Shreds: Fine-grained Execution Units with Private Memory

Yaohui Chen Sebassujeen Reymondjohnson Zhichuang Sun Long Lu

Department of Computer Science
Stony Brook University
{yaohchen, sreymondjohn, zhisun, long}@cs.stonybrook.edu

***Abstract*—Once attackers have injected code into a victim program's address space, or found a memory disclosure vulnerability, all sensitive data and code inside that address space are subject to thefts or manipulation. Unfortunately, this broad type of attack is hard to prevent, even if software developers wish to cooperate, mostly because the conventional memory protection only works at process level and previously proposed in-process memory isolation methods are not practical for wide adoption.**

We propose shreds, a set of OS-backed programming primitives that addresses developers' currently unmet needs for fine-grained, convenient, and efficient protection of sensitive memory content against in-process adversaries. A shred can be viewed as a flexibly defined segment of a thread execution (hence the name). Each shred is associated with a protected memory pool, which is accessible only to code running in the shred. Unlike previous works, shreds offer in-process private memory without relying on separate page tables, nested paging, or even modified hardware. Plus, shreds provide the essential data flow and control flow guarantees for running sensitive code. We have built the compiler toolchain and the OS module that together enable shreds on Linux. We demonstrated the usage of shreds and evaluated their performance using 5 non-trivial open source software, including OpenSSH and Lighttpd. The results show that shreds are fairly easy to use and incur low runtime overhead (4.67%).

I. Introduction

Many attacks on software aim at accessing sensitive content in victim programs' memory, including *secret data* (*e.g.,* crypto keys and user passwords) and *critical code* (*e.g.,* private APIs and privileged functions). To achieve the goal, such attacks succeed as soon as they manage to execute code in target programs' process context, which is usually achieved via remote exploitations or malicious libraries. For instance, the HeartBleed attack on OpenSSL-equipped software reads private keys by exploiting a memory disclosure vulnerability [1]; the malicious libraries found in mobile apps covertly invoke private framework APIs to steal user data [2]. Obviously, this whole class of attacks cannot succeed if target programs are able to protect its sensitive data and code against hostile code running in the same process, such as injected shellcode and malicious libraries. We generally refer to this class of attacks as *in-process abuse*.

Developers are virtually helpless when it comes to preventing in-process abuse in their programs, due to a lack of support from underlying operating systems (OS): the memory isolation mechanisms provided by modern OS operate merely at the process level and cannot be used to establish security boundaries inside a process. As a result, protecting sensitive memory content against malicious code inside the same process remains an open issue, which has been increasingly exploited by attackers.

To address this open issue, some recent work proposed the 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 regional code patches, to deal with the added concurrency. Third, threads with private memory tend to incur much higher overhead than normal threads due to the additional page table switches, TLB flushes, or nested page table management upon context switches. We aim to tackle these challenges by proposing a practical and effective system to realize in-process private memory.

In this paper, we present a new execution unit for user-space code, namely *shred*, which represents an arbitrarily sized segment of a thread (hence the name) and is granted exclusive access to a protected memory pool, namely *shred-private pool* (or *s-pool*). Figure 1 depicts shreds in relation to the conventional execution units. Upon its creation, a shred is associated an s-pool, which can be shared among multiple shreds. Shreds address developers' currently unmet needs for fine-grained, convenient, and efficient protection of sensitive memory content against in-process adversaries. To prevent sensitive content in memory from in-process abuse, a developer includes into a shred the code that needs access to the sensitive content and stores the content in the shred's s-pool. For instance, an encryption function can run in a shred with the secret keys stored in the s-pool; a routine allowed to call a private API can run in a shred whose s-pool contains the API code.

We design shreds under a realistically adversarial threat model. We assume attackers may have successfully compromised a victim program, via either remote exploitation or malicious local libraries. Attackers' goal is to access the sensitive content, including both data and code, in the victim program's virtual memory space. Further, we expect unknown vulnerabilities to exist inside shreds (*e.g.,* control flow hijacks and data leaks are possible). On the other hand, we assume a clean OS, which serves as the TCB for shreds. The assumption

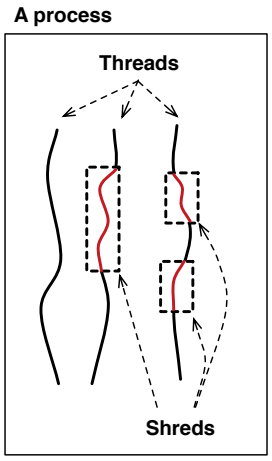

Fig. 1: Shreds, threads, and a process

is reasonable because the attacks that shreds aim to prevent, in-process abuse, would become unnecessary had attackers already subverted the OS. In fact, we advocate that, future OS should support shreds, or more generally, enable private memory for execution units of smaller granularities than the scheduling units.

We realize the concept of shreds by designing and building: (*i*) a set of easy-to-use APIs for developers to use shreds and s-pools; (*ii*) a compilation toolchain, called *S-compiler*, automatically verifying, instrumenting, and building programs using shreds; (*iii*) a loadable kernel extension, called *S-driver*, enabling the support and protection of shreds on commodity OS. Figure 2 shows an overview of the entire system and the workflow. A developer creates a shred and associates it with a selected s-pool by calling the `shred_enter` API and supplying the s-pool descriptor as the argument. Code inside a shred may access content in the associated s-pool as if it were a normal region in the virtual memory space. But the s-pool is inaccessible outside of the associated shred(s). S-pools are managed and protected by S-driver in a way oblivious to developers or applications. With the help of use-define chain analysis on labeled sensitive variables, shreds can also be created automatically at compile time.

As shown in Figure 2, while compiling programs that use shreds, S-compiler automatically verifies the safe usage of shreds and instruments in-shred code with inline checks. The verification and instrumentation regulate sensitive data propagation and control flows inside shreds so that unknown vulnerabilities inside shreds cannot lead to secret leaks or shred hijacking. During runtime, S-driver serves as the manager for s-pools and the security monitor for executing shreds. It creates and resizes s-pools on demand. It enables a per-CPU locking mechanism on s-pools and ensures that only authorized shreds may access s-pools despite concurrent threads.

S-driver leverages an under-exploited CPU feature, namely ARM *memory domains* [4], to efficiently realize s-pools and enforce shred-based access control. Unlike the previously proposed thread-level memory isolations, our approach neither requires separate page tables nor causes additional page table switches or full TLB flushes. Our approach also avoids the need for a hypervisor or additional levels of address translates (*e.g.,* nested paging). Although our reference design and implementation of s-pools are based on ARM CPUs, they are compatible with future x86 architectures, which will be equipped with a feature similar to memory domain [5], [6].

Shreds have the following key advantages:

- Shreds are *fine-grained*. Depending on developers' needs, the scope of a shred can range from a few lines of code to an entire thread, enabling private memory for execution units of various sizes.
- Shreds are *convenient to use*. Unlike splitting programs to processes or threads, creating shreds does not require major software redesigns to deal with concurrency, synchronization, memory sharing, *etc.*
- Shreds are *efficient*. They introduce neither additional process or thread switches nor scheduling constraints.

We implement S-compiler based on LLVM [7] and S-driver as a kernel module for Linux. We evaluate shreds' compatibility and the ease of adoption by manually retrofitting shreds into 5 non-trivial open source software, including OpenSSL and Lighttpd. We show that developers can easily adopt shreds in their code without design-level changes or sacrifice of functionality. Our evaluation shows that shreds incurs an average end-to-end overhead of 4.67%. We also conduct security analysis on shreds, confirming that possible attacks allowed in our thread model are prevented. Overall, our results indicate that shreds can be easily adopted in real software for fine-grained protection of sensitive memory content while incurring very low overhead.

In summary, our work makes the following contributions:

- We identify and address the open challenges facing the previously proposed in-process memory protection, which suffer from rigid granularity, difficult adoption, and high overhead.
- We propose a new OS primitive, namely shred, which represents an arbitrary fragment of a user-space thread execution. Code running inside a shred has access to the shred's private memory pool where sensitive data and code can be stored.
- We build and evaluate the compiler toolchain and the OS module for realizing shreds and assisting the use of shreds. We show that shreds are fine-grained, easily adoptable, and efficient.
- We demonstrate the use cases of shreds in 5 nontrivial open source software, including OpenSSL and lighthttpd. We also evaluate the performance of shreds in these software.

The rest of the paper is organized as follows: in § II we lay out the background of in-process memory abuse and the exist-

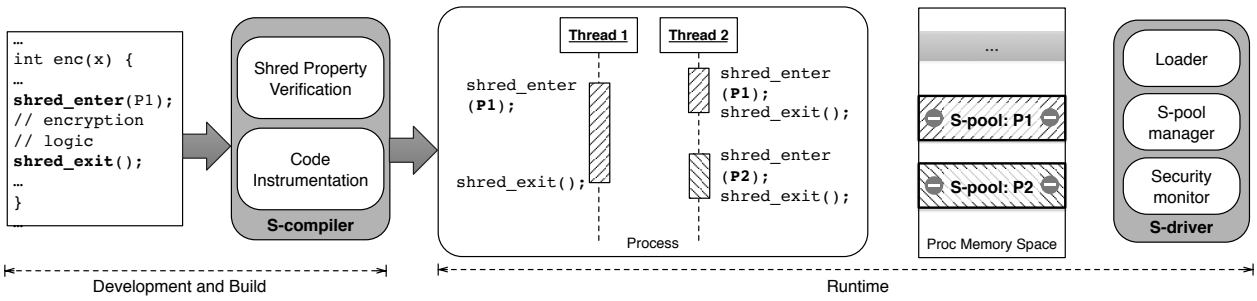

Fig. 2: Developers create shreds in their programs via the intuitive APIs and build the programs using S-compiler, which automatically verifies and instruments the executables (left); during runtime (right), S-driver handles shred entrances and exits on each CPU/thread while efficiently granting or revoking each CPU's access to the s-pools.

ing solutions; we also derive the fundamental requirements for an ideal solution. In § III we discuss the design of our proposed system, comprising s-compiler and s-driver. We also explain in details the security properties of shreds and s-pools and how these properties are enforced. The implementation details of s-compiler and s-driver are outlined in § IV respectively. We then examine the security of our system and evaluate its performance in § V. We contrast the related work in § VI and conclude the paper in § VII.

II. Background

A. In-process Abuse of Memory Content

Ranging from data theft to privilege escalation, a variety of user-space attacks, launched either remotely or locally, can succeed because they can freely access (or abuse) target programs' memory content once they have penetrated into targets' process context.

Stealing secret data: Via either injected shellcode or planted malicious libraries, attackers may obtain unchecked access to victim processes' memory, and in turn, exfiltrate cleartext secrets. Moreover, data thefts may happen without code injection if memory disclosure vulnerabilities exist in target programs. Such attacks are often seen in both network-facing programs and mobile apps. The former, including web servers and browsers, is often prone to remote secret thefts (*e.g.,* the Heartbleed bug) whereas the latter tends to contain many untrusted libraries (*e.g.,* advertisement and analytics).

Executing private code: Malicious code injected into a victim process can freely execute the code loaded in the memory space, including private and privileged functions that are only intended for a few pieces of code in the same process. For instance, `dlopen` is a private API on iOS and is not allowed to be directly called by apps. However, since iOS runtime always loads `dlopen` and other private APIs inside every app process, malicious apps can stealthily invoke these system-reserved APIs to escalate privileges or bypass security checks.

Mining memory: In-process malicious code may silently scan the entire user-space memory for private data, such as passwords, credit card numbers, and sensitive logs. For instance, memory-scrapping code are found inside almost all the recent attacks on POS (Point of Sale) machines. Memory scrapers are also used for illegally identifying and tracking users.

Despite their different goals, the aforementioned attacks all hinge on the same capability to succeed—access to data or code that belong to other components (*e.g.,* functions, modules, libraries, *etc.*) executing in the same process context. We refer to this essential attacking capability as *in-process memory abuse*.

In light of the surging issue of in-process memory abuse and the lack of effective defense, it becomes obvious that new memory protection mechanisms are much needed to protect sensitive data and code against malicious code that manage to run in a same process context.

B. In-process Private Memory: Requirements and Intuitions

A direct solution to in-process abuse is to enable private memory in individual processes and allow application developers to protect their sensitive code and data against malicious code that manages to enter the same processes. We identify four basic requirements for such a solution to be useful, practical, and secure:

R1 - Flexible granularity: A solution needs to recognize accessing entities at various granularities, from compilation modules, to functions, and even to sub-routines. This is necessary for developers to define the minimum entities that should be permitted to access the private memory (*i.e.,* minimizing the exposure of protected memory). It also narrows the scope of code that developers need to change when adopting the solution.

R2 - Easy adoption and deployment: A solution must cover not only newly developed software but also legacy ones. It needs to be easily adoptable by developers and practically deployable in real-world. This requirement boils down to simple APIs, minimal required code changes, and realistic deployment restrictions.

R3 - Robustness against attacks: Accessing entities, even when minimally defined, may still contain vulnerable or faulty code, which can lead to data leakage from private memory or code injection into the accessing entities. A solution must consider and prevent such vulnerabilities or attacks.

R4 - Low overhead: Memory content that need protection may be frequently used throughout a program's execution. Therefore, a solution providing in-process private memory must not incur high overhead. It should be independent from, and more efficient than, the conventional paging-based memory protection.

C. Existing Solutions

Several existing security methods and mechanisms can be used to mitigate in-process memory abuse. However, they fail to meet some or all of the above requirements, and therefore, cannot fully or effectively prevent in-process memory abuse.

Employing privilege separation [8]–[10], developers can protect the sensitive memory content by executing code of different levels of privileges in isolated processes. If OS is not trusted, stronger isolation can be achieved with hardware support, such as virtualization [11] and SGX [12]. However, needless to say virtual machines or enclaves, processes are often too coarse and too restrictive for protecting sensitive code and data that are tightly integrated and frequently used in applications. Software fault isolation (SFI) [13] and similar techniques can confine untrusted code inside a memory region and prevent it from adversely impacting the rest of an application. However, it requires potentially offensive code to be instrumented and verified during compilation, which is unrealistic for stealthily injected malicious code that performs in-process abuse. Some recent works [3] have enabled thread-level memory isolation. Although taking an important step towards mitigating in-process memory abuse, the approach still relies on the scheduling units (*e.g.,* threads and processes) for memory isolation, which makes the solutions coarse-grained, cumbersome to adopt, and inefficient. More detailed discussion about the related works is in § VI.

Motivated by the need for in-process private memory that meets R1-R4, we design and build the shred system.

III. System Design

A. Overview

We introduce a new OS primitive, namely *shred*, for securely executing certain (sensitive) pieces of application code against in-process attacks. Shreds are thread segments of various sizes (Figure 1), which are defined by application developers. Code running inside a shred can store and access secrets in an assigned memory pool (s-pool), which is inaccessible to the rest of the thread or other threads in the same process, despite that they all share the same virtual memory space. By running sensitive code pieces in individual shreds and storing secrets in associated s-pools, developers prevent malicious or erroneous code running in the same thread or process from retrieving the secrets, and in turn, defend against in-process abuse attacks.

Shreds' security is guaranteed by three properties:

- **P1 - Exclusive access to s-pool:** An s-pool is solely accessible to its associated shreds. Other shreds or threads, even when running concurrently with the associated shreds, cannot access the s-pool.
- **P2 - Non-leaky entry and exit:** Data loaded into s-pools cannot have copies elsewhere in memory or be exported without sanitization.
- **P3 - Untampered execution:** Shred execution cannot be altered or diverted outside of the shred.

$P1$ enables the very protection of a shred's sensitive memory against other unrelated shreds or out-shred code that run in the same address space. $P2$ avoids secret leaks when data are being loaded into or exported out of s-pools (*e.g.,* ensuring that no secret is buffered in unprotected memory as a result of standard I/O). $P3$ prevents in-process malicious code from manipulating shreds' control flow. Such manipulation can cause, for instance, ROP that forces a shred to execute out-shred code and expose its s-pool.

We design and implement a system that enables shreds and the aforementioned properties for Linux/ARM platforms. Our system consists of a compilation toolchain (S-compiler) and a dynamic loadable kernel extension (S-driver). Developers can adopt shreds in their programs using a set of simple APIs: two APIs for entering and exiting a shred; two APIs for allocating and freeing memory in an s-pool. S-compiler is needed to build programs that contain shreds. S-compiler performs the code analysis and instrumentation that are necessary to ensure $P2$ and $P3$. During runtime, S-driver handles shred creations and terminations. It manages and protects s-pools in accordance to $P1$. Our design makes a novel use of memory domains, an under-exploited feature in ARM CPUs, to efficiently protect s-pools and shred executions.

The rest of the section explains the detailed designs of shred APIs, S-compiler, and S-driver. It then examines the designs against the requirements ($R1$-$R4$).

B. Shred APIs and Usages

Application developers use shreds and s-pools via the following intuitive APIs:

```
err_t shred_enter(int pool_desc);
err_t shred_exit();
void * spool_alloc(size_t size);
void spool_free(void *ptr);
```

These APIs internally make requests to S-driver via `ioctl` for managing shreds and s-pools. To explain the API usage, we use the lightweight open-source web server, Lighttpd, as an example, where we employ shreds to protect the HTTP authentication password in Lighttpd's virtual memory. By wrapping the code that receives and checks the password in two shreds and storing the password in an s-pool, the

modified Lighttpd prevents out-shred code, including third-party and injected code, from accessing the password in memory. Listings 1-3 show the code snippets that contain the modifications (lines marked with "+").

A successful call to **shred_enter** starts a shred execution on the current thread. It also causes a switch to a secure execution stack allocated in s-pool, which prevents potential secret leaks via local variables after the shred exits. The thread then is given exclusive access to the associated s-pool, which is specified by the developer using the *pool_desc* parameter of **shred_enter**. Our design allows developers to associate an s-pool with multiple shreds by using the same descriptor at shred creations (*e.g.,* an encryption shred and a decryption shred may need to share the same s-pool storing keys). The two shreds in Lighttpd, created on Line 9 in Listing 1 and Line 3 in Listing 3, share the same s-pool. However, as a security restriction, shreds in different compilation units cannot share s-pools. Therefore, even if shreds from different origins happen to use the same descriptor value, their s-pools are kept separate.

The **shred_exit** API stops the calling shred, revokes the current thread's access to the s-pool, and recovers the original execution stack. It is called immediately after a self-contained operation or computation on the s-pool finishes, as shown on Line 22 in Listing 1 and Line 8 in Listing 3. The shred enter and exit APIs must be used in pairs without nesting. To facilitate verification, an enter-exit pair must be called inside a same function. In principle, a shred should contain a minimum body of code that corresponds to a single undividable task requiring access to an s-pool. In the example, since Lighttpd separates the parsing and processing of HTTP requests, we naturally used two small shreds, rather than one big shred, to respectively read the password from network and checks if the hash value of the password matches with the local hash.

To allocate memory from its associated s-pool, in-shred code calls **spool_alloc**, in a same way as using libc's `malloc`. Similar to regular heap-backed memory regions, buffers allocated in s-pools are persistent and do not change as code execution enters or exits shreds. They are erased and reclaimed by S-driver when in-shred code calls **spool_free**. In the Lighttpd example, an s-pool named `AUTH_PASSWD_POOL` is used for storing the password that the server receives via HTTP authentication requests. The password enters the s-pool immediately after being read from the network stream and stays there till being erased at the end of its lifecycle.

```
1  int http_request_parse(server *srv,
2      connection *con) {
3  ...
4    /* inside the request parsing loop */
5     char *cur;  /* current parsing offset */
6  +  char auth_str[] = "Authorization";
7  +  int auth_str_len = strlen(auth_str);
8  +  if (strncmp(cur, auth_str, auth_str_len)==0){
9  +   shred_enter(AUTH_PASSWD_POOL);
10 +   /* object holding passwd in spool */
11 +   data_string *ds = s_ds_init();
12 +   int pw_len = get_passwd_length(cur);
13 +   cur += auth_str_len + 1;
14 +   buffer_copy_string_len(ds->key, auth_str,
15 +     auth_str_len);
16 +   buffer_copy_string_len(ds->value, cur, pw_len)
       ;
17 +   /* add ds to header pointer array */
18 +   array_insert_unique(parsed_headers, ds);
19 +   /* only related shreds can deref ds */
20 +   /* wipe out passwd from input stream */
21 +   memset(cur, 0, pw_len);
22 +   cur += pw_len;
23 +   shred_exit();
24 +  }
25 ...
   }
```

Listing 1: `lighttpd/src/request.c` – The HTTP request parser specially handles the AUTH request inside a shred: it allocates a `data_string` object in the s-pool (Line 11), copies the input password from the network stream to the object (Line 12-15), saves the object pointer to the array of parsed headers (Line 17), and finally erases the password from the input buffer before exiting the shred.

```
1  /* called inside a shred */
2  data_string *s_ds_init(void) {
3     data_string *ds;
4  +  ds = spool_alloc(sizeof(*ds));
5  +  ds->key = spool_alloc(sizeof(buffer));
6  +  ds->value = spool_alloc(sizeof(buffer));
7  ...
8     return ds;
9  }
10
11 /* called inside a shred */
12 void s_ds_free(data_string *ds) {
13 ...
14 +  spool_free(ds->key);
15 +  spool_free(ds->value);
16 +  spool_free(ds);
17    return;
18 }
```

Listing 2: `lighttpd/src/data_string.c` – We added s-pool support to the `data_string` type in Lighttpd, which allows the HTTP parser to save the AUTH password, among other things, in s-pools and erase them when needed.

```
1  ...
2  /* inside HTTP auth module */
3  +  shred_enter(AUTH_PASSWD_POOL);
4     /* ds points passwd obj in spool */
5     http_authorization = ds->value->ptr;
6     ... // hash passwd and compare with local copy
7  +  s_ds_free(ds);
8  +  shred_exit();
9  ...
```

Listing 3: `lighttpd/src/mod_auth.c` – When the authentication module receives the parsed headers, it enters a shred, associated to the same s-pool as the parser shred. It retrieves the password by dereferencing `ds`, as if the password resided in a regular memory region (Line 5)

Code included in a shred need to follow two rules. First, it cannot copy data from an s-pool to unprotected memory

without applying any transformation (*e.g.,* encryption). This rule prevents unexpected secret leaks from s-pools and is needed for achieving $P2$. Second, in-shred code can only use libraries built using S-compiler. This rule allows all code inside shreds to be checked and instrumented for $P3$. Although seemingly restrictive, the second rule is not impractical: the commonly used libraries, such as libc and libm, can be pre-compiled and installed along with S-driver as part of system deployment; the uncommon libraries required in shreds for processing sensitive data are usually in-house developed or open source, and therefore, can be recompiled by developers. Both rules are enforced by S-compiler.

C. S-compiler: automatic toolchain for shred verification and instrumentation

Developers use S-compiler to build programs that use shreds. In addition to regular compilation, S-compiler performs a series of analysis and instrumentation to verify programs' use of shreds and prepare the executables so that S-driver can enforce the security properties ($P1$-$P2$) during runtime. Unlike general-purpose program analysis, S-compiler's analysis is mostly scoped within the code involved in shred executions, and therefore, can afford to favor accuracy over scalability. Prior to the analysis and transformation, S-compiler translates an input program into an intermediate representation (IR) in the single static assignment (SSA) form.

Checking shred usage: To verify that all shreds in the program are properly closed, S-compiler first identifies all the shred creations sites(*i.e.,* calls to **`shred_enter`**), uses them as analysis entry points, and constructs a context-sensitive control flow graph for each shred. S-compiler then performs a code path exploration on each graph in search for any unclosed shred (or unpaired use of **`shred_enter`** and **`shred_exit`**), which developers are asked to fix. This check is sound because it is not inter-procedural (*i.e.,* a pair of shred enter and exit APIs must be called inside a same function) and it conservatively models indirect jumps.

To prevent potential secret leaks, S-compiler performs an inter-procedural data-flow analysis in each shred. Potential leaks happen when secrets are either loaded from, or stored to, unprotected memory. The data-flow analysis checks for both cases. First, it ensures that data stored in s-pools do not pre-exist in regular memory (*i.e.,* such data must be directly loaded into s-pools from input channels, such as `stdin` or file system. Second, the analysis checks for any unsanitized data propagation from an s-pool object to a regular heap destination. Thanks to the explicit memory allocations and aliasing in s-pool, the data-flow analysis needs neither manually defined sources or sinks nor heuristic point-to analysis. In addition, this analysis strikes a balance between security and usability: it captures the common forms of secret leaks (*e.g.,* those resulted from bugs) while permitting intentional data exports (*e.g.,* saving encrypted secrets).

Buffered I/O, when used for loading or storing s-pool data, may implicitly leak the data to pre-allocated buffers outside of s-pools, which data-flow analysis can hardly detect. Therefore, S-compiler replaces any buffered I/O (*e.g.,*`fopen`) with direct I/O (*e.g.,*`open`) in shreds.

Hardening in-shred control flows: We adopt a customized form control-flow integrity (CFI) to ensure that in-process malicious code cannot hijack any shred execution. To that end, S-compiler hardens in-shred code during compilation. Based on the control flow graphs constructed in the previous step, S-compiler identifies all dynamic control flow transfers, including indirect jumps and calls as well as returns, inside each shred. It then instruments these control flow transfers so that they only target basic block entrances within containing shreds. This slightly coarse-grained CFI does not incur high overhead as the fine-grained CFI and at the same time is sufficiently secure for our use. It prevents shred execution from being diverted to out-shred code. Furthermore, since shreds are usually small in code size (*i.e.,* few ROP gadgets) and our CFI only allows basic block-aligned control transfers, the chance of in-shred ROP is practically negligible.

The control flow hardening only applies to in-shred code. If a function is called both inside and outside of a shred, S-compiler duplicates the function and instruments the duplicate for in-shred use while keeping the original function unchanged for out-shred use. S-compiler creates new symbols for such duplicates and replaces the in-shred call targets with the new symbols. As a result, a function can be used inside shreds and instrumented without affecting out-shred invocations. Using function duplicates also allows S-compiler to arrange the code reachable in a shred in adjacent memory pages, which facilitates the enforcement of control flow instrumentations and improves code cache locality.

Binding shreds and s-pools: Developers define a constant integer as the pool descriptor for each s-pool they need. To associate an s-pool with a shred, they use the constant descriptor as the *`pool_desc`* parameter when calling **`shred_enter`**. This simple way of creating the association is intuitive and allows explicit sharing of an s-pool among multiple shreds. However, if not protected, it may be abused by in-process malicious code (*e.g.,* creating a shred with an association to an arbitrary s-pool). S-compiler prevents such abuse by statically binding shreds and their s-pools. It first infers the pool-shred association by performing a constant folding on the *`pool_desc`* used in each **`shred_enter`** invocation. It then records the associations in a special section (`.shred`) in the resulting executable, to which S-driver will refer during runtime when deciding if a shred (identified by its relative offset in memory) indeed has access to a requested s-pool. Thanks to the static binding, dynamically forged pool-shred association is prevented, so is s-pool sharing across different compilation units.

Similar to previous works employing code instrumentation and inline reference monitoring, we assume that attackers cannot rewrite executables produced by S-compiler. Further, S-driver write-protects the instrumented code and their critical

runtime data structures in memory. More details about the security and robustness of system are discussed in § V-A.

D. S-driver: OS-level manager for shreds and s-pools

S-driver is a dynamically loadable kernel extension. It can be easily installed on a system as a regular driver. S-driver provides the OS-level support and protection for shreds and s-pools.

ARM memory domains: S-driver leverages a widely available yet rarely used ARM CPU feature, namely the the memory domain mechanism, to realize s-pools or create specially protected memory regions inside a single virtual memory space. At the same time, our design is not specific to ARM and can realize s-pools using a mechanism similar to memory domains in future Intel CPUs [5], [6]. On ARM platforms, domains are a primary yet lesser known memory access control mechanism, independent of the widely used paging-based access control. A memory domain represents a collection of virtual memory regions. By setting a 4-bit flag in a Page Directory Entry (PDE), OS assigns the memory region described by the PDE to one of the 16 (2^4) domains supported by the CPU. Since each PDE has its own domain flag, the regions constituting a domain do not have to be adjacent. Upon each memory access, the hardware Memory Management Unit (MMU) determines the domain to which the requested memory address belongs and then decides if the access should be allowed based on the current access level for that domain. The access level for each domain is recorded in the per-core Domain Access Control Registers (DACR) [14], and therefore, can be individually configured for each CPU core.

Creation and management of s-pools: Although memory domains are ideal building blocks for s-pools thanks to their efficient hardware-enforced access control, memory domains are not originally designed for this purpose and cannot directly enable s-pools due to two limitations. First, only a total of 16 memory domains are available. If intuitively using one domain for creating one s-pool, the limited domains will soon run out as the number of s-pools used in a program increases. Second, the access control on memory domains is very basic and does not concern the subject of an access (*i.e.,* who initiates the access). However, access control for s-pools must recognize subjects at the granularity of shreds. S-driver overcomes both limitations of memory domains by multiplexing the limited domains and introducing shred identities into the access control logic.

S-driver uses the limited domains to support as many s-pools as an application may need. Rather than permanently assigning an s-pool to a domain, S-driver uses domains as temporary and rotating security identities for s-pools in an on-demand fashion. Specifically, it uses a total of $k = Min(N_{dom} - 1, N_{cpu})$ domains, where N_{dom} is the number of available domains and N_{cpu} is the number of CPU (or cores) on a system. The first k domains are reserved for the first k CPUs.

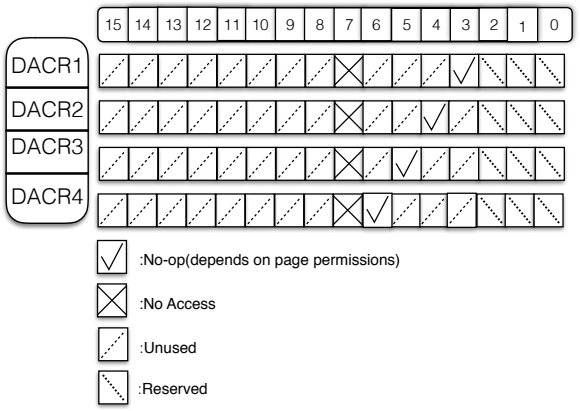

Fig. 3: The DACR setup for a quad-core system, where $k = 4$. The first 3 domains ($Dom_0 - Dom_2$) are reserved by Linux. Each core has a designated domain ($Dom_3 - Dom_6$) that it may access when executing a shred. No CPU can access Dom_7.

S-driver sets the per-CPU DACR in a way such that, Dom_i is only accessible to shreds running on CPU_i, for the first k CPUs; Dom_{k+1} is inaccessible to any CPU in user mode. Figure 3 shows an example DACR setup.

S-driver uses the k CPUs and the $k + 1$ domains for executing shreds and protecting s-pools. When a shred starts or resumes its execution on CPU_i, S-driver assigns its associated s-pool to Dom_i, and therefore, the shred can freely access its s-pool while other concurrent threads, if any, cannot. When the shred terminates or is preempted, S-driver assigns its s-pool to Dom_{k+1}, which prevents any access to the pool from that moment on. As a result, S-driver allows or denies access to s-pools on a per-CPU basis, depending on if an associated shred occupies the CPU. Even if any malicious code manages to run concurrently alongside the shred inside the same process on another CPU, it cannot access the shred's s-pool without triggering domain faults. Thus, $P1$ is achieved.

It is reasonably efficient to switch s-pools to different domains upon shred entries and exits are. These operations do not involve heavy page table switches as process- or VM-based solutions do. They only require a shallow walk through of the first level page table and updates to the PDEs pointing to the s-pools in question. Besides, they do not trigger full TLB flushes as our design uses the per-address TLB eviction interface (`flush_tlb_page`) and only invalidates the TLB entries related to the updated PDEs. To further reduce the overhead, we invent a technique called *lazy domain adjustment*: when a shred is leaving CPU_i, without adjusting any domain assignment, S-driver quickly changes the DACR to revoke the CPU's access to Dom_i and lets the CPU's execution continue. It does not assign the s-pool used by the previous shred to Dom_{k+1} until a domain fault happens (*i.e.,* another shred coming to the CPU and accessing its s-pool). The lazy domain adjustment avoids unnecessary domain changes and halves the already small overhead in some test cases (see § V).

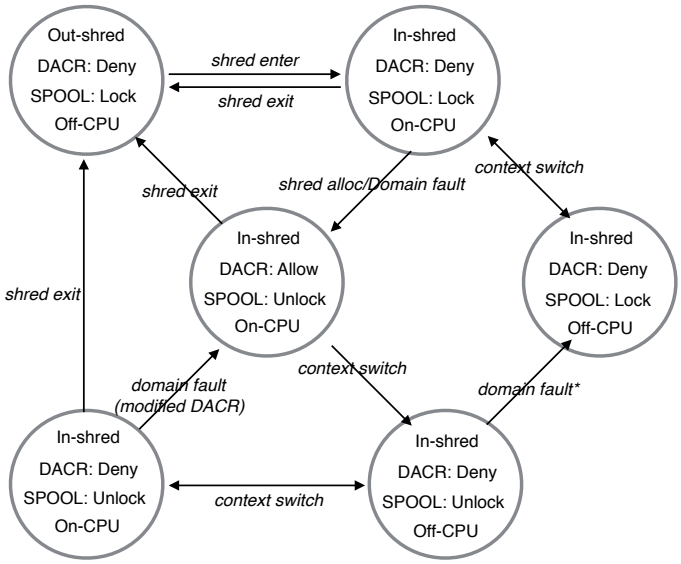

Fig. 4: A shred's transition of states

Figure 4 shows how S-driver orchestrates the transitions of a shred's states in response to the API calls, context switches, and domain faults. Each state is defined by a combination of four properties:

- $Shred$ = {In-shred | Out-shred}: if the shred has started or exited.
- $DACR$ = {Allow | Deny}: if the DACR allows or denies the current CPU to access its domain.
- $SPOOL$ = {Lock | Unlock}: if the associated s-pool is locked or not.
- CPU = {On-CPU | Off-CPU}: if the shred is running on a CPU or not.

The transition starts from the top, left circle, when the shred has not started and its s-pool is locked. After `shred_enter` is called, S-driver starts the shred, but it will not adjust the DACR or the s-pool access till a domain fault or a `spool_alloc` call due to the lazy domain adjustment in effect. When a context switch happens in the middle of the shred execution with unlocked DACR and s-pool, S-driver instantly sets the DACR to Deny but (safely) leaves the s-pool open. Later on, if a domain fault occurs, S-driver locks the previous s-pool because the fault means that the current code running on the CPU is in-shred and is trying to access its s-pool. If a domain fault never occurs till the shred regains the CPU, S-driver does not need to change any domain or s-pool settings, in which case the lazy domain adjustment saves two relatively heavy s-pool locking and unlocking operations.

Secure stacks for shreds: Although S-compiler forbids unsanitized data flows from s-pools to unprotected memory regions, it has to allow in-shred code to copy s-pool data to local variables, which would be located in the regular stack and potentially accessible to in-process malicious code. To prevent secret leaks via stacks, S-driver creates a secure stack for each shred, allocated from its associated s-pool. When code execution enters a shred, S-driver transparently switches the stack without the application's knowledge: it copies the current stack frame to the secure stack and then overwrites the stack pointer. When the shred exits or encounters a signal to be handled outside of the shred, S-driver restores the regular stack. As a result, local variables used by shreds never exist in regular stacks, and therefore cannot leak secrets.

Runtime protection of shreds: In addition to enabling and securing shreds and s-pools, S-driver also protects the inline reference monitor (IRM) that S-compiler plants in shred code. S-driver write-protects the memory pages containing the instrumented code and the associated data in memory. It also pins the pages in s-pools in memory to prevent leaks via memory swap. Given that our threat model assumes the existence of in-process adversaries, S-driver also mediates the system calls that malicious code in user space may use to overwrite the page protection, dump physical memory via `/dev/*mem`, disturb shreds via `ptrace`, or load untrusted kernel modules. For each program using shreds, S-driver starts this mediation before loading the program code, avoiding pre-existing malicious code.

S-driver's system call mediation also mitigates the attacks that steal secret data, not directly from s-pools, but from the I/O media where secret data are loaded or stored. For instance, instead of targeting the private key loaded in an s-pool, an in-process attacker may read the key file on disk. S-driver monitors file-open operations insides shreds. When the first time a file F is accessed by a shred S, S-driver marks F as a shred-private file and only allows shreds that share the same s-pool with S to access F. This restriction is persistent and survives program and system reboots. As a result, an attacker can read F only if she manages to intrude the program during its first run and access F before a shred does. Although not completely preventing such attacks, S-driver makes them very difficult to succeed in reality. For a complete remedy, we envision a new primitive for in-shred code to encrypt and decrypt secret data with a persistent key assigned to each s-pool and automatically managed by S-driver. However, our current prototype does not support this primitive.

It is worth noting that, although the system call mediation can prevent user-space malicious code that tries to break shreds via the system interfaces, it is a more intrusive and less configurable design choice than the well-known access control and capability frameworks, such as SELinux, AppArmor, and Capsicum [15]. However, we leave the integration with those systems as future work because the system call mediation is easy to implement and is sufficient for the prototyping purpose.

E. Satisfied Requirements

We now examine if the design of shreds meets the requirements for in-process private memory (R1-R4 in § II-B).

Shreds remove the historical constraint facing developers

that, memory protection domains can only be created at the granularities of the rigid scheduling units, namely processes. This constrain poses a major challenge to defend in-process abuse. Using shreds, developers can flexibly create execution units of various sizes and individually grant these units access to protected memory pools. As a result, shreds allow for fine-grained protection domains inside processes, and thus, meet R1.

Developers create shreds and use s-pools via the four intuitive APIs. They can easily adopt shreds in either new or legacy applications, without major design changes. When building applications, S-compiler automatically verifies shred usages and hardens the resulting executables. S-driver, the dynamically loadable kernel extension, enables runtime support and protection of shreds without requiring a new or rebuilt OS. The entire system is easy to use and deploy, which meets R2.

Our design assumes that in-shred code would contain vulnerabilities that may lead to secret leaks or control flow hijacks. Our design either precludes such vulnerabilities or prevents them from being exploited. Specifically, S-compiler rejects code containing unsanitized data flows from s-pools to unprotected memory regions. It also inserts checks before statically undecidable memory dereferences whose values may flow to regular memory, preventing potential leaks during runtime. These static and dynamic checks together eliminate outbound propagations of plain data in s-pools, and therefore, enforce the data flow property in R3. S-compiler also instruments indirect control flow transfers in shreds, whose destinations are checked during runtime and assured to be basic block entrances inside containing shreds. These checks enforce the control flow property in R3.

To efficiently enable s-pools, or the in-process private memory regions, our design leverages a widely available yet largely overlooked feature in ARM CPUs, namely memory domains [4] (Intel has prototyped a similar feature for future CPUs [5], [6]. Compared with paging-based memory access control, our domain-based design does not require page table switches, full TLB flushes, or disabling concurrent threads when (un)locking s-pools. Besides, S-driver changes domain assignments and access levels in a lazy fashion, which further reduces the security enforcement overhead. As shown in our evaluation (§ V), using shreds and s-pools only slows down programs by 4.67%, which indicates that R4 is satisfied.

IV. System Implementation

We fully implemented our designs of S-compiler and S-driver. We built S-compiler based on LLVM [7] and its C front-end Clang [16]. We built S-driver with Linux as the reference OS. The implemented system was deployed and evaluated on a quad-core ARM Cortex-A7 computer (Raspberry Pi 2 Model B running Linux 4.1.15). Table I shows the SLoC of the implementation.

		Language	SLOC
S-compiler	*Analysis Pass*	C++	1345
	Instrumentation Pass	C++	275
S-driver		C	1205

TABLE I: The SLoC for S-compiler and S-driver.

S-compiler: The modular and pass-based architecture of LLVM allows us to take advantage of the existing analyzers and easily extends the compilation pipeline. S-compiler adds two new passes to LLVM: the shred analysis pass and the security instrumentation pass. Both operate on LLVM bitcode as the IR.

The analysis pass carries out the checks on the usages and security properties of shreds, as described in § III-C. We did not use LLVM's built-in data flow analysis for those checks due to its overly heuristic point-to analysis and the unnecessarily conservative transfer functions. Instead, we implemented our specialized data flow analysis based on the basic round-robin iterative algorithm, with weak context sensitivity and a straightforward propagation model (i.e., only tracking value-conserving propagators). We also had to extend LLVM's compilation pipeline because it by default only supports intra-module passes while S-compiler needs to perform inter-module analysis. We employed a linker plugin, called the Link-Time Optimization (LTO), to cross link the IR of all compilation modules and feed the linked IR to our analyzers.

The instrumentation pass uses the LLVM IR manipulation interfaces to insert security checks into the analyzed IR that, which are necessary for enforcing the in-shred control flow regulations and preventing dynamic data leaks, as discussed in § III-C.

S-driver: We built S-driver into a Loadable Kernel Module (LKM) for Linux. S-driver creates a virtual device file (`/dev/shreds`) to handle the `ioctl` requests made internally by the shred APIs. It uses 13 out of 16 memory domains to protect s-pools because the recent versions of Linux kernel for ARM already occupies 3 domains (for isolating device, kernel, and user-space memory). S-driver uses the available domains to protect unlimited s-pools and controls each CPU's access to the domains as described in § III-D. Since Linux does not provide callback interfaces for drivers to react to scheduling events, in order to safely handle context switches or signal dispatches in shreds, S-driver dynamically patches the OS scheduler so that, during every context switch, the DACR of the current CPU is reset, which locks the open s-pool, if any. The overhead of this operation is negligible because resetting the DACR only takes a single lightweight instruction. To capture illegal access to s-pools and lazily adjust domain assignments, S-driver registers itself to be the only handler of domain faults and is triggered whenever a domain violation happens. Algorithm 1 shows how S-driver handles a domain fault. Purely implementing S-driver as a LKM allows shreds

Algorithm 1: Domain Fault Handler

```
input : The faulting virtual address fault_addr
result: Recover from the domain fault, or kill the faulting
        thread
/*Identity check*/
s_pool ← FindSpool(fault_addr);
s_owner ← GetOwner(s_pool);
if fault_thread is NOT in shred then
    goto bad_area
if fault_thread is NOT s_owner then
    goto bad_area
/*Recover from domain fault*/
cpu_domain ← GetCPUDomain();
s_pool_domain ← GetSpoolDomain(s_pool);
if s_pool is unlocked then
    if cpu_domain = s_pool_domain then
        /*No need to change domain for s_pool*/
        RestoreDACR();
    else
        AdjustSPool(cpu_domain)
else
    UnlockSPool(cpu_domain)
LockOtherActiveSPools(s_pool);
```

to be introduced into a host without installing a custom-build kernel image.

V. Analysis and Evaluation

A. Security Analysis

We now analyze our system design in terms of its robustness against the evasions or manipulations that attacks may pursue. For this analysis, we assume an attacker has already take control of a victim process, either via remote exploitation or malicious local libraries, which is the most powerful attacker possible under our threat model. With the goal of accessing an s-pool used by the victim program, the attacker may attempt to bypass the security enforcement of shreds in the following ways, all of which are prevented by our design.

First, the attacker may create a shred of her own and try to associate the shred with the target s-pool by specifying a same s-pool descriptor. This attack will fail because: (*i*) if the attacker creates the shred via code in a different executable or compilation unit (*e.g.,* a malicious library), s-driver forbids sharing of s-pools among different compilation units by localizing s-pool descriptors during executable load; (*ii*) if the attacker creates the shred by injecting code in the compromised process, s-driver denies the shred creation because no statically verified information about this shred exits in the executable file.

Second, the attacker may try to hijack a shred execution. She can exploit in-shred code and diverging the the control flow to selected malicious code. In that case, the malicious code would run inside a shred and in turn gain access to the s-pool. However, s-compiler and s-driver together prevent such control flow manipulations via code instrumentation and runtime protection. The instrumentation code checks, among other things, if an indirect control flow transfer is bound by the code coverage of the (vulnerable) shred, as determined by s-compiler (§ III-C).

Third, the attacker may direct the control flow to a legitimate shred entry point in an ROP fashion, hoping to regain the control after the next return instruction or the shred exits. Since s-driver assigns a separate and protected stack for each shred execution, the attacker cannot set up the stack to launch ROP inside the shred. Even if the attacker regains the control immediately after the shred exits, she cannot not learn anything about the data processed in that shred because s-driver resets the stack where before the shred is executed. Moreover, s-driver also prevents other types of manipulation of legitimate shreds, such as hooking the shred APIs and modified the verified code mapped in memory.

Finally, using inline security checks and saving the shred information in executables make an implicit security assumption that, attackers cannot rewrite the executables generated by s-compiler, such as removing the inline security checks or modifying the shred section. We note that this is a common assumption shared by all inline reference monitors. It is feasible in the context of preventing in-process memory abuse: if attackers already control the executable of a program, memory abuse would become unnecessary.

B. Experiments and Evaluation

Our experiments sought to answer the following questions:

- How easy or difficult for developers to adopt shreds in their code?
- How compatible and useful are shreds to real-world programs?
- How do shreds affect the application's and system's performance?

Choice of Applications: We selected 5 popular open source applications to evaluate our prototype system. The applications are shown in Table II, ranging from the small HTTP server, lighttpd, to the complex cryptography library, OpenSSL. The applications were chosen because each of them has at least one piece of sensitive data that is subject to in-process abuse, and therefore, warrants shred's protection. Moreover, the applications represent a good variety of software of different functionalities and codebase sizes.

Adoption Tests: To measure the efforts required to adopt shreds in reality, we hired several CS graduate students to incorporate shreds into the 5 selected applications. They were first given a short tutorial on how to use shreds and s-pools, and then asked to adopt shreds into the application source code. The adoption in these tests did not intend to protect all kinds of sensitive data in the applications, which is unrealistic given that the student participants in the tests are not the original developers of the applications and are unlikely to identify all types of sensitive data. Instead, we asked the participants to protect only one specific type of sensitive data

TABLE II: 5 open source softwares used in evaluation

	Executable Size(byte)	Category	Protected Data Type	Program Size(KLOC)
curl	227071 curl	http client	password	177
minizip	80572 miniunz 97749 minizip	file compression tool	password	7
openssh	2207588 ssh	remote login tool	credential	130
openssl	3093920 libcrypto.so	crypto library	crypto key	526
lighttpd	85135 mod_auth.so	web server	credential	56

in each application (as shown in the Protected Data Type column in Table II). This measurable and realistic task for the participants allowed us to examine how easy or difficult to use shreds correctly and effectively in practice.

After the tests finished, we manually confirmed the correctness and completeness of the code changes. The modified applications compile and run without any issue. As shown in Table III, on average, the participants spent an hour on lighttpd and 15 min on minizip, representing the longest and shortest adoption time measured in the tests. These numbers show that shreds are intuitive even to first-time users. Given that the participants spent most of the time understanding the codebase, we expect that the time needed for adopting shreds will be even shorter when the original application developers perform the tasks. The number of shreds created and the number of SLoC changes do not exhibit direct correlation with the adoption time. The code changes are very small compared with the size of the applications, which indicates that no major design changes are required to apply shreds to existing applications.

TABLE III: Code changed and time spent in adoption tests

Application	Shred numbers	Code change(SLoC)	Adoption time(min)
curl	2	13	30
minizip	4	23	15
openssh	1	8	20
openssl	3	34	35
lighttpd	2	27	60

Compilation Tests: To test the performance and compatibility of our offline analysis and compilation methods, we instrumented S-compiler in order to measure the overhead and log potential errors, if any, while building the 5 software packages that use shreds. Figure 5 shows the time and space overhead introduced by S-compiler, relative to the performance of a vanilla LLVM Clang compiling the unchanged applications. On average, S-compiler delays the building process by 24.58% and results in a 7.37% increase in executable sizes. The seemingly significant delays in compilation are in fact on par with static analysis and program instrumentation tools of similar scale. They are generally tolerable because compilations take place offline in background and are usually not considered to be time-critical. The executable file size increases are mainly resulted from the in-shred instrumentation and are below 2% except for the outliers. We encountered no error when building these applications using S-compiler. The built applications run without issues during the course of the tests.

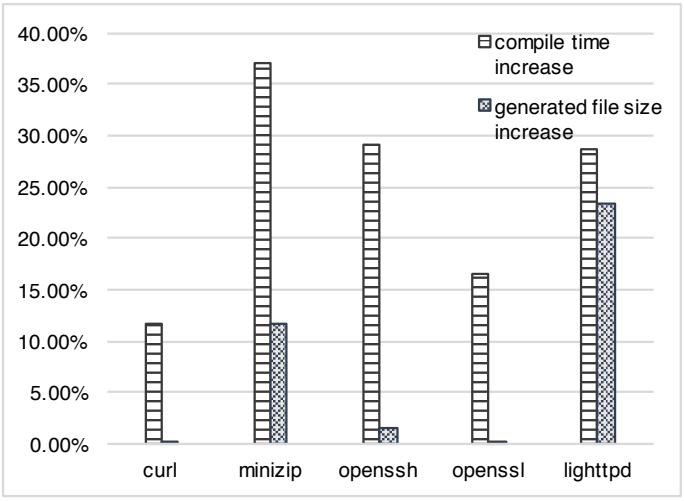

Fig. 5: The time and space overhead incurred by S-compiler during the offline compilation and instrumentation phase

Performance Tests: This group of tests examines the runtime performance of shreds and s-pools. We performed both micro-benchmarkings and end-to-end tests, which respectively reveal the performance cost associated with shreds' critical operations and the overhead exhibited in the 5 applications retrofitted with shreds.

In the micro-benchmarking tests, we developed unit test programs that force shreds to go through the critical operations and state changes, including shred entry, exit, and context switch. We measured the duration of these operations and state changes, and then compared them with the durations of the equivalent or related operations without shreds. Figure 6 shows the absolute time needed for a context switch that preempts a shred-active thread, a regular thread, and a regular process, respectively. It is obvious that, switching shred-active threads is marginally more expensive than switching regular threads (about $100\mu s$ slower); switching shred-active threads is much faster than a process context switch. This is because when a shred is preempted, S-driver does not need to make any change to page tables or TLB. Instead, it only performs a single DACR reset operation, which is very lightweight.

We also compared the time needed for completing the shred API calls (invoking `ioctl` internally) with several reference system calls, as shown in Figure 7. The `getpid`, one of the fastest system calls, serves as a baseline for comparison. The `shred_enter` API is compared with the `clone` system call (without address space change), and is slightly faster, which means creating a shred takes less time than creating a thread. The s-pool allocation API is mildly slower than `mmap` due to the additional domain configurations. But the overhead is low enough to easily blend in the typical program performance fluctuations.

Furthermore, we measured the performance improvement enabled by the lazy domain adjustment optimization. We applied shreds to five SPEC CINT2006 benchmark programs written in C (Figure 8), where a number of shreds were created to perform intensive access to s-pools. We note that this test is designed only for the performance evaluation purpose while recognizing that these benchmark programs do not need shreds' protection. The result shows that in all but one case the optimization brings the overhead under 1% whereas the non-optimized implementation of shreds incurs an average overhead of 2.5%.

Those micro-benchmark tests together indicate that the shred primitives are lightweight and the performance impact that shred state changes and s-pool operations may pose to the application or the system is very mild.

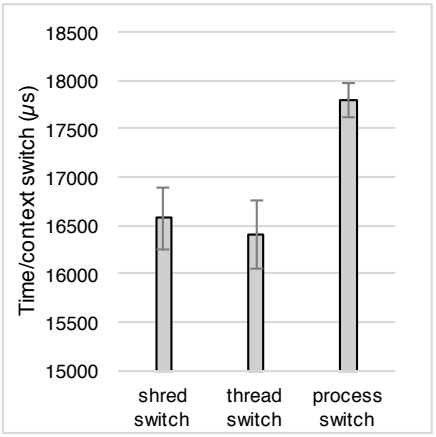

Fig. 6: The time needed for a context switch when: (1) a shred-active thread is switched off, (2) a regular thread is switched off but no process or address space change, and (3) a regular thread is switched off and a thread from a different process is scheduled on.

In the end-to-end tests, we let each of the 5 open-source applications perform a self-contained task twice, with and without using shreds to protect their secret data (*e.g.,* Lighttpd fully handling an HTTP auth login and OpenSSL carrying out a complete RSA key verification). We instrumented the applications with timers. For each application, we manually drove it to perform the task, which fully exercises the added shreds. We measured both the time and space costs associated

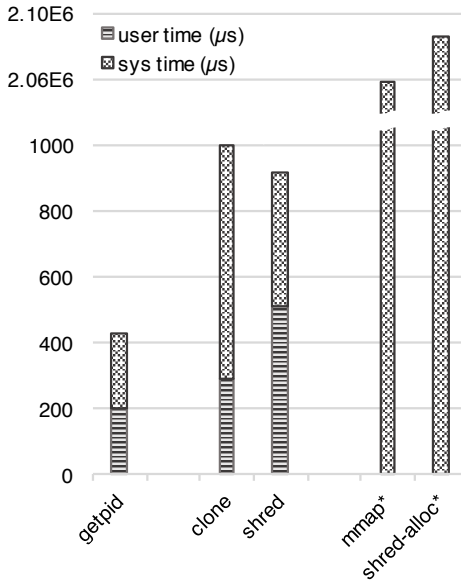

Fig. 7: Invocation time of shred APIs and reference system calls (the right-most two bars are on log scale). It shows that shred entry is faster than thread creation, and s-pool allocation is slightly slower than basic memory mapping.

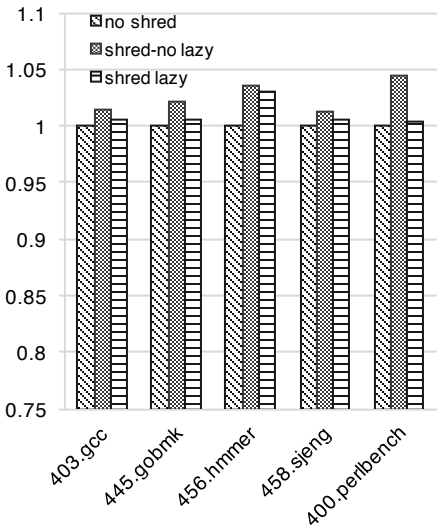

Fig. 8: Five SPEC2000 benchmark programs tested when: (1) no shred is used, (2) shreds are used but without the lazy domain adjustment turned on in S-driver, and (3) shreds are used with the lazy domain adjustment.

with using shreds in these tests. The absolute costs and the relative increases are shown in Table IV. On average, the per-task slow down among the applications is 4.67% and the memory footprint increase is 7.26%. The results show that shreds are practical for real applications of various sizes and functionalities. The overhead is hardly noticeable to the end users of the applications.

Security Coverage Test: Finally, we tested the coverage of shred protection in the 5 modified applications. These tests not only check if the shred adoption is correct and complete in these applications, but also demonstrate the security benefits uniquely enabled by shreds in these applications. We conducted these tests using a simple memory scraper that scans each application's virtual memory in search for the known secrets. The tests simulate the most powerful in-process abuse where an adversary has full visibility into the user-space virtual memory of the application and can perform brute-force search of secrets. For each application, our memory scraper runs as an independent thread inside the application and verifies if any instance of the secret data can be found in memory via a value-based exhaustive search. We ran this test in two rounds, one on a vanilla version of the application and the other on the shred-enabled version.

In the first round, where shreds are not used, the memory scraper found at lease one instance of the secret values in memory for all the applications, which means that these secrets are subject to in-process abuse. In the second round, where shreds are used, the memory scraper failed to detect any secrete in the applications' memory, which means that the secrets are well contained inside the s-pools and protected from in-process abuse. The results show that, the applications have correctly adopted shreds for processing the secret data in memory and stored such data only in s-pools. Moreover, the tests show that, without significant design changes, applying the shred primitives in these real applications creates needed protection for the otherwise vulnerable passwords, crypto keys, and user credentials.

VI. Related Work

Our system is related to the following lines of works, in terms of the addressed problems or the employed techniques.

Program module isolation: The previous works have studied the problem of isolating the executions of mutually distrusting modules, ranging from libraries in user-space programs to drivers in the OS. SFI [13] and its variants [17], [18] establish strict boundaries in memory space to isolate potentially faulty modules and therefore contain the impact resulted from the crashes or malfunctions of such modules. SFI has also been extended to build sandboxes for untrusted plugins and libraries on both x86 [19], [20] and ARM [21], [22]. Extending module isolation into kernel-space, some previous works [18], [23] contain faulty drivers as well as user-space modules. Unlike these works, which focus on fault isolation or sandboxing, our work aims to prevent the in-process memory abuse launched by either vulnerable or malicious code. Our work allows developers to run sensitive code in flexibly-defined and lightweight execution units (*i.e.,* shreds), where the code has exclusive access to private memory pools, in addition to the regular memory regions, and the execution is protected from other code running (concurrently) in the same address space. The aforementioned works require verification and instrumentation of all untrusted code modules, whereas our work only needs to analyze and harden trusted in-shred code. We repurpose the ARM memory domain to efficiently realize the design of shreds and the protection against in-process abuse. Furthermore, SFI and similar techniques assume that isolated modules should be logically independent and not interact closely, whereas shreds neither impose such restrictions nor incur additional overhead when accessing regular memory, invoking third-party library functions, or making system calls.

Process- and thread-level isolation: Arranging program components into different processes has long been advocated as a practical approach to achieving privilege and memory separation [8]–[10]. Many widely used software, such as OpenSSH and Chrome, have adopted this approach. Separated components run in their own address spaces and are immune from memory abuse by other components. However, process separation faces three major limitations when being used for defending memory abuse. First, due to the coarse granularity of a process, memory abuse may still happen inside a component process as a result of a library call or a code injection, as shown in several real attacks on Chrome. Second, using process separation usually requires major software design changes due to the added concurrency and restrictions, which prevents wide adoption. Third, process separation can cause high overhead, particularly when separated components frequently interact. Wedge enables thread-level memory isolation [3]. While incurring slightly lower overhead than process-level isolation, it still suffers from the fixed granularity and require major software changes to be adopted. In comparison, shreds are flexibly grained and easy to adopt. Shreds are also more efficient because, unlike the aforementioned works, our design does not rely on the heavy paging-based memory access control.

Protected execution environments: A number of systems were proposed for securely executing sensitive code or performing privileged tasks. Flicker [24] allows for trusted code execution in full isolation to OS or even BIOS and provides remote attestation. TrustVisor [25] improves on performance and granularity with a special-purpose hypervisor. SeCage [11] runs sensitive code in a secure VM. SICE [26] protects sensitive workloads purely at the hardware level and supports current execution on multicore platforms. SGX [12], an upcoming feature in Intel CPUs, allows user-space programs to create so-called enclaves where sensitive code can run securely but has little access to system resources or application context. In general, these systems are designed for self-contained code that can run independently in isolated or constrained environ-

TABLE IV: End-to-end overhead observed while tested programs performing a complete task: the left-side part of the table shows the executing time and the right-side part shows the memory footprint.

	End-to-end time			**Memory footprint(Max RSS)**		
	w/ shred(ms)	w/o shred(ms)	time increase	w/ shred(KB)	w/o shred(KB)	size increase
curl	154	163	5.80%	4520	5104	12.90%
minizip	23770	25650	7.90%	3004	3064	1.90%
openssh	158.1	163.3	3.20%	3908	4644	18.80%
openssl	2502	2546	1.75%	3892	3908	0.40%
lighttpd	501	525	4.70%	3364	3440	2.30%
Avg.			4.67%			7.26%

ments. They are neither suitable nor practical for preventing memory abuses, which can target data or code that cannot be jailed in these isolated environments. In addition, these systems do not need to consider the case where the protected execution can be exploited, whereas our design does and enforces security checks on in-shred executions.

Memory encryption and protection: Several memory protection mechanisms were proposed before. Overshadow [27] uses virtualization to render encrypted views of application memory to untrusted OS, and in turn, protects application data. Mondrian [28] is a hardware-level memory protection scheme that enables permission control at word-granularity and allows memory sharing among multiple protection domains. Another scheme [29] provides memory encryption and integrity verification for secure processors. While offering strong protection, these schemes all require hardware modifications have not been adopted in real-world. In fact, this work was partly motivated by the lack of a practical and software-based memory protection mechanism. Recently, protecting cryptographic keys in memory became a popular research topic. Proposed solutions range from minimizing key exposure in memory [30]–[32], to avoiding key presence in the RAM by confining key operations to CPUs [33], [34], GPUs [35], and hardware transactional memory [36]. Although effective at preventing key thefts, a major common type of memory abuse, these works can hardly protect other types of sensitive data or code in memory.

Dynamic information flow tracking: Many previous works have used dynamic flow tracking for detecting and defending a range of attacks, including privacy leaks [37] and control flow manipulations [38]. HiStar [39] and Flume [40] enabled system-wide tracking. While information flow tracking can be an ideal solution to memory abuse in theory, it is arguably difficult to use in reality, especially for average programmers. In contrast, our work takes a more practical approach to address a less broad security issue. We aim to provide easy-to-use primitives that help developers efficiently protect their sensitive data and code.

Granular sandbox and compartmentalization: Some recent works proposed fine-grained and flexible application sandbox [15], [41] and compartmentalization [42] frameworks. These works mainly aim at mitigating memory-related exploitations by reducing the capabilities and privileges for untrusted or vulnerable code. In contrast, our work adopts a reversed model of trust: code in an application is by default untrusted and only the explicitly created and statically verified shreds are given the extra privilege during runtime to access the associated s-pools. However, despite the difference between their goals and ours, shreds are related to this line of woks for two reasons: (1) we faced a same technical challenge of efficiently isolating in-process memory, and overcame it via a new and effective approach suitable to our goal, and (2) shreds can employ compartmentalization to achieve more systematic mediation of untrusted code, as discussed in § III-D.

VII. Conclusion

We propose shreds, a set of OS-backed programming primitives that addresses developers' currently unmet needs for fine-grained, convenient, and efficient protection of sensitive memory content against in-process adversaries. A shred can be view as a flexibly defined segment of a thread execution (hence the name). Each shred is associated with a protected memory pool, which is accessible only to code running in the shred. Unlike previous works, shreds offer in-process private memory without relying on separate page tables, nested paging, or even modified hardware. Plus, shreds provide the essential data flow and control flow guarantees for running sensitive code. We have built the compiler toolchain and the OS module that together enable threads on Linux. We demonstrated the usage of shreds and evaluated their performance using 5 non-trivial open source software, including OpenSSH and Lighttpd. The results show that shreds are fairly easy to use and incur low runtime overhead.

VIII. Acknowledgment

We thank the anonymous reviewers for their insightful comments. We thank our shepherds, Robert Watson and Brent Kang, for their guidance on the final paper revisions. We thank our colleagues at the National Security Institute at Stony Brook, in particular, Mingwei Zhang, for their feedback to the work. This project was supported by the National Science Foundation (Grant#: CNS-1421824 and CNS-1514142) and the Office of Naval Research (Grant#: N00014-15-1-2378).

Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Office of Naval Research.

REFERENCES

[1] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, 2014, pp. 475–488.

[2] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "iris: Vetting private api abuse in ios applications," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 44–56. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813675

[3] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: Splitting applications into reduced-privilege compartments." in *NSDI*, vol. 8, 2008, pp. 309–322.

[4] "Memory domains," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211k/Babjdffh.html.

[5] J. Corbet, "Memory protection keys," https://lwn.net/Articles/643797/, May 2015.

[6] D. Hansen, "[rfc] x86: Memory protection keys," https://lwn.net/Articles/643617/, May 2015.

[7] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[8] D. Kilpatrick, "Privman: A library for partitioning applications." in *USENIX Annual Technical Conference, FREENIX Track*, 2003, pp. 273–284.

[9] N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation." in *USENIX Security*, vol. 3, 2003.

[10] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, 2004, pp. 57–72.

[11] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, 2015.

[12] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013, pp. 1–1.

[13] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5. ACM, 1994, pp. 203–216.

[14] "Domain access control register," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0434b/CIHBCBFE.html.

[15] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for unix." in *USENIX Security Symposium*, 2010, pp. 29–46.

[16] "clang: a c language family frontend for llvm," http://clang.llvm.org/.

[17] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 45–58.

[18] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 75–88.

[19] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the x86." in *USENIX Annual Technical Conference*. Boston, MA, 2008, pp. 293–306.

[20] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 2009, pp. 79–93.

[21] "Arm 32-bit sandbox," https://developer.chrome.com/native-client/reference/sandbox_internals/arm-32-bit-sandbox. [Online]. Available: https://developer.chrome.com/native-client/reference/sandbox_internals/arm-32-bit-sandbox

[22] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for arm," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 558–569.

[23] R. Strackx and F. Piessens, "Fides: Selectively hardening software application components against kernel-level or process-level malware," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 2–13.

[24] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4. ACM, 2008, pp. 315–328.

[25] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "Trustvisor: Efficient tcb reduction and attestation," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 2010, pp. 143–158.

[26] A. M. Azab, P. Ning, and X. Zhang, "Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 375–388.

[27] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2. ACM, 2008, pp. 2–13.

[28] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 304–316. [Online]. Available: http://doi.acm.org/10.1145/605397.605429

[29] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 339.

[30] K. Harrison and S. Xu, "Protecting cryptographic keys from memory disclosure attacks," in *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*. IEEE, 2007, pp. 137–143.

[31] Akamai Technologies, "Secure storage of private (rsa) keys," https://lwn.net/Articles/594923/.

[32] MSDN, "Securestring class," https://msdn.microsoft.com/en-us/library/system.security.securestring.aspx.

[33] T. Müller, F. C. Freiling, and A. Dewald, "Tresor runs encryption securely outside ram." in *USENIX Security Symposium*, 2011, pp. 17–17.

[34] L. Guan, J. Lin, B. Luo, and J. Jing, "Copker: Computing with private keys without ram," in *21st ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[35] G. Vasiliadis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Pixelvault: Using gpus for securing cryptographic operations," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1131–1142.

[36] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang, "Protecting private keys against memory disclosure attacks using hardware transactional memory," in *Security and Privacy (SP), 2015 IEEE Symposium on*, May 2015, pp. 3–19.

[37] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: protecting sensitive data leaks using application-level taint tracking," *ACM SIGOPS Operating Systems Review*, vol. 45, no. 1, pp. 142–154, 2011.

[38] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Acm Sigplan Notices*, vol. 39, no. 11. ACM, 2004, pp. 85–96.

[39] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 263–278.

[40] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, "Information flow control for standard os abstractions," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 321–334.

[41] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged cpu features," in *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 335–348.

[42] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 20–37.