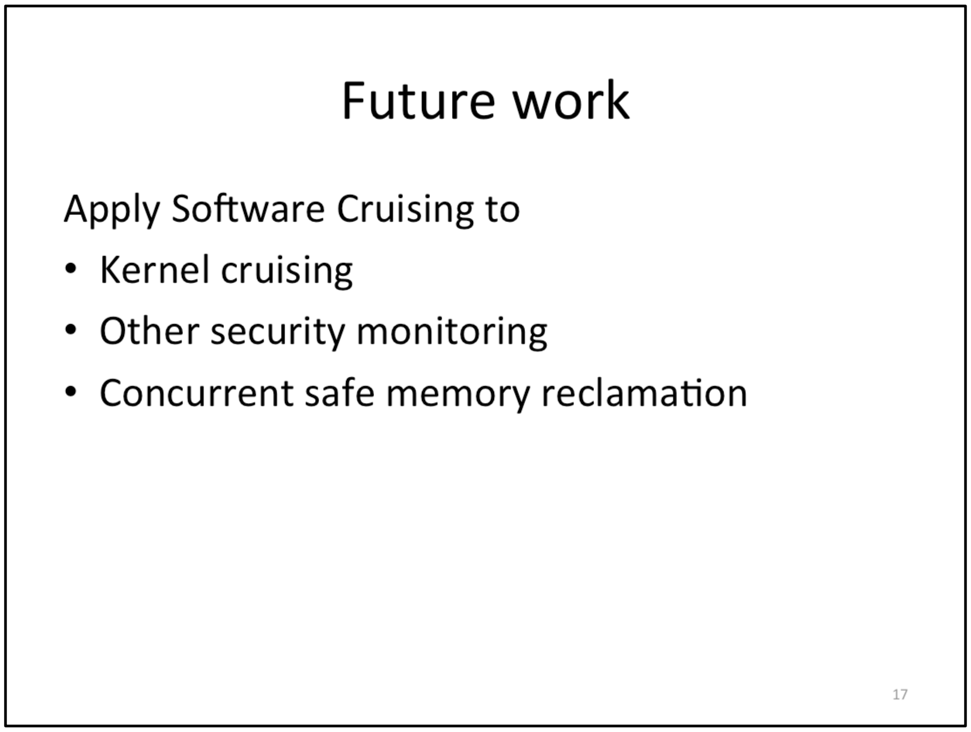


Summary

- The first work utilizing custom lock-free data structures and concurrent monitoring approach to detecting buffer overflows
- A novel program monitoring methodology, *software cruising*
  - concurrent monitoring leveraging multicore
  - non-blocking and light-weight synchronization

16

In conclusion, to the best of knowledge, Cruiser is the first work utilizing custom lock-free data structures and concurrent monitoring approach to detecting buffer overflows. We proposed a novel program monitoring methodology, named software cruising. The major properties include concurrent monitoring leveraging multicore architectures, non-blocking and light-weight synchronization,



Future work

Apply Software Cruising to

- Kernel cruising
- Other security monitoring
- Concurrent safe memory reclamation

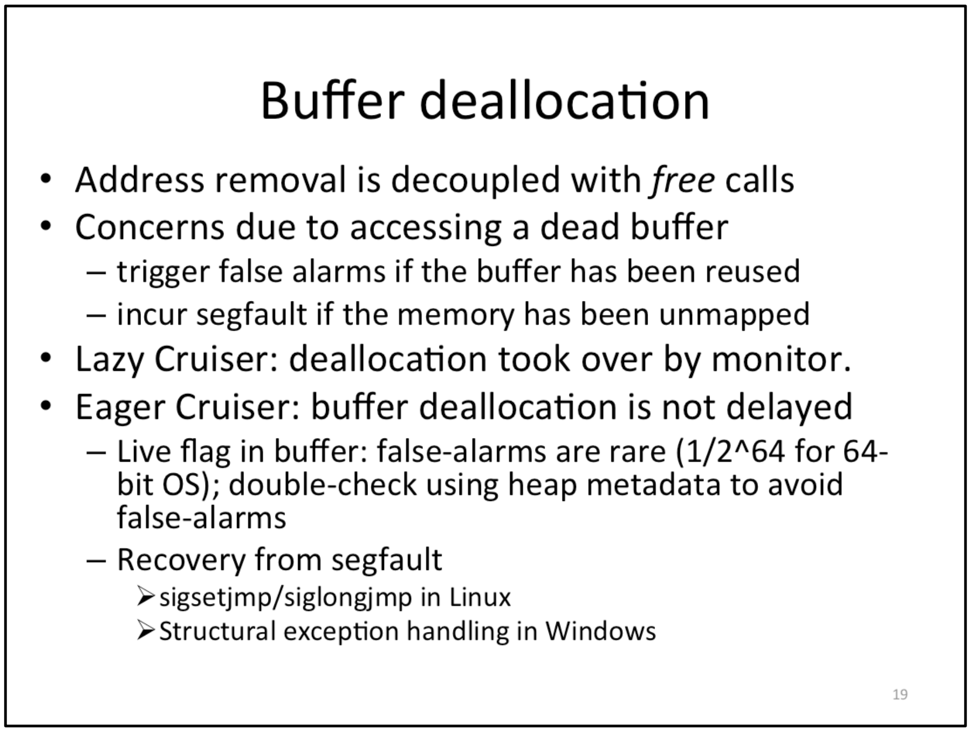
17

For our future work, we are applying software cruising to Operating System kernels to enhance system security. We will apply software cruising to other security monitoring. It's also potential to be applied to concurrent safe memory reclamation, which is a fundamental problem for lock-free data structures.

Thank you!

Questions?

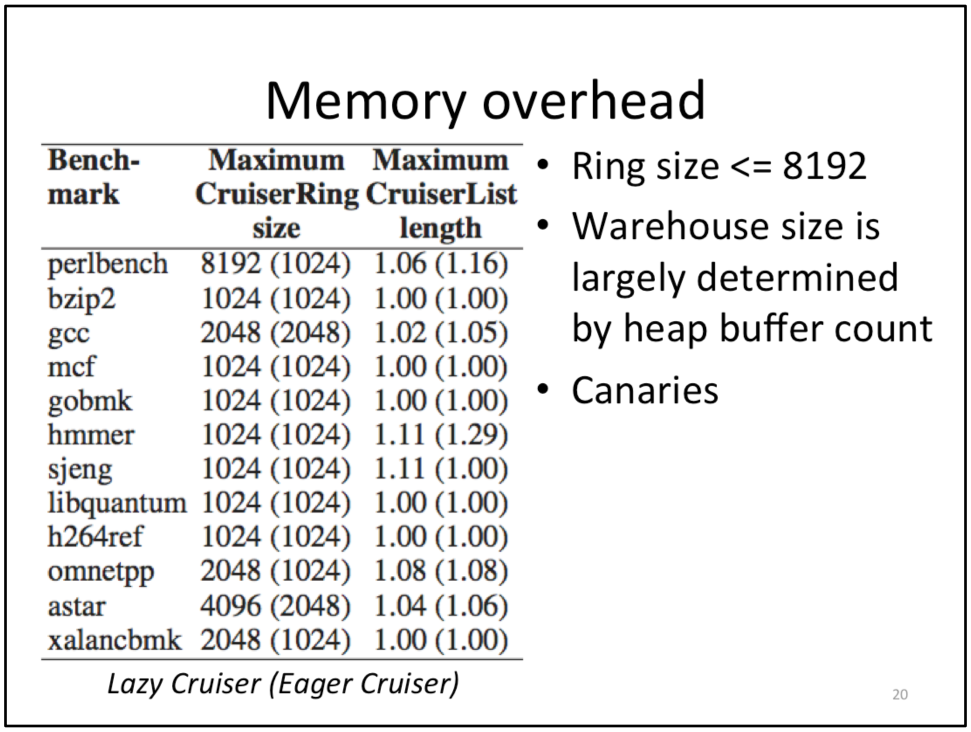
18



Buffer deallocation

- Address removal is decoupled with *free* calls
- Concerns due to accessing a dead buffer
  - trigger false alarms if the buffer has been reused
  - incur segfault if the memory has been unmapped
- Lazy Cruiser: deallocation took over by monitor.
- Eager Cruiser: buffer deallocation is not delayed
  - Live flag in buffer: false-alarms are rare (1/2^64 for 64-bit OS); double-check using heap metadata to avoid false-alarms
  - Recovery from segfault
    - ➢ sigsetjmp/siglongjmp in Linux
    - ➢ Structural exception handling in Windows

19



Memory overhead

Bench-mark	Maximum CruiserRing size	Maximum CruiserList length
perlbench	8192 (1024)	1.06 (1.16)
bzip2	1024 (1024)	1.00 (1.00)
gcc	2048 (2048)	1.02 (1.05)
mcf	1024 (1024)	1.00 (1.00)
gobmk	1024 (1024)	1.00 (1.00)
hmmer	1024 (1024)	1.11 (1.29)
sjeng	1024 (1024)	1.11 (1.00)
libquantum	1024 (1024)	1.00 (1.00)
h264ref	1024 (1024)	1.00 (1.00)
omnetpp	2048 (1024)	1.08 (1.08)
astar	4096 (2048)	1.04 (1.06)
xalancbmk	2048 (1024)	1.00 (1.00)

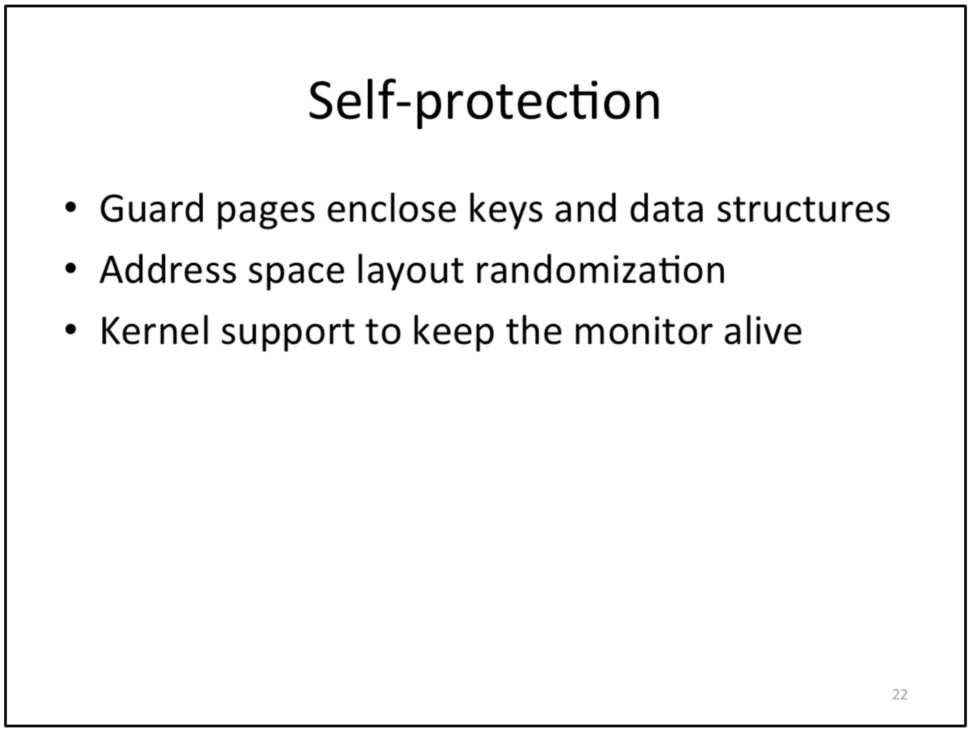
Lazy Cruiser (Eager Cruiser)

- Ring size <= 8192
- Warehouse size is largely determined by heap buffer count
- Canaries

20

Raised the bar for attacks

- Probabilistic prevention:
  - Exploit latency T1, cruise cycle T2
  - If T1 >= T2, definitely thwart exploits
  - If T1 < T2, probability is T1/T2
- Minimal evil time: less than one cruise cycle; typically on the order of microseconds.



Self-protection

- Guard pages enclose keys and data structures
- Address space layout randomization
- Kernel support to keep the monitor alive

22