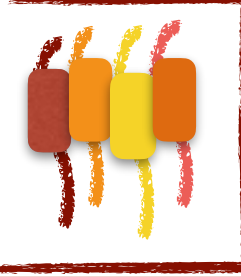


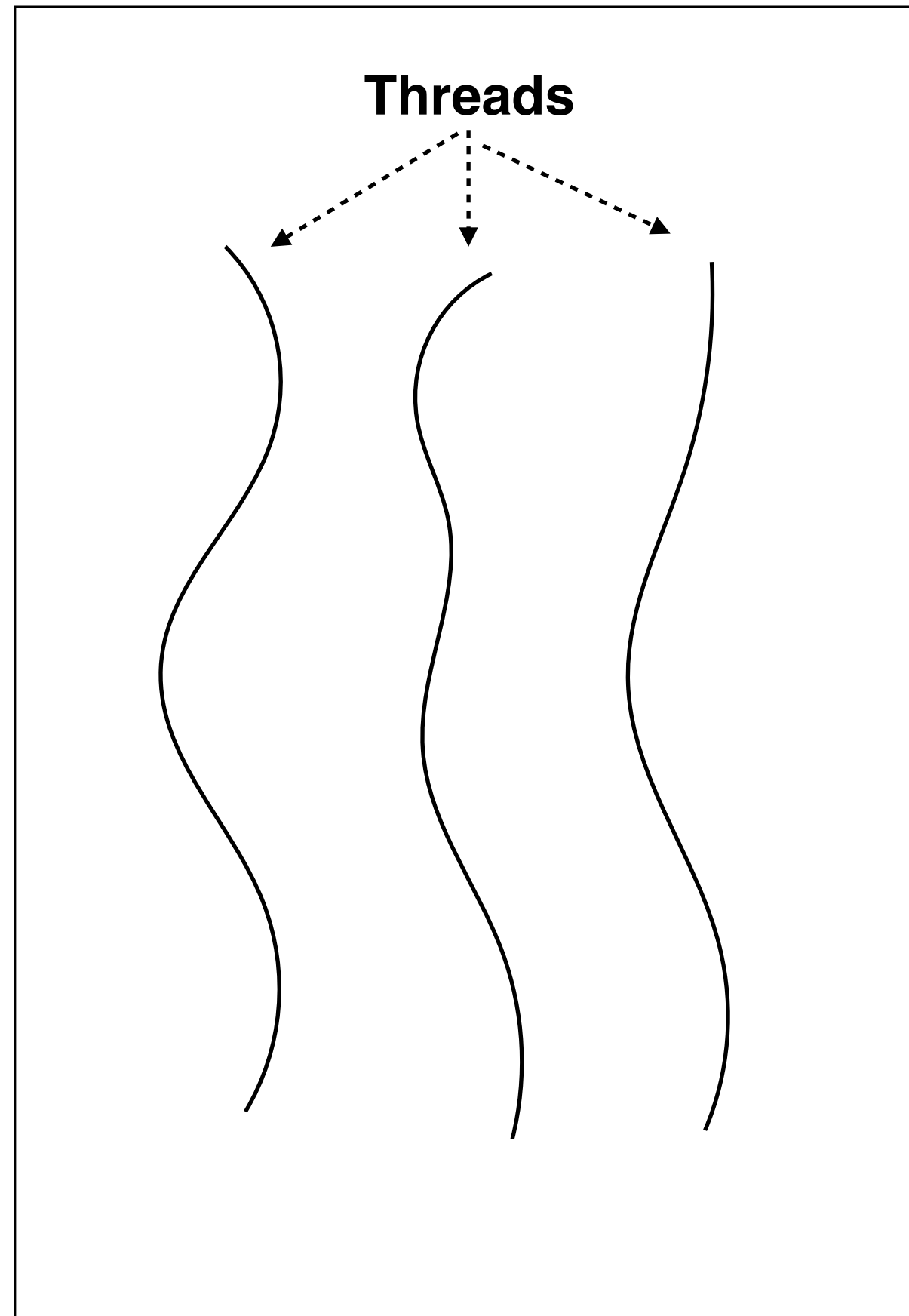
Shreds: **Fine-grained Execution Units with Private Memory**

Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, Long Lu

RiS3 Lab / Computer Science / Stony Brook University



A Process

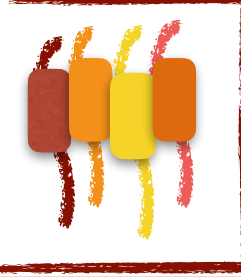


• Traditional Execution Units

- Processes
 - Separate address spaces
- Threads
 - Sharing one address space

**IT'S NOT
ENOUGH**

In-process Memory Abuses



- **Definition:**

Malicious or compromised components try to steal data or execute code of other components running in the same process.

- **Two examples**

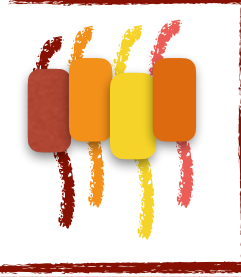
- **Stealing secret data**

- The Heartbleed bug 

- **Executing private code**

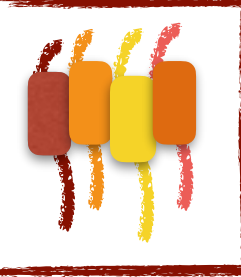
- Private APIs in iOS Apps

Potential Mitigations of in-Process Abuse

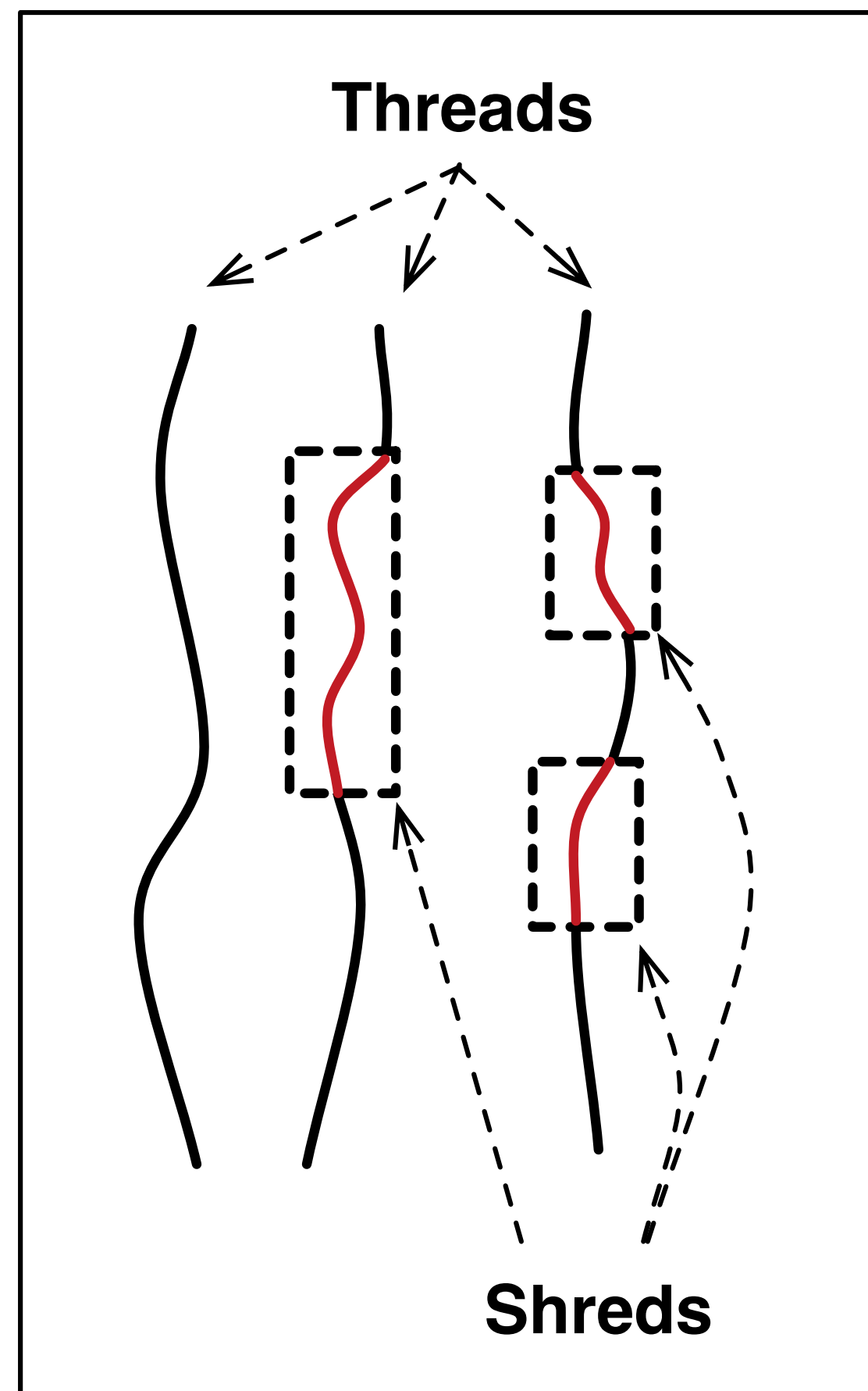


Techniques	Why unsuitable
Process-level isolation <i>(OpenSSH, Chrome)</i>	<ul style="list-style-type: none">• IPC is expensive• Adoption effort
Software fault isolation-like techniques <i>(Native Client)</i>	<ul style="list-style-type: none">• Require instrumenting untrusted code• Ineffective on dynamic or external code
Hardware-assisted techniques <i>(SGX, Trustzone)</i>	<ul style="list-style-type: none">• Overly restrictive execution environment• Semantic gap

Introducing Shred



A process



- **Shred**
 - Arbitrarily scoped segment of a thread execution
- **S-pool**
 - The private memory pool for each shred
- **Shred APIs & OS-level supports**

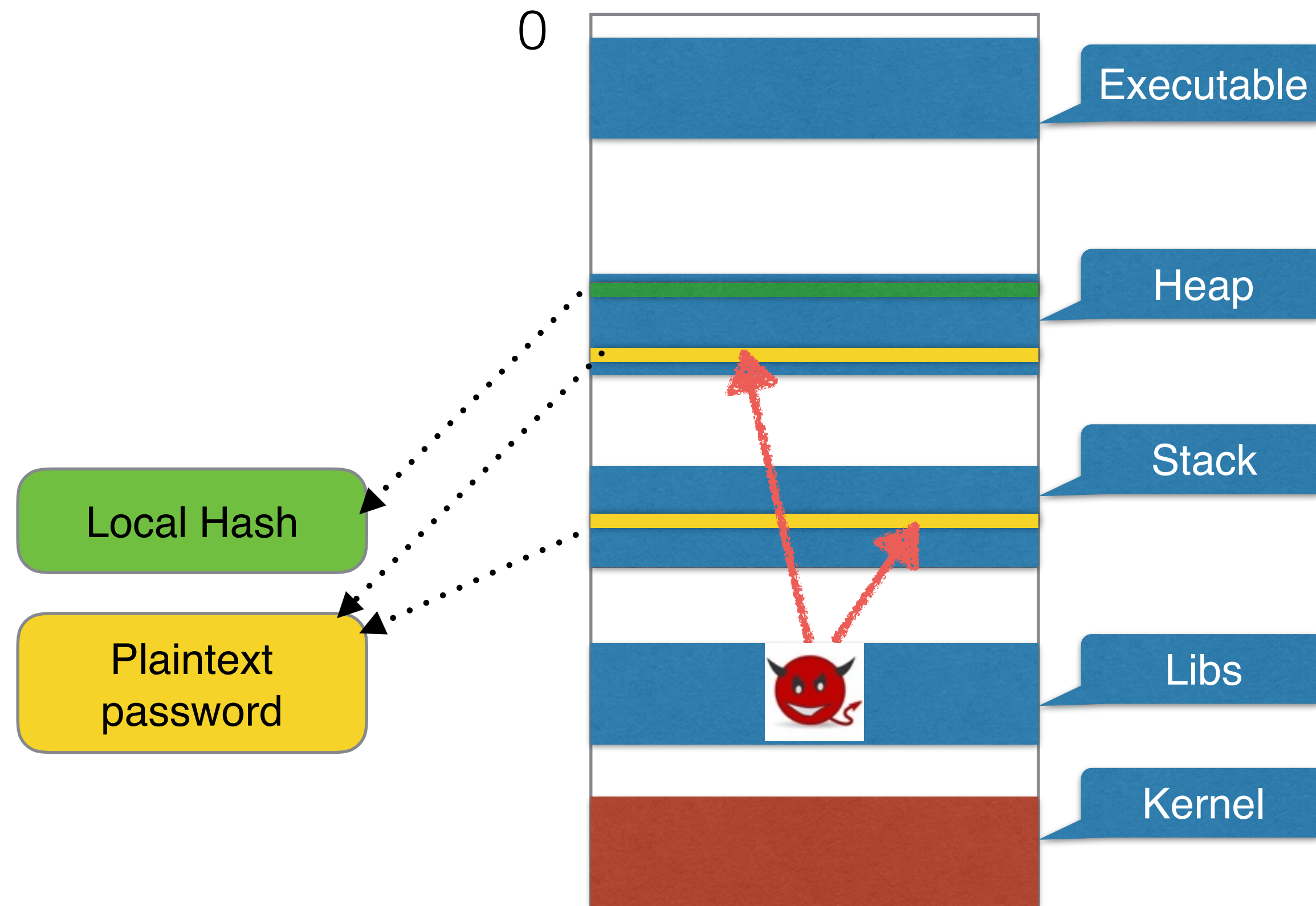
Threat Model

- Trusted OS
- Untrusted component

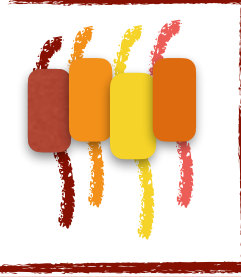


Example Use Case

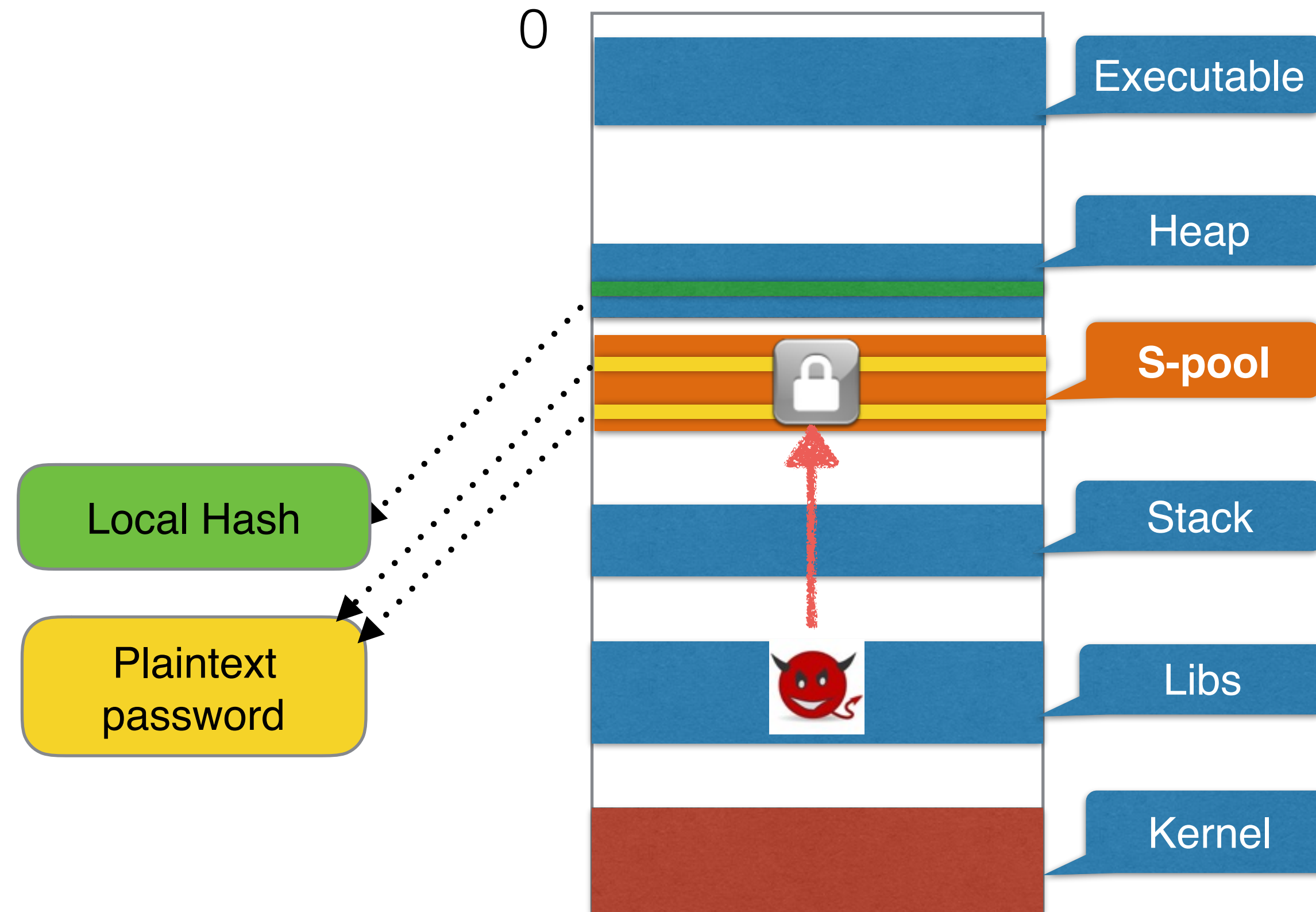
- Password authentication on web server(w/o shred)

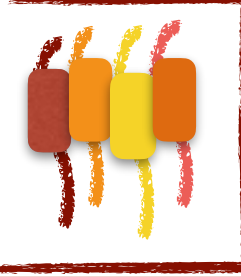


Example Use Case cont.



- Password authentication on web server(w/ shred)





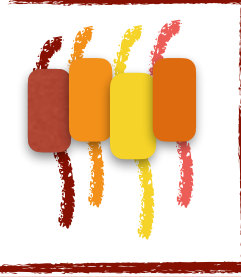
- `err_t shred_enter(int pool_desc);`
 - Start a shred execution on the current thread
 - Unlock s-pool
- `err_t shred_exit();`
 - Terminate a shred execution
 - lock down the s-pool

Shred creation APIs

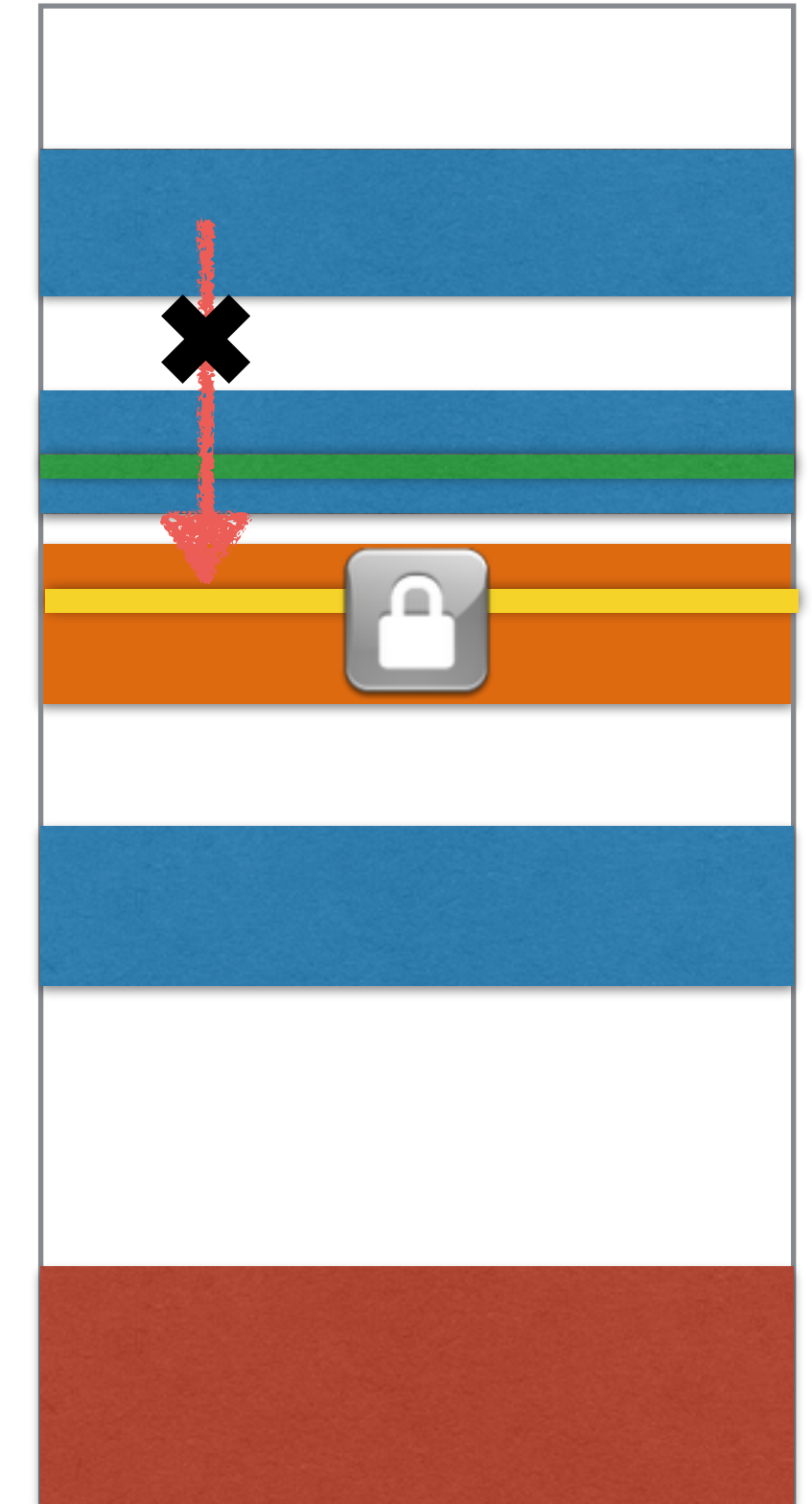
- `void * spool_alloc(size_t size);`
 - Allocate memory inside S-pool
- `err_t spool_free(void *ptr);`
 - Free memory inside S-pool

S-pool allocation APIs

Code Example—Lighttpd

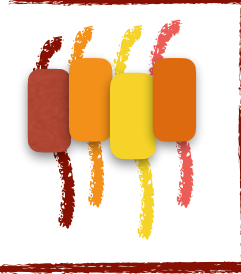


```
int http_request_parse(server *srv, connection *con) {
    ...
    char *cur; /* to receive password */
+   if (strncmp(cur, auth_str, auth_str_len)==0) {
+       shred_enter(AUTH_PASSWD_POOL);
+       /* receive and save password */
+       data_string *ds = s_ds_init();
+       int pw_len = get_passwd_length(cur);
+       cur += auth_str_len + 1;
+       buffer_copy_string_len(ds->key, auth_str, auth_str_len);
+       buffer_copy_string_len(ds->value, cur, pw_len);
+       cur += pw_len;
+       shred_exit();
+   }
    ...
}
```



Listing 1: *lighttpd/src/request.c*

Code Example cont.



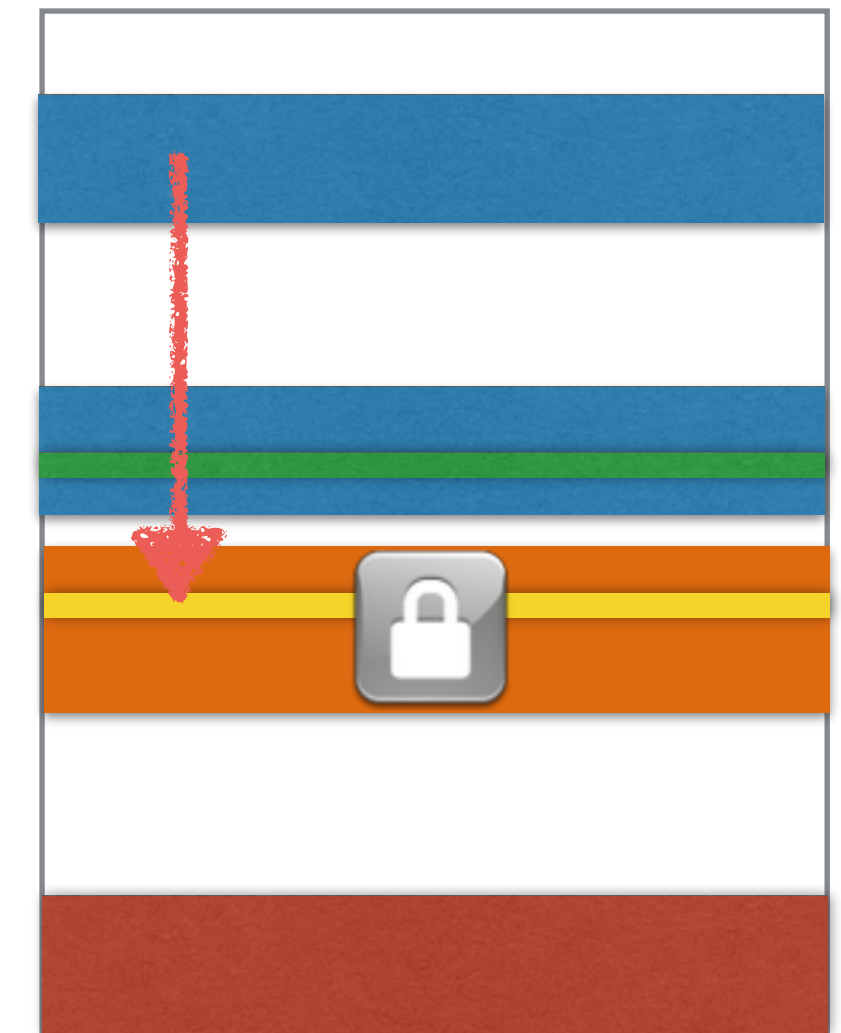
```
/* called inside a shred */
data_string *s_ds_init(void) {
    data_string *ds;
+   ds = spool_alloc(sizeof(*ds));
    ...
    return ds;
}
```

```
/* called inside a shred */
void s_ds_free(data_string *ds) {
+   ...
+   spool_free(ds->key);
+   ...
    return;
}
```

S-pool allocation APIs wrapper

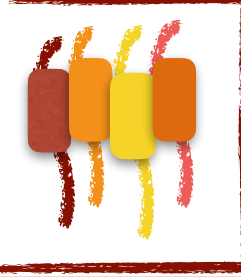
Listing 2: *lighttpd/src/data_string.c*

```
...
/* inside HTTP auth module */
+   shred_enter(AUTH_PASSWD_POOL);
/* ds points passwd obj in spool */
    http_authorization = ds->value->ptr;
/*hash passwd and compare with local copy*/
+   s_ds_free(ds);
+   shred_exit();
...
```

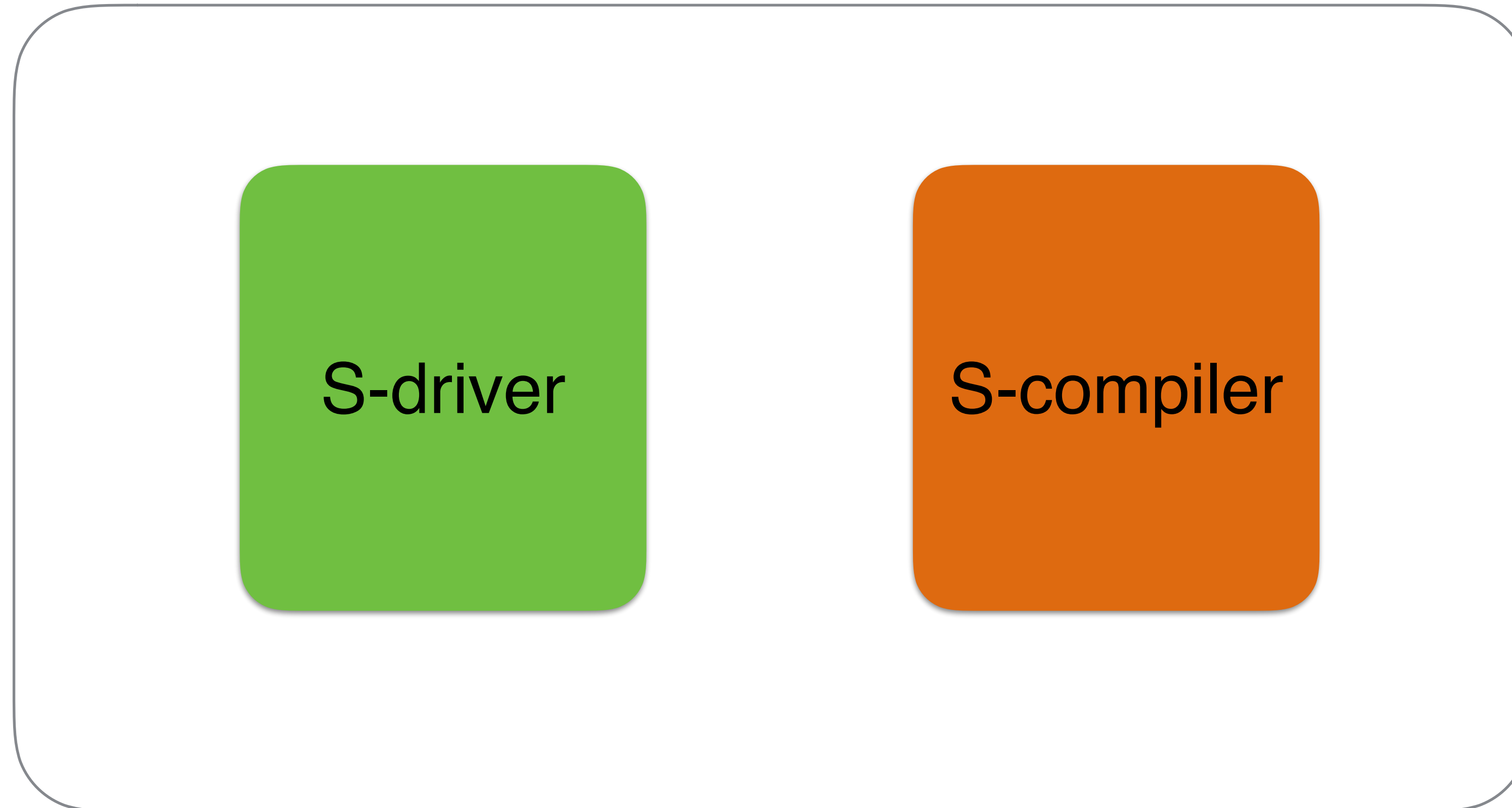


Listing 3: *lighttpd/src/mod_auth.c*

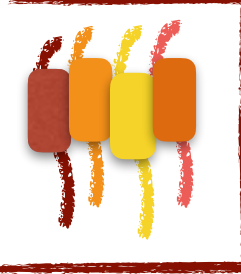
System overview



- **Two major components**



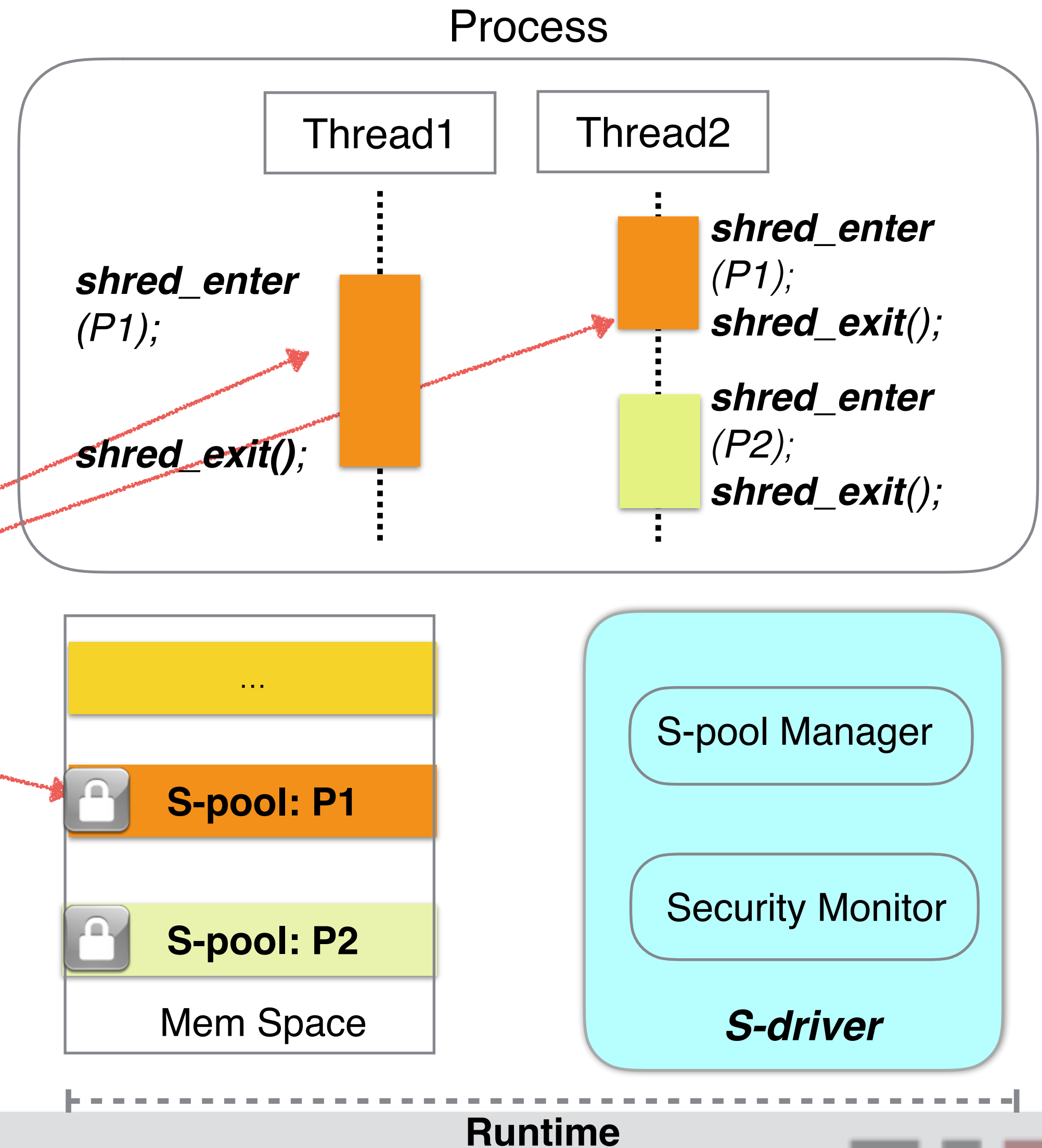
System Component: S-driver



S-driver

- Entry/exit of shreds
- S-pool (de)allocations
- Controls the access to S-pools

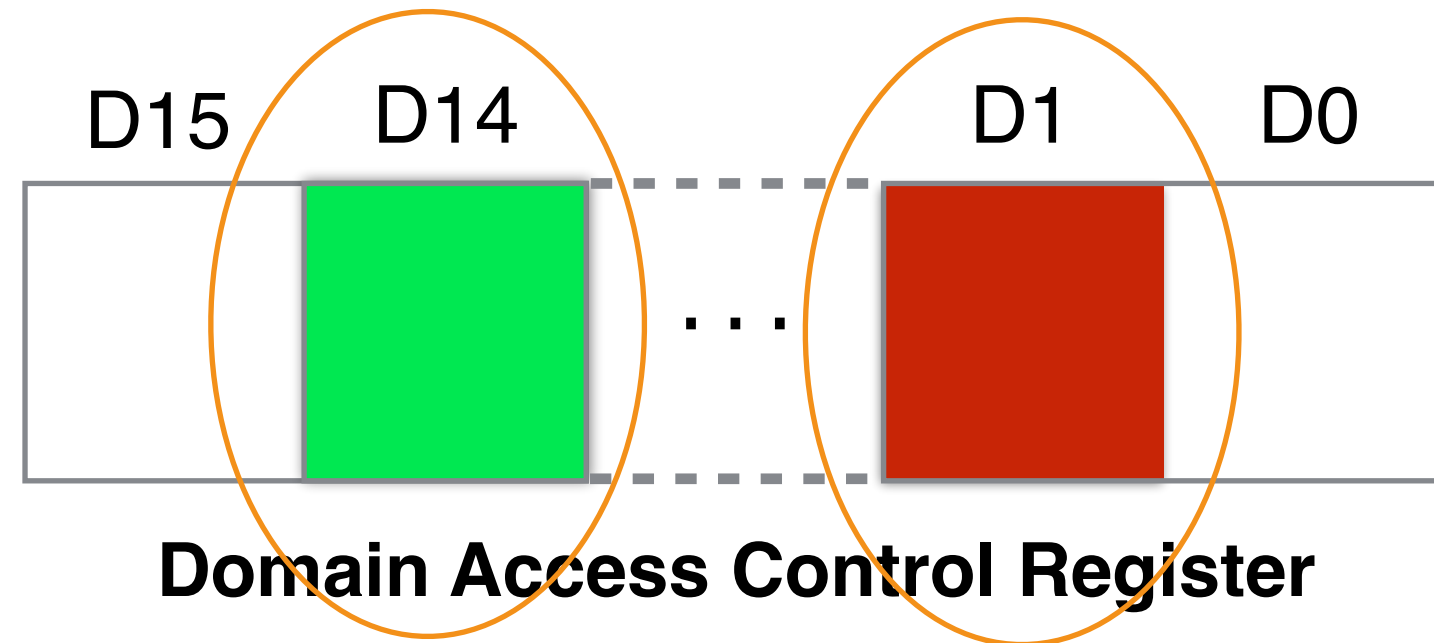
S-pool sharing





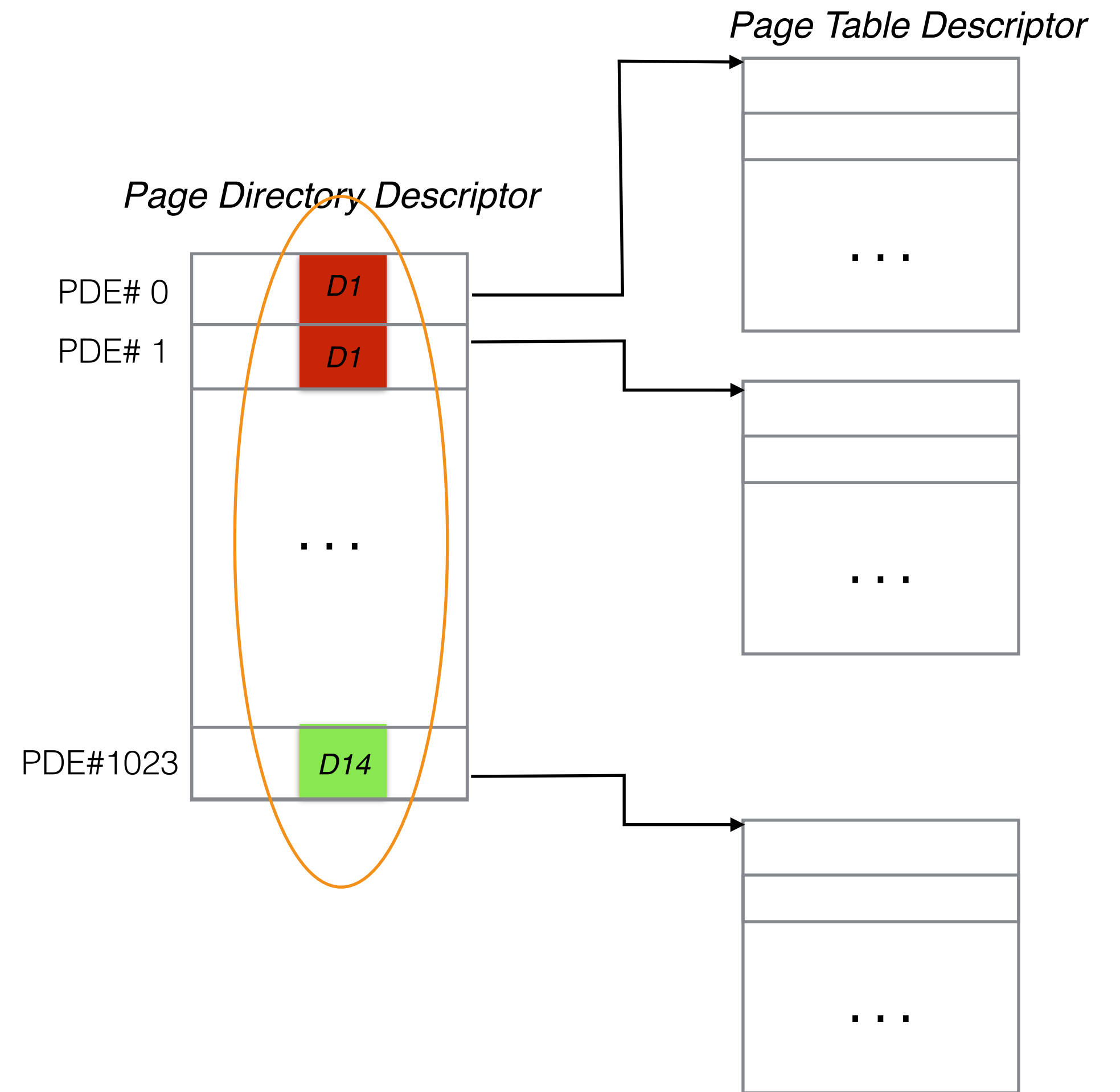


How S-pool is Built

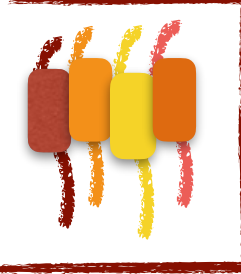
Intel: Memory protection keys — The building block



 : Accessible
 : Not accessible



Challenges & Solutions



- ARM Memory Domains

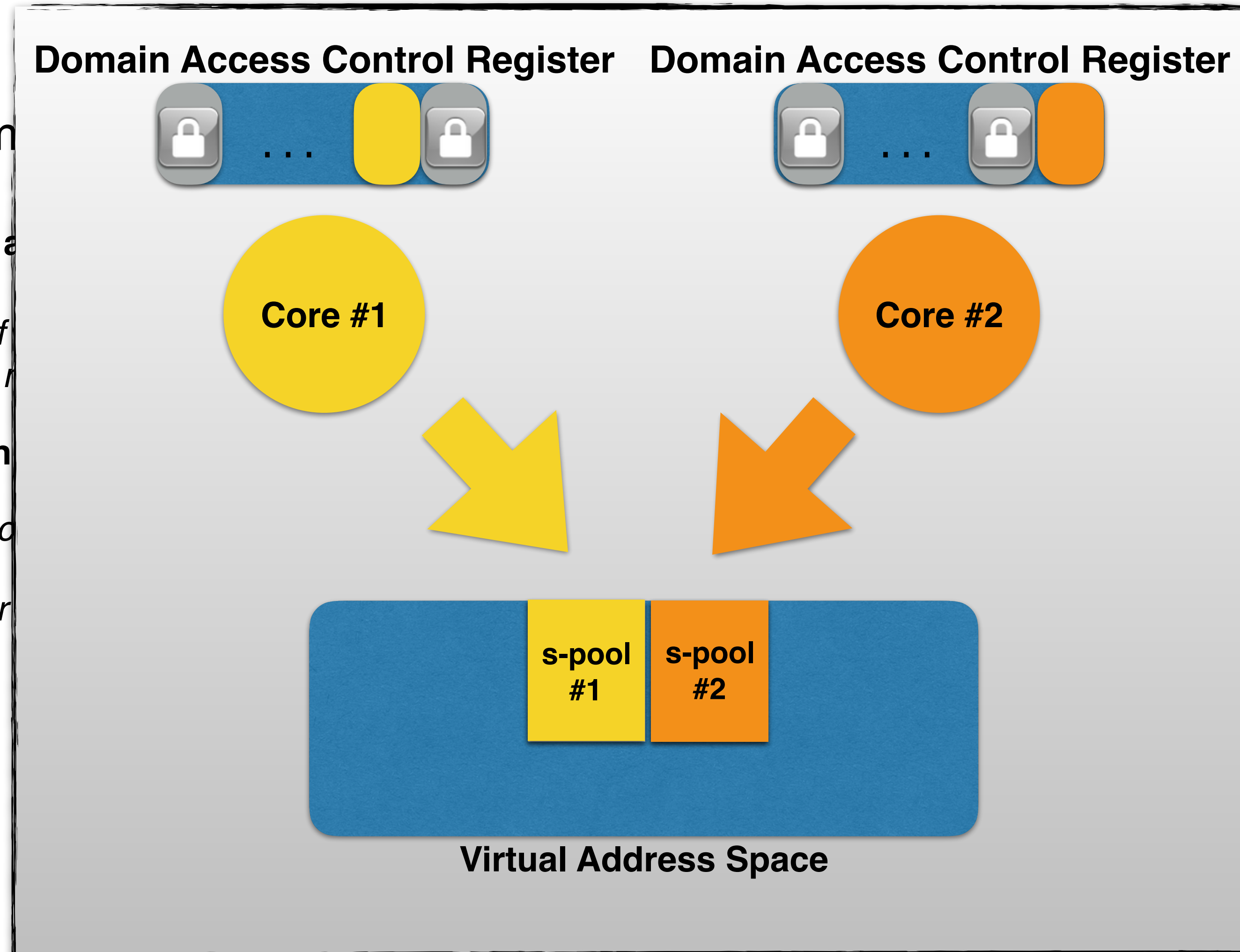
- 1) The granularity of the address space

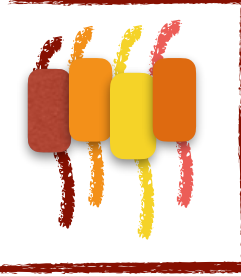
- ✓ Create the notion of a domain and use S-driver to manage it

- 2) Limited Domains: Only one per core

- ✓ Statically bind an address space to a core

- ✓ Reuse a domain for multiple cores



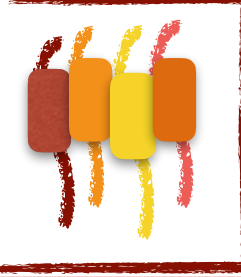


S-driver will,

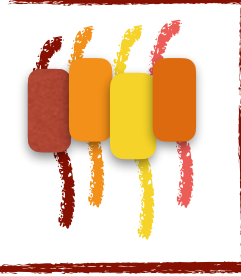
- Lock s-pool when,
 - Shred exits
 - Context-switch Out
 - Asynchronous events: signal handling, etc

- Unlock s-pool when,
 - Shred enters
 - Context-switch in
 - Resuming from asynchronous events

Moving the Domain Adjustments Off the Critical Path



- Changing PDE is relatively cumbersome
 - Page table walking
 - TLB invalidation
- TWO knobs to control the accessibility of S-pool
 - Domain of the corresponding page table entry
 - Value of corresponding DACR entry
- Changing DACR value is much faster, only one instruction
 - `MCR p15, 0, <Rd>, c3, c0, 0 ; Write DACR`
 - Develop the **domain fault handler** to handle domain fault *lazily*
 - Detecting attacks
 - Recover from legitimate domain faults



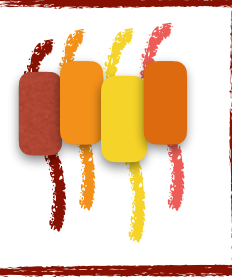
Secure stacks

- Each shred has a secure stack allocated from its s-pool

System interface protection

- ptrace()
- /dev/mem
- Directly read secret from file
- etc

System Component: S-compiler



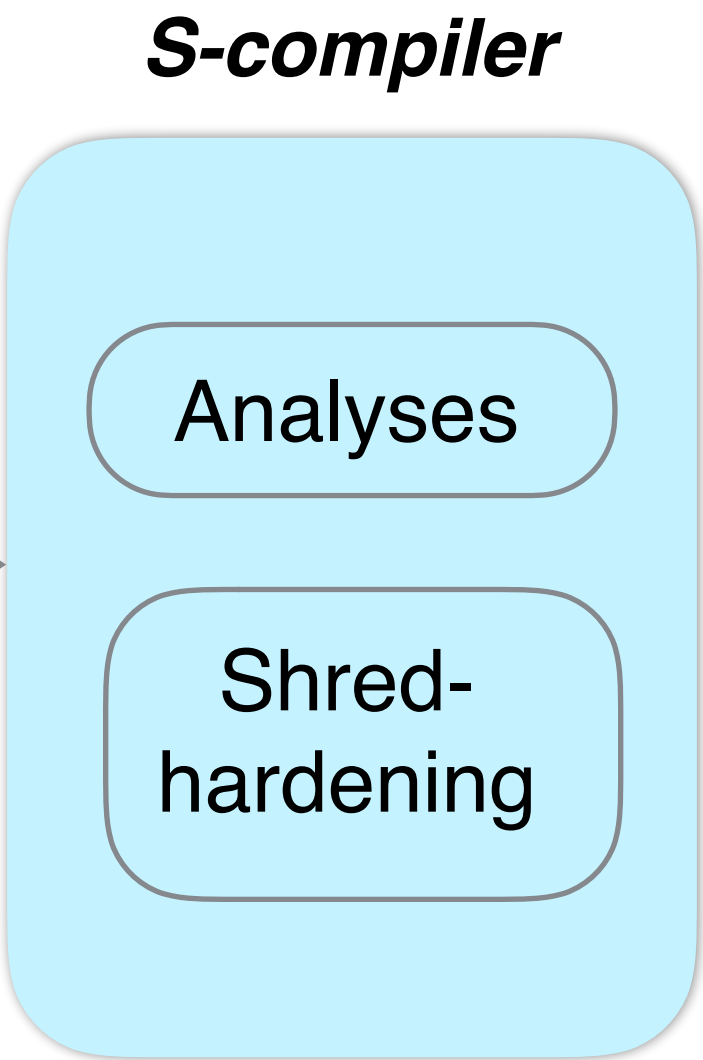
S-compiler

- Shred usage verification
- Associate each shred with its s-pool
- Control flow hardening for in-shred code
- Data flow checking to prevent direct-propagation

```
src.c
...
int enc(x) {
...
  shred_enter(p1);

  //encryption logic

  shred_exit();
...
}
```

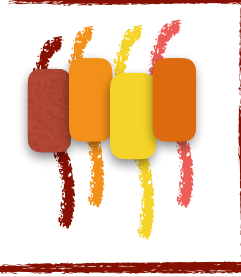


Development and build



Shreds: Fine-grained Execution Units with Private Memory
Yaohui Chen Sebasujeen Reymondjohnson Zhichuang Sun Long Lu
{yaohchen, sreymondjohn, zhisun, long}@cs.stonybrook.edu
Department of Computer Science
Stony Brook University

Abstract—Once attackers have injected code into a victim program's address space, or found a memory disclosure vulnerability, sensitive data and code inside that address space are subject to hijacking or manipulation. Unfortunately, this broad type of attack is hard to prevent, even if software developers wish to cooperate at process level and previously proposed protection methods are not practical for wide adoption. We propose a set of OS-backed programming primitives that enable fine-grained protection of sensitive memory regions (hence the name). A shred can be viewed as a private memory pool, which is a unit of memory that is granted to a process and remains an open issue, which has been increasingly exploited by attackers. To address this open issue, some recent work proposed thread-level memory isolation [3], which allows developers to limit the sharing of a thread's memory space with other threads in the same process. However, this line of works faces three major limitations. First, thread-level memory isolation is still too coarse to stop in-process abuse because exploitable or malicious code often run in the same thread as the legitimate code that needs to access sensitive memory content. Second, adopting these solutions requires significant efforts from developers. Separating application components into different threads (i.e., scheduling units) demands major design changes, as opposed to applying patches, to deal with the added concurrency. Third, threads with private memory tend to have higher overhead than normal threads due to the frequent context switches, TLB flushes, or nested page table switches. We aim to tackle these issues by introducing a practical and effective system for user-level memory isolation. We grant an arbitrarily sized memory pool to a process, which is called a shred. The pool is used to store the code and data of the process, and the pool is managed by the OS. The pool is managed by the OS, and the pool is managed by the OS.



- Hardware spec: Raspberry Pi 2 Model B (Quad-core Cortex-A7 Processor with 1GB RAM)

Softwares

- **Curl**
- **Minizip**
- **OpenSSH**
- **OpenSSL**
- **Lighttpd**

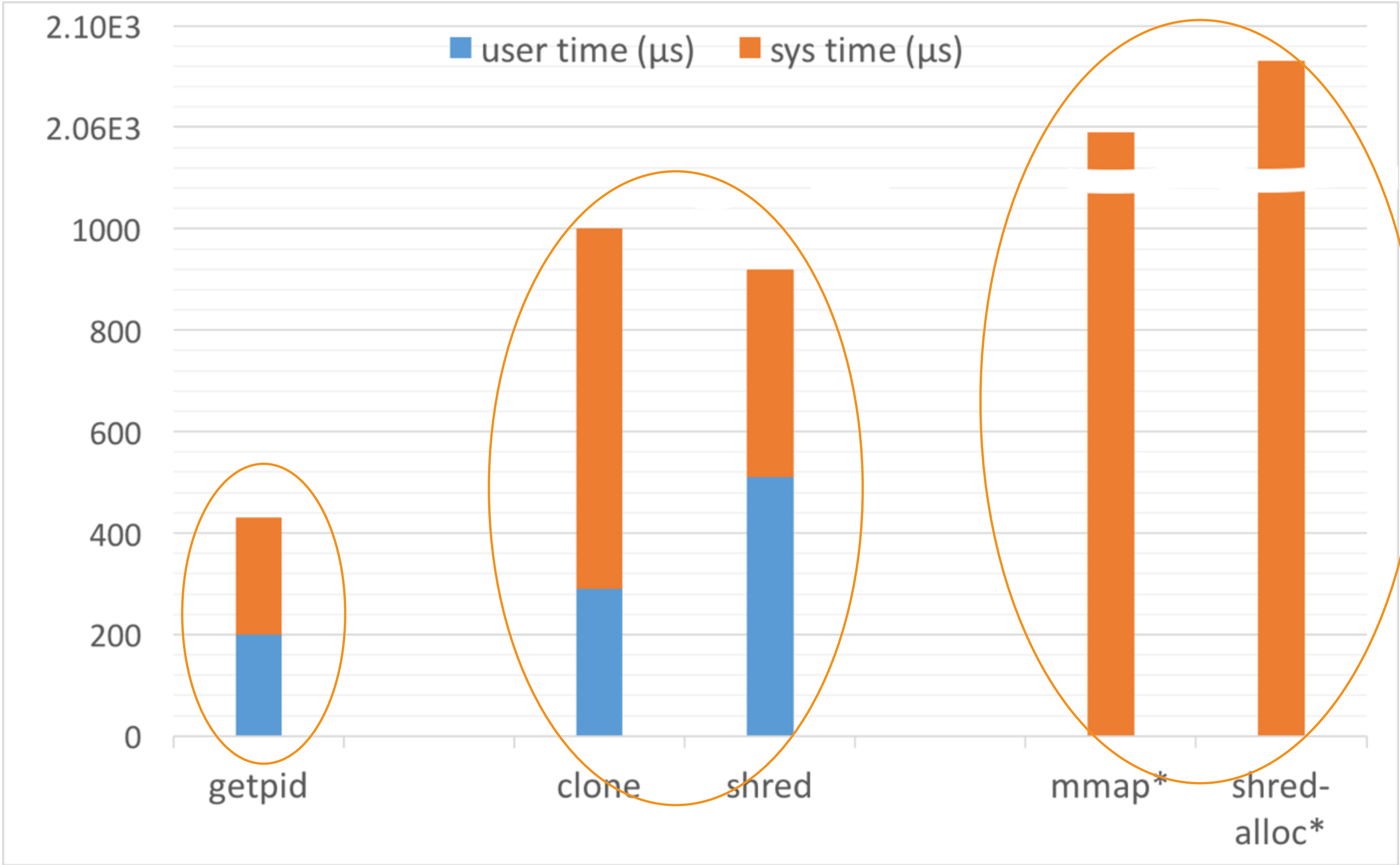
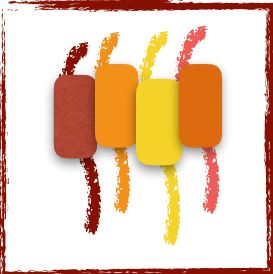
Easy adoption

- Avg. **21** SLOC change
- Avg. **32 min** adoption time

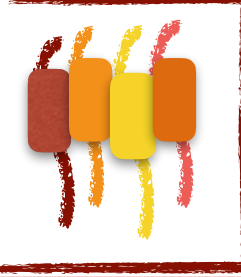
Low overhead

- Avg. **4.67%** slowdown
- Avg. **7.26%** RSS(resident set size) overhead

Evaluation cont.



Conclusion



- Goal— To help developers protect sensitive code/data from **in-process abuse**
- To achieve the goal we propose **shreds** with **private** memory
 - Fine-grained: Flexibly scoped segments of thread executions
 - Efficient and compatible : MMU based domain check
 - ▶ No multiple page tables
 - ▶ No nested paging
 - ▶ No heavy instrumentations
 - ▶ No hardware modifications
 - Robust:
 - ▶ Prevent out-shred attacks + intra-shred vulnerabilities