Securing Low-level System Software with Symbolic Execution

PhD Thesis Proposal

Changming Liu

Northeastern University
Khoury College of Computer Sciences

PhD Committee

Long Lu, Northeastern University
Engin Kirda, Northeastern University

May 4, 2023

Abstract

Low-level system software are the fundamental components of today's cyberspace. They operate the power grids (e.g., the Programmable Logic Controller), factory machines as well as our home devices (e.g., the Linux kernel). Unlike their desktop-side or server-side counterparts, they feature in 1). Intensive and direct interactions with the hardware without the abstraction from the operating system (OS), 2). Relatively large code space, and 3). Usually employs an event-driven design. These features uniquely pose significant challenges on automatic detection of their defects. These defects subsequently lead to all kinds of security problems and attacks.

Symbolic execution, at the same time, has been extensively studied to become the cornerstone technique of automatically discovering the defects inside the desktop or server programs. Unfortunately, due to the aforementioned challenges, it has not achieved so much success in the low-level system field as it has done in those traditional OS-based environments.

In this thesis, I present three novel systems which address these challenges in order to better apply symbolic execution, this powerful program analysis tool, to secure the low-level system software. First, to address the intensive interactions with the hardware, I present SPEAR, a novel symbolic execution framework which runs the firmware directly on the real device while channelling it to the symbolic engine on the workstation, taking the advantages of both worlds. Second, to scale symbolic execution to a large code base, I present KUBO, an inter-procedural static analysis tool based on symbolic execution to reason the undefined behaviors (UB) inside the linux kernel. KUBO uniquely combines UB Sanitizers and backward symbolic execution to statically and soundly confirm the triggerability of the UB bugs inside the linux kernel. Finally in the project of SHIELD, because of low-level system software's event-driven design, I incorporate the interrupt firing sequence as well as the code coverage into mutation strategy as well as constraint solving.

Contents

1 Introduction

1.1 Low-level system software security

Low-level system software plays a pivot role in the whole cyberspace that we live in – they run everywhere because they connect the users who send commands through them and the hardware that physically performs all kinds of computational tasks. Conceptually, they accept and parse commands from the user, break down the tasks, convey and execute them on the real hardware. They are essential for this information age and the world. Noticeable examples of these low-level system software for today are 1). the OS kernel (e.g., the Linux kernel), and 2). the firmware. The OS kernel runs mostly on desktop and server environments to efficiently drive the common commodity hardware pieces (e.g., hard drive, CPU and RAM) and deal with the users' requests while the firmware usually runs in a more constrained hardware environment like the Micro-control Unit (MCU) and is not as user-facing as the former.

Because of such an important role they play, they are usually placed in a high-privileged execution environment and subsequently suffer the victim of extremely calculated and powerful cyberattacks [70, 75, 81]. Due to the fundamental nature of these critical software, patching their vulnerabilities after deployment is usually delayed, inconvenient and error-prone which leaves a window for attacks.add citation here As a result, detecting the potentially exploitable bugs before the software being deployed in practice is crucial to mitigate these security threats.

In terms of reasoning and analyzing software, symbolic execution as well as fuzz testing have seen huge success in various aspects of securing normal user-space programs. In fact, seminal works add citation here have identified countless potentially dangerous bugs in various user-space programs. Also since we have access to their source code, the efficacy of bug detection from these approaches are usually more powerful than when there is only binary. Thus it would be very tempting to apply these efficient bug-detecting techniques to the low-level system software to identify their defects.

However, unlike the normal user-space programs which enjoy a normalized and abstract execution environment provided by the operating system, low-level system software runs in much more complex and diverse environments. This inevitably poses unique technical challenges when applying these well-established bug detection techniques to them. We identify these technical challenges as follows:

Direct and intensive interaction with the hardware: Unlike user-space programs which interact with the hardware through the well-defined system calls, low-level system software directly interact with the hardware through hardware-provided interfaces, namely, 1). Memory-map Input and Output (MMIO), 2). Interrupt and 3). Direct Memory Access (DMA). So when designing a symbolic execution or fuzz testing framework, instead of focusing on handling the generic well-defined system call interfaces, one has to consider these hardware dependencies. This mainly incurs three problems:

- Limited visibility and computing resource: User-space program enjoys much visibility and abundance of computing resources while the low-level system software is not meant for being debugged thus has much less visibility, debugging utility as well as computing resources (e.g, RAM). This is problematic since most symbolic execution and fuzz testing frameworks are built on the assumptions that it has full visibility into the programs' states and relatively large amount of computing resources.

- Hardware fidelity: Due to the diversity of the hardware, apart from directly accessing them, many existing works tried to emulate the hardware or to abstract them away. But hardware by itself is much like the software in terms of containing complex logic, thus simple abstraction usually lead to low hardware fidelity thus false positives.

- Handle hardware interfaces properly: Even with the aforementioned two problems solved, there is no existing work that tried to handle these three hardware interfaces like they do for the system calls. It is still yet an open problem to handle the MMIO, interrupt, and DMA properly and authentically.

Large code base: Low-level system software tend to have relatively large code base mainly because 1). On the hardware's front, as the hardware usually contains complex functionalities, in order to fully drive each feature from the hardware, the software on top of it needs to enumerate and handle each one of them. 2). On the user's front, it needs to accept, parse and break down the relatively semantically-rich user commands and formulate them into machine-executable tasks, this also takes a large code base to achieve.

Heavily event-driven design: Due to the critical role that the low-level system software is playing, its performance is carefully optimized and essential to all the applications that are built on top of it. As a result, an event-driven design is usually employed to further improve performance. However, rarely does the existing works that take these events into consideration during the design of their solutions. They usually just use a simple heuristic-based method, which often can not be realized in practice.

In this thesis, we design three different novel solutions targeting the aforementioned challenges respectively based on the current state-of-the-art.

1.2 Thesis Statement

In light of the current dire security situation in the low-level system software, also to apply symbolic execution as well as fuzz testing to better secure the low-level system software, my research takes three steps to refine and improve the symbolic execution and fuzz testing techniques to make them more fit for those critical low-level software that we daily depend on.

First, we need to have fast and high-fidelity access to the real hardware so that the quality of the software testing can be drastically improved. Current solutions either connect to the real hardware devices through debugging port or emulate them in the emulators. The former is costly, requires high customization, and does not support all the hardware features. The latter causes high false positives, i.e., the bugs it detected often cannot be realistically replayed in the real hardware.

Second, we need to tailor the costly symbolic execution towards the large code base that the low-level system software usually has. At least for certain type of bugs, we can optimize the symbolic execution strategy to reason those types of bugs more efficiently so that it can scale to a large code base.

Lastly, we need to adapt those bug detection techniques towards a more event-driven design. The current bug detection techniques mainly focus on testing sequential code without worrying about interrupt, as interrupts are abstracted away by the operating system from the user-space programs. But interrupt serves as an essential part for low-level system software, thus we need to incorporate interrupts into the design for a more efficient and comprehensive testing.

1.3 Approaches Overview

In this thesis, I present three novel techniques aiming to make symbolic execution and fuzz testing more efficient and effective for the critical low-level software.

First, I present SPEARadd citation, a new symbolic execution framework designed for embedded applications which uniquely provides high-fidelity and high-speed execution environment. Old symbolic execution designs suffer when the firmware interacts with the hardware. More specifically, due to their migrating firmware from its native environments to the emulators running on the workstation, they have to either emulate those missing peripherals or forward peripheral accesses to the real hardware. The former usually results in false program state, the latter is slow. SPEAR uniquely runs the concrete execution on the real hardware. The concrete execution is only responsible for reporting runtime information necessary for the expensive symbolic execution running on the workstation. This simple design ensures the highest of hardware fidelity since all concrete execution is carried out on the real hardware. It is also very fast out-performing the state-of-the-art symbolic execution framework by 1.5 to 2 times.

Second, I present KUBO [72], a scalable symbolic execution based static detector for undefined behaviors in the OS kernels. Undefined behaviors plague the OS kernel because the OS kernel, as an important example of the system software, has to deal with all sort of different architectures and hardware. While C programming language, the predominant language used in the low level system software defines all different expected behaviors, it still has many behaviors that are left undefined due to the architectural differences. For example, signed integer overflow is a well-known undefined behavior in C language standard [13]. Although for one specific architecture, e.g., x86, its behavior is fixed, i.e., overflown integer will wrap around (i.e., $\mathbf{INT\_MAX + 1 = INT\_MIN}$). However, since on other architectures (e.g., the GPU) overflown integer can saturate rather than wrapping around. These irreconcilable differences make it impossible for the C language standard to pick one expected behavior. Previous works cannot scale undefined behavior detection to the whole kernel code space. They usually just do an intra-procedural per-function analysis, leaving false positive to be as high as 87%. In KUBO, we examined all the undefined-behavior-fixing patches in linux kernel over the past several years and systemized the bug triggering inputs from the userspace. In doing this, we constructed the bare minimal program slice for determining the triggerability of each UB bug thus scaling symbolic execution while taking advantage of its precision. In our experiment we found 23 new undefined behavior bugs in the linux kernel out of which 14 are confirmed.

Finally, I propose Shield, the first hybrid testing system driven by bug search and empowered with buggy condition verification. Differing from the existing hybrid testing tools, Savior prioritizes the concolic execution to solve branch predicates guarding more potential vulnerabilities and then drives fuzz testing to reach code with more potential vulnerabilities. Going beyond that, Savior inspects all vulnerable candidates along the running program path in concolic execution. By modeling the faulty situations with SMT constraints, it solves proofs of valid vulnerabilities and outputs concrete test cases. Our evaluation shows that the bug-driven approach outperforms the state-of-art code coverage driven hybrid testing tools in vulnerability detection. In our preliminary experiments comparing Savior with state-of-the-art software testing techniques on the widely used LAVA benchmark [47]. Within 5 hours, in addition to triggering all bugs in `base64`, `md5sum` and `uniq`, Savior found 1904 bugs in `who` and many other unlisted bugs. This is, to the best of our knowledge, by far the best results in the existing literatures.

2 Related Work

In this section, I survey related works in the structure of mechanisms providing new memory protection and better isolations between mutually untrusted components; mitigations of code reuse attacks and automated bug finding systems that identify software defects leading to in-process memory abuse.

2.1 Memory Protection and Isolation

Several memory protection mechanisms were proposed before. Overshadow [30] uses virtualization to render encrypted views of application memory to untrusted OS, and in turn, protects application data. Mondrian [117] is a hardware-level memory protection scheme that enables permission control at word-granularity and allows memory sharing among multiple protection domains. Another scheme [103] provides memory encryption and integrity verification for secure processors. Recently, protecting cryptographic keys in memory became a popular research topic. Proposed solutions range from minimizing key exposure in memory [61, 11, 83], to avoiding key presence in the RAM by confining key operations to CPUs [84, 57], GPUs [111], and hardware transactional memory [58].

There are numerous research proposing isolation for untrusted components, ranging from libraries in user-space programs to drivers in the OS. Software Fault Isolation (SFI [112]) and its variants [27, 48] 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 [50, 119] and ARM [2, 126]. Extending module isolation into kernel-space, some previous works [48, 102] contain faulty drivers as well as user-space modules. Separating program components into different processes has long been advocated as a practical approach to achieve privilege and memory separation [67, 91, 24, 43]. However, process separation can cause high overhead, particularly when separated components frequently interact. Wedge enables thread-level memory isolation [21]. While incurring slightly lower overhead than process-level isolation, it still suffers from the fixed granularity and require major software changes to be adopted. Some recent works proposed fine-grained and flexible application sandbox [115, 17] and compartmentalization [116] frameworks. These works mainly aim at mitigating memory-related exploitations by reducing the capabilities and privileges for untrusted or vulnerable code.

A number of systems were proposed for securely executing sensitive code or performing privileged tasks. Flicker [79] allows for trusted code execution in full isolation to OS or even BIOS and provides remote attestation. TrustVisor [78] improves on performance and granularity with a special-purpose hypervisor. SeCage [73] runs sensitive code in a secure VM. SICE [15] protects sensitive workloads purely at the hardware level and supports current execution on multicore platforms. SGX [80], a recent 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.

2.2 Mitigations of Code Reuse Attacks

Over the years, there has been an ongoing race between code reuse attacks (e.g, ROP) and corresponding defense countermeasures. Such code reuse attacks keep evolving into new forms with more complex attack steps (e.g., Blind-ROP [20], JIT-ROP [100]). To defend against them, three categories of countermeasures (e.g., ASLR, XOM, CFI) have been proposed from different perspectives.

ASLR is a practical and popular defense deployed in modern operating systems to thwart code reuse attacks [106]. It randomizes the memory address and makes the locations of ROP gadgets unpredictable. However, the de-facto ASLR only randomizes the base address of code pages. It becomes ineffective when facing recent memory-disclosure-based code reuse attacks [20, 100]. Such attack explores the address space on-the-fly to find ROP gadgets via a memory disclosure vulnerability. Although fine-grained ASLR increases the entropy of randomization, such as compile-time code randomization [18] and load-time randomization [42, 62, 66, 114, 69] the memory disclosure attack is not directly addressed, since code pages can still be read by attackers [100]. Runtime randomization [40, 33, 19] is thus proposed to introduce more uncertainty into the program's address space.

To address the memory disclosure attack, researchers proposed execute-only but non-readable ($R \otimes X$) memory pages to hinder the possibility of locating reusable code (or ROP gadgets). However, one fundamental challenge to achieve this defense is that it is non-trivial to identify and

separate legitimate data read operations in code pages. When source code is available, existing works like Readactor [38, 39] and LR2 [23] rely on compilers to separate data reads from code pages and then enforcing XOM via either hardware-based virtualization or software-based address masking. On the other hand, for COTS binaries, which are more common in the real-world scenario, XnR [16] blocks direct memory disclosure by modifying the page fault handler in operating systems to check whether a memory read is inside a code or data region of a process. However, it cannot handle embedded data mixed in code region. HideM [53] utilizes split-TLB features in AMD processors to distinguish direct code and data access to different physical pages to prevent reading code. Unfortunately, recent processors no longer support split-TLB.

Enforcing CFI is a general defense against code reuse attacks. Proposed a decade ago by PaX Team and Abadi et al. [107, 10], CFI has been tuned by researchers over the years [109, 86, 87, 77, 82, 108], from its early form coarse-grained CFI to its current mature appearance as fine-grained CFI. The fundamental difference is that a coarse-grained CFI allows forward edges in the control flow graph (CFG) to point at any node in the graph and back-ward edges to return to any call preceded destination, whilst a fine-grained CFI has a more precise set of destinations for both forward and backward edges. bin-CFI [124] and CCFIR [123] enforce the coarse-grained CFI policy on Linux and windows COTS binaries respectively. Unfortunately, enforcing a fine-grained CFI requires a more precise CFG to be built as the ground truth, which is difficult to obtain in practice based on static analysis, even when source code is available. In addition, researchers found that it is still possible to launch code reuse attacks when fine-grained CFI solution is in place due to the difficulty of extracting a perfect CFG in practice [56, 41, 26, 49].

2.3 Automated Bug Finding Tools

Symbolic execution, a systematic program testing method introduced in 1970s [68, 63], has attracted new attentions due to the advances in the satisfiability modulo theory [52, 45, 44]. However, classic symbolic execution suffers from the expensive computation cost as well as the state explosion problem: it solves the feasibility of both conditional branches when reaching a branching pivot point, and generates unbiased new states accordingly. To tackle these issues Sen proposed concolic execution [94], a variant of symbolic execution, by combining the concrete input generation from symbolic execution and the fast execution of random testing. Concolic execution increases the coverage of random testing [55, 54] while also scaling to large applications, hence has been studied in various frameworks [95, 96, 25, 34]. It also plays an critical role in the automated vulnerability detection and exploitation, where the concolic component generates security-related input by incorporating extra safety predicates [28, 14]. However, concolic execution runs in virtual machines, and the execution overhead challenges its application to practical software with rich environmental interactions. In contrast, fuzz testing executes in native speed but it lacks transparent information about the code, whereas mixing these two techniques becomes a promising schema.

Majundar et al. [76] introduced the idea of Hybrid Concolic Testing around one decade ago. This idea offsets the deficiency of both random testing and concolic execution. Specifically, their approach interleaves random testing and concolic execution to deeply explore a wide space of program state. Subsequent development reinforces hybrid testing by replacing random testing with guided fuzzing [88]. This approach could rapidly increase the code coverage by contributing more high-quality seeds to concolic execution. Recently, Driller [101] engineers the state-of-art hybrid testing system. It more coherently combines fuzzing and concolic execution and can seamlessly test a wide range of software systems. Despite of the remarkable advancement in hybrid testing, Driller still suffers from the unsound vulnerability detection. DigFuzz [125] is a more recent work that tries to better coordinate the fuzzing and concolic execution components of hybrid testing. Based on a Monte Carlo algorithm, DigFuzz predicts the difficulty for a fuzzer to explore a path and prioritize exploring seeds with higher difficulty score. Moreover, motivated by the increasing security demands in software systems, researchers have been reasoning the performance of hybrid testing. One common intuition is that hybrid testing is largely restricted by the slowness of concolic executor. In particular, QSYM [121] implements a symbolic executor that tailors the heavy but unnecessary computations involved in symbolic emulation and constraint solving. It can often lead to times of acceleration.

Many recent works focus on improving the capability of code exploration in fuzzing. CollAFL [51] aims to reduce hash collision to avoid false negatives when exploring new code. To generate smarter seeds, ProFuzzer [120] infers the format of inputs and use it as guidance in later stage fuzzing. Along the line of smart fuzzing, Angora [29] assumes a blackbox function for each program condition and uses gradient descent to search for satisfying input bytes. This method is later improved by NEUZZ [98] using a smooth surrogate function to approximate the behavior of the tested program.

3 Fine-grained Execution Units with Private Memory

When it comes to preventing in-process memory abuse, developers are virtually helpless 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.

3.1 Overview

In this section, we present a new execution unit for userspace 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 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 9 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

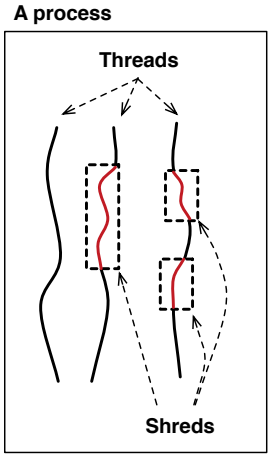


Figure 1: Shreds, threads, and a process

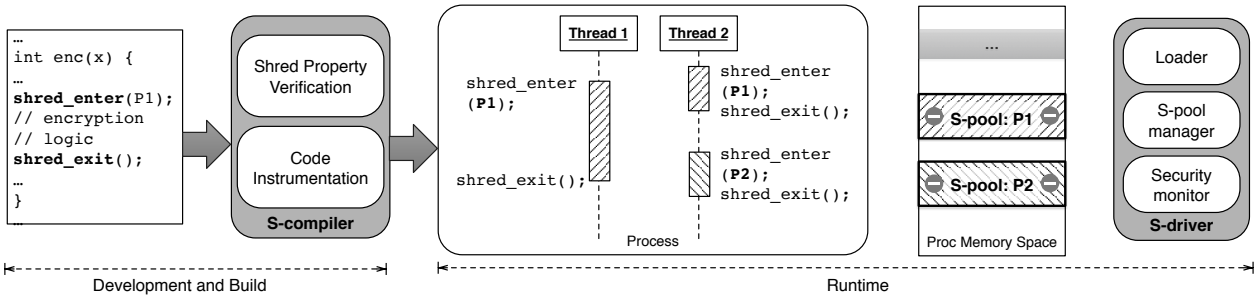


Figure 2: Developers create shreds in their programs via the intuitive APIs and build the programs using S-compiler, which automatically verifies and instruments the executables (left); during runtime (right), S-driver handles shred entrances and exits on each CPU/thread while efficiently granting or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

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 9, 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 [8], to efficiently realize s-pools and enforce shred-based access control. Unlike the previously proposed thread-level memory isolations [21], 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 [37, 59].

We implement S-compiler based on LLVM [71] 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.

3.2 Design

3.2.1 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 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, auth_str_len);
15 +   buffer_copy_string_len(ds->value, cur, pw_len);
16 +   /* add ds to header pointer array */
17 +   array_insert_unique(parsed_headers, ds);
18 +   /* only related shreds can deref ds */
19 +   /* wipe out passwd from input stream */
20 +   memset(cur, 0, pw_len);
21 +   cur += pw_len;
22 +   shred_exit();
23 +  }
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    }
19
```

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     ...
10
```

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)

3.2.2 Security Properties

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.

Next, we explain how we design S-compiler and S-driver together to ensure these properties.

3.2.3 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. In addition, S-compiler checks that code included in a shred follows 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 $P3$.

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 sensitive data in the s-pool are propagated to unprotected memory. To ensure that, the data-flow 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.

3.2.4 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 [37, 59]. 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) [5], 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

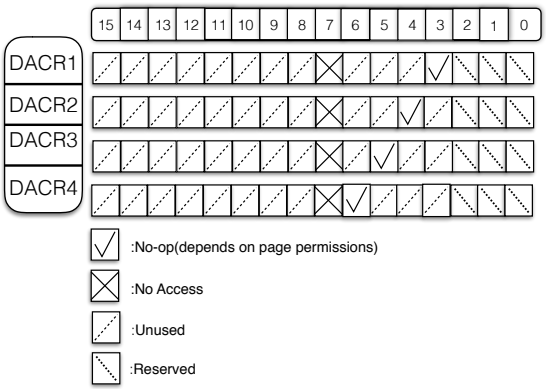


Figure 3: The DACR setup for a quad-core system, where $k = 4$. The first 3 domains ($Dom_0 - Dom_2$) are reserved by Linux. Each core has a designated domain ($Dom_3 - Dom_6$) that it may access when executing a shred. No CPU can access Dom_7.

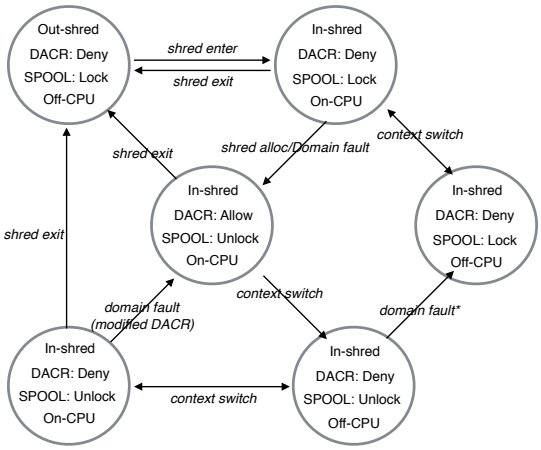


Figure 4: A shred's transition of states

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. 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.

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 [115]. 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.

3.3 Implementation

We built S-compiler based on LLVM [71] and its C front-end Clang [4]. 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).

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 § 3.2. 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 Builder interfaces to insert security checks into the analyzed IR, which are necessary for enforcing the in-shred control flow regulations and preventing dynamic data leaks.

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);
```

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

3.4 Evaluation

Our evaluation sought to answer the following questions:

- How compatible and useful are shreds to real-world programs?
- How do shreds affect the application's and system's performance?

Table 1: 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

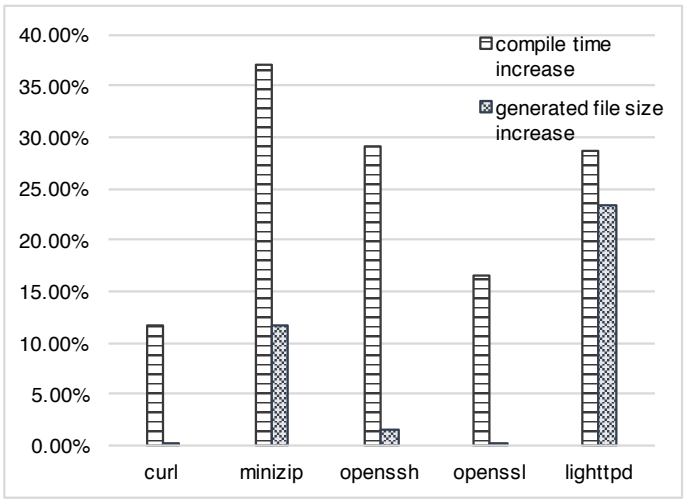


Figure 5: The time and space overhead incurred by S-compiler during the offline compilation and instrumentation phase

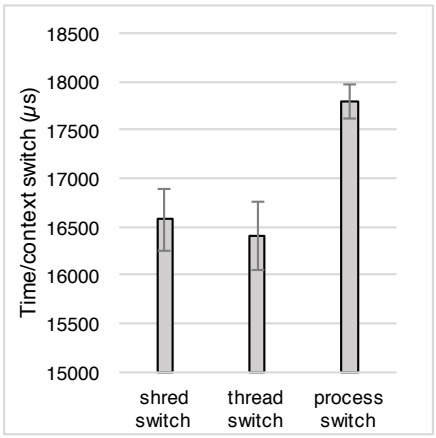


Figure 6: The time needed for a context switch when: (1) a shred-active thread is switched off, (2) a regular thread is switched off but no process or address space change, and (3) a regular thread is switched off and a thread from a different process is scheduled on.

- How shreds help mitigate in-process memory abuse?

Choice of Applications: We selected 5 popular open source applications to evaluate our prototype system. The applications are shown in Table 1, 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.

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.

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 100μs 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.

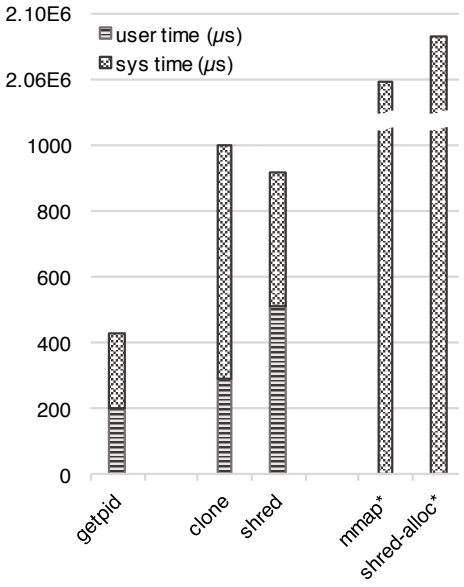


Figure 7: Invocation time of shred APIs and reference system calls (the right-most two bars are on log scale). It shows that shred entry is faster than thread creation, and s-pool allocation is slightly slower than basic memory mapping.

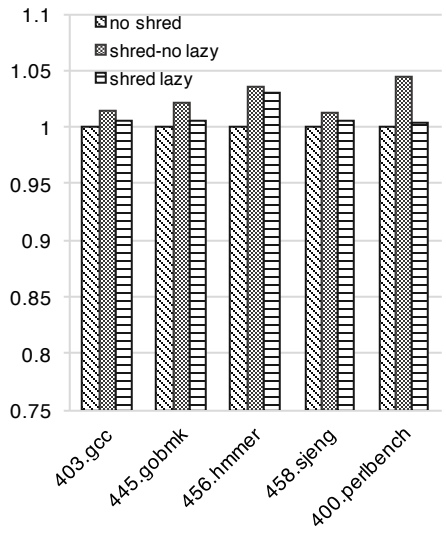


Figure 8: Five SPEC2000 benchmark programs tested when: (1) no shred is used, (2) shreds are used but without the lazy domain adjustment turned on in S-driver, and (3) shreds are used with the lazy domain adjustment.

Table 2: 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%

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 with using shreds in these tests. The absolute costs and the relative increases are shown in Table 2. 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 matches 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.

4 Enabling Execute-Only Memory for COTS Binaries

4.1 Overview

Although creating isolations among mutually untrusted component with finer granularity can largely prevent in-process memory abuse, attackers could still exploit a vulnerable components directly. Numerous studies [74, 99] have shown that attackers are able to exploit code running inside

trusted executing environments, bypassing all deployed mitigations [49, 100, 93]. Such attacks increasingly leverage code-reuse techniques [93, 118, 22] to gain control of vulnerable programs. Since contemporary softwares widely employ code integrity protection techniques, such as data execution prevention (DEP [110]), to prevent traditional code injection attacks. In code reuse attacks, a target application's control flow is manipulated in a way that snippets of existing code (called gadgets) are chained and run to carry out malicious activities.

Knowledge of process memory layout is a key prerequisite for code-reuse attacks to succeed. Attackers need to know the exact binary instruction locations in memory to assemble the chain of gadgets. Commodity operating systems widely adopt address space layout randomization (ASLR), which loads code binaries at random memory locations unpredictable to attackers. Without knowing the locations of needed code or gadgets, attackers cannot build code-reuse chains.

However, *memory disclosure* attacks can use information leaks in programs to de-randomize code locations, thus defeating ASLR. Such attacks either read the program code (*direct* de-randomization) or read code pointers (*indirect* de-randomization). Given that deployed ASLR techniques only randomize the load address of a large chunk of data or code, leaking a single code pointer or a small sequence of code allows attackers to identify the corresponding chunk, infer its base address, and calculate the addresses of gadgets contained in the chunk.

More sophisticated fine-grained ASLR techniques [42, 89, 62, 66, 114] aim at shuffling code blocks within the same module to make it more difficult for attackers to guess the location of binary instructions. Nevertheless, research by Snow et al. [100] proves that memory disclosure vulnerabilities can bypass the most sophisticated ASLR techniques.

Therefore, a robust and effective defense against code-reuse attacks should combine fine-grained ASLR with memory disclosure prevention. Some recent works proposed to prevent memory disclosures using compile-time techniques [38, 39, 23]. Despite their effectiveness, these solutions cannot cover COTS binaries that cannot be easily recompiled and redeployed. These binaries constitute a significant portion of real-world applications that need protection.

In this section, we present Norax [1], which protects COTS binaries from code memory disclosure attacks. The goal of Norax is to allow COTS binaries to take advantage of execute-only memory (XOM), a new security feature that recent AArch64 CPUs provide and is widely available on today's mobile devices. While useful for preventing memory disclosure-based code reuse [100, 20], XOM remains barely used by user and system binaries due to its requirement for recompilation. Norax removes this requirement by automatically patching COTS binaries and loading their code to XOM. As a result, when used together with ASLR, Norax enables robust mitigation against code reuse attacks for COTS binaries.

Norax consists of four major components: *NDisassembler, NPatcher, NLoader, and NMonitor*. The first two perform offline binary analysis and transformation. They convert COTS binaries built for AArch64 without XOM support into one whose code can be protected by XOM during runtime. The other two components provide supports for loading and monitoring the patched, XOM-enabled binaries during runtime. The design of Norax tackles a fundamentally difficult problem: identifying data embedded in code segments, which are common in ARM binaries, and relocating such data elsewhere so that during runtime code memory pages can be made executable-only while allowing all embedded data to be readable.

We apply Norax to Android system binaries running on SAMSUNG Galaxy S6 and LG Nexus 5X devices. The results show that Norax on average slows down the transformed binaries by 1.18% and increases their memory footprint by 2.21%, suggesting Norax is practical for real-world adoption.

4.2 Design

NORAX Workflow: Norax consists of four major components: *NDisassembler, NPatcher, NLoader, and NMonitor*, as shown in Figure 9. The first two components perform offline binary analysis and transformation and the last two provide runtime support for loading and monitoring the patched, XOM-compatible executables and libraries. In addition to disassembling machine code, NDisassembler scans for all executable code that needs to be protected by XOM. A major challenge it solves is identifying various types of data that ARM compilers often embed in the code section, including jump tables, literals, and padding. Unlike typical disassemblers, NDisassembler has to precisely differentiate embedded data from code. Taking input from NDisassembler, NPatcher transforms the binary so that its embedded data are moved out of code sections and their references are collected for later adjustment. After the transformation, NPatcher inserts a unique magic number in the binary so that it can be recognized by NLoader during load-time. NPatcher also stores Norax metadata in the binary, which will be used by NLoader and NMonitor. When a patched binary is being loaded, NLoader takes over the loading process to (*i*) load the Norax metadata into memory, (*ii*) adjust the NPatcher-collected references as well as those dynamically created references to the linker-related sections (e.g .hash, .rela.*), and (*iii*) map all memory pages that contain code to XOM. During runtime, NMonitor, an OS extension, handles read accesses to XOM. While such accesses are rare and may indicate attacks, they could also be legitimate because NPatcher may not be able to completely recognize dynamic references to the relocated embedded data (e.g., those generated at runtime). When there are missed data references, the access will trigger an XOM violation, which NMonitor verifies and, if legitimate, facilitates the access to the corresponding data.

4.2.1 NDisassembler: Static Binary Analyzer

NDisassembler first converts an input binary from machine code to assembly code and then performs analysis needed for converting the binary into an XOM-compatible form. It disassembles the binary in a linear sweep fashion, which yields a larger code coverage than recursive disassembling [12].

However, the larger code coverage comes at a cost of potentially mis-detecting embedded data as code (e.g., when such data happen to appear as syntactically correct instructions). NDisassembler addresses this problem via an iterative data recognition technique. Along with this process, it also finds instructions that reference embedded data.

The data recognition technique is inspired by the following observations:

- Although it is difficult to find all instructions referencing some embedded data at a later point in the running program, it is relatively easy to locate the code that computes these references in the first place.
- To generate position-independent binaries, compilers can only use PC-relative addressing when emitting instructions that need to reference data inside binaries.
- AArch64 ISA only provides two classes of instructions for obtaining PC-relative values, namely the `ldr (literal)` instructions and `adr(p)` instructions.

NDisassembler uses Algorithm 2 to construct an initial set of embedded data (IS) and a set of reference sites (RS). For embedded data whose size cannot be precisely bounded, NDisassembler collects their seed addresses (AS) for further processing. As shown in Line 5–9 in

[1] Norax stands for **NO R**ead **A**nd e**X**ecute.

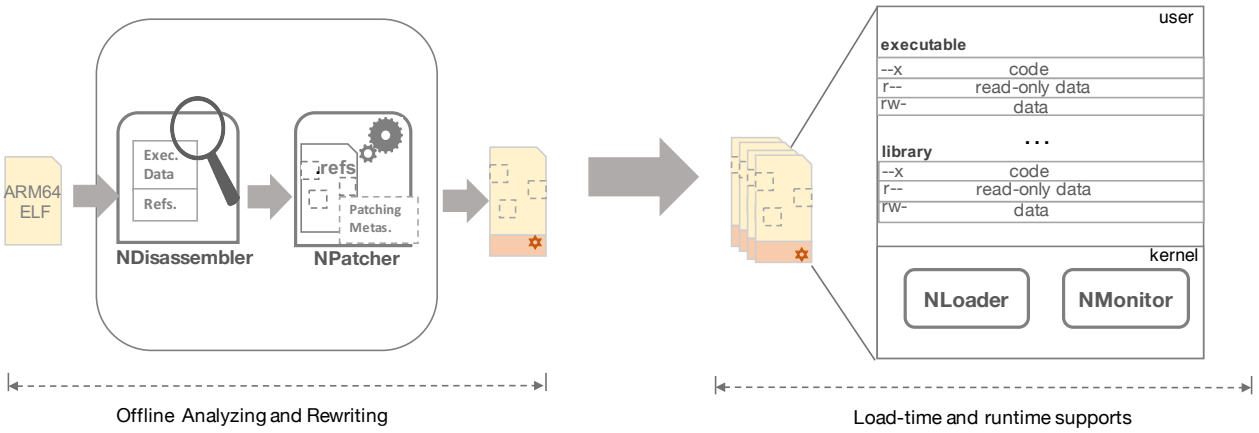


Figure 9: NORAX System Overview: the offline tools (left) analyze the input binary, locate all the executable data and their references (when available), and then statically patch the metadata to the raw ELF; the runtime components (right) create separated mapping for the *executable data sections* and update the recorded references as well as those generated at runtime.

Algorithm 2, since the load size for `ldr-literal` instructions is known, the identified embedded data are added to IS. On the other hand, the handling for `adr` instructions is more involved, as shown in Line 10–27. NDisassembler first performs forward slicing on `xn` — the register which holds the embedded data address. All instructions that have data dependencies on `xn` are sliced, and `xn` is considered *escaped* if any of its data-dependent registers is either (*i*) stored to memory or (*ii*) passed to another function before being killed. In either case, the slicing also stops. If not all memory dereferences based on `xn` can be identified due to reference escaping, the size of the embedded data cannot be determined. Therefore, NDisassembler only adds the initial value of `xn` to AS, as a seed address (Line 24–26).

Line 10–23 of Algorithm 2 deal with the sliced instructions. If a memory load based on `xn` is found, RS is updated with the location of the original address-taking instruction. Moreover, NDisassembler analyzes the address range for each memory load. Note that oftentimes the address range is bounded because embedded data are mostly integer/floating point constants, or jump tables. In the former case, the start address of memory load is typically `xn` plus some constant offset, while the load size is explicit from the memory load instruction. In the latter, well-known techniques for determining jump table size [35] are utilized. In both cases, the identified embedded data are added into IS. However, if there is a single memory load whose address range cannot be bounded, NDisassembler adds the seed address to AS.

If Algorithm 2 is not able to determine the sizes of all embedded data, the initial set (IS) is not complete. In this case, the seed addresses in AS are expanded using Algorithm 3 to construct an over-approximated set of embedded data (DS). The core functions are *BackwardExpand* (line 4) and *ForwardExpand* (line 5). The backward expansion starts from a seed address and walks backward from that address until it encounters a *valid* control-flow transfer instruction: i.e., the instruction is either a *direct* control-flow transfer to a 4-byte aligned address in the address space, or an *indirect* control-flow transfer. All bytes walked through are marked as data and added to DS. On the other hand, the forward expansion walks forward from the seed address. It proceeds aggressively for a conservative inclusion of all embedded data. It only stops when it has strong indication that it has identified a valid code instruction. These indicators are one of the following: (*i*) a *valid* control-flow transfer instruction is encountered, (*ii*) a direct control-flow transfer target (originating from other locations) is reached, and (*iii*) an instruction is confirmed as the start of a function[92]. In the last case, comprehensive control-flow and data-flow properties such as parameter passing and callee saves are checked before validating an instruction as the start of a function.

Finally, DS contains nearly all embedded data that exists in the binary. Although we could further leverage heuristics to include *undecodable* instructions as embedded data, it is not necessary because our conservative algorithms already cover the vast majority (if not all) of them, and the rest are mostly padding bytes which are never referenced. Theoretically, failure to include certain *referenced* embedded data could still happen if a chunk of data can be coincidentally decoded as a sequence of instructions that satisfies many code properties, but in our evaluation of *over 300* stripped Android system binaries, we never encountered such a case. RS contains a large subset of reference sites to the embedded data. Since statically identifying all indirect or dynamic data references may not always be possible, NDisassembler leaves such cases to be handled by NMonitor.

4.2.2 NPatcher: XOM Binary Patcher

Data Relocation: An intuitive design choice is to move the executable data out of the code segment. But doing so affects backward compatibility as the layout of the ELF and the offsets of its sections will change significantly. Another approach is to duplicate the executable data, but this would increase binary sizes and memory footprint significantly.

Instead, NPatcher uses two different strategies to relocate those executable data without modifying code sections or duplicating all read-only data sections. For data located in code segment but are separated from code text (i.e., read-only data), NPatcher does not duplicate them in binaries but only records their offsets as metadata, which will be used by NLoader to map such data into read-only memory pages. For data mixed with code (i.e., embedded data), NPatcher copies them into a newly created data section at the end of the binary. The rationale behind the two strategies is that read-only data usually accounts for a large portion of the binary size and duplicating it in binary is wasteful and unnecessary. On the other hand, embedded data is usually of a small size, and duplicating it in binaries does not cost much space. More importantly, this is necessary for security reasons. Without duplication, code surrounding data would have to be made readable, which reduces the effectiveness of XOM.

Data Reference Collections: NPatcher only collects the references from *.text* to *.text* (embedded data) and to *.rodata* because they can be statically recognized and resolved. Other types of references are either from outside the module or statically unavailable, which are handled by NLoader.

For references to embedded data, NPatcher can directly include them based on NDisassembler's analysis results. But there is one caveat – the instructions used to reference embedded data (i.e., *adr* and *ldr-literal*) have a short addressing range. Therefore, when we map their target data to different memory pages, it is possible that the instructions cannot address or reach the relocated data. To solve this issue without

Algorithm 2: Initial embedded data and references collection

INPUT:
code[] - An array of disassembly output
OUTPUT:
IS - Initial set of embedded data
AS - The set of seed addresses for embedded data
RS - The set of reference sites to embedded data

```
1:  procedure InitialSetCollection
2:      IS = {}
3:      AS = {}
4:      RS = {}
5:      for each (ldr-literal addr) ∈ code[] at curr do
6:          size = MemLoadSize(ldr)
7:          IS = IS ∪ {addr, addr+1, ..., addr+size-1}
8:          RS = RS ∪ {curr}
9:      end for
10:     for each (adr xn, addr) ∈ code[] at curr do
11:         escaped, depInsts = ForwardSlicing (xn)
12:         unbounded = False
13:         for each inst ∈ depInsts do
14:             if inst is MemoryLoad then
15:                 RS = RS ∪ {curr}
16:                 addr_expr = MemLoadAddrExpr(inst)
17:                 if IsBounded(addr_expr) then
18:                     IS = IS ∪ {AddrRange(addr_expr)}
19:                 else
20:                     unbounded = True
21:                 end if
22:             end if
23:         end for
24:         if escaped or unbounded then
25:             AS = AS ∪ {addr}
26:         end if
27:     end for
28: end procedure
```

Algorithm 3: embedded data set expansion

INPUT:
AS - The set of seed addresses for embedded data
IS - Initial set of embedded data
OUTPUT:
DS - conservative set of embedded data

```
1: procedure SetExpansion
2:     DS = IS
3:     for addr in AS do
4:         c1 = BackwardExpand (addr, DS)
5:         c2 = ForwardExpand (addr, DS)
6:         DS = DS ∪ c1 ∪ c2
7:     end for
8: end procedure
```

breaking backward-compatibility, NPatcher generates stub code to facilitate access to out-of-range data. The instructions of short addressing range are replaced with an unconditional branch instruction[2], which points to the corresponding stub entry. The stub code only contains unconditional load and branch instructions pointing to fixed immediate offsets. This design ensures that these stub entries cannot be used as ROP gadgets.

For references to the *.rodata*, there is no addressing capability problem, because *adrp* is used instead of *adr*. However, a different issue arises. There are multiple sources from which such references could come. We identify 5 sources in our empirical study covering all Android system executables and libraries. NPatcher can only prepare the locations of the first three offline while leaving the last two to be handled by NLoader after relocations and symbol resolving are finish.

- **References from code (*.text*)**: these are usually caused by access to constant values and strings.
- **References from symbol table (*.dynsym*)**: when a symbol is located in *.rodata*, there will be an entry in the symbol table, whose *value* field contains the address of the exposed symbol.
- **References from relocation table (*.rela.dyn*)**: for a relocatable symbol located in *.rodata*, the relocation table entry's *r_addend* field will point to the symbol's address.
- **References from global offset table (*.got*)**: when a variable in *.rodata* cannot be addressed due to the addressing limit(e.g., *adrp* can only address +/- 4GB), an entry in the global offset table is used to address that far-away variable.
- **References from read-only global data (*.data.rel.ro*)**: most binaries in Android disable lazy-binding. The *.data.rel.ro* section contains the addresses of global constant data that need to be relocatable. After the dynamic linker finishes relocating them, this table will be marked as read-only, as opposed to the traditional *.data* section.

Finally, the metadata (duplicates and references), the data-accessing stub code and the Norax header are appended to the end of the original binary, as shown in Figure 10. Note that by appending the Norax-related data to the end of the binary, we allow patched binaries

[2] *ADR* can address +/- 1MB, while *B(ranch)* can access +/- 128MB, which is far enough for regular binaries.

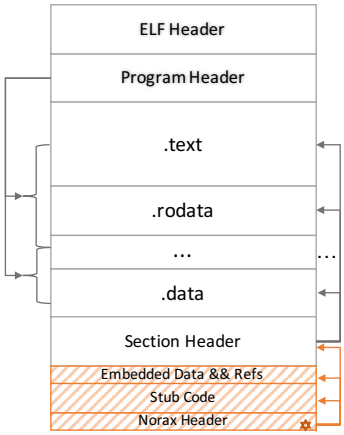


Figure 10: The layout of ELF transformed by NORAX. The shaded parts at the end are the generated NORAX-related metadata.

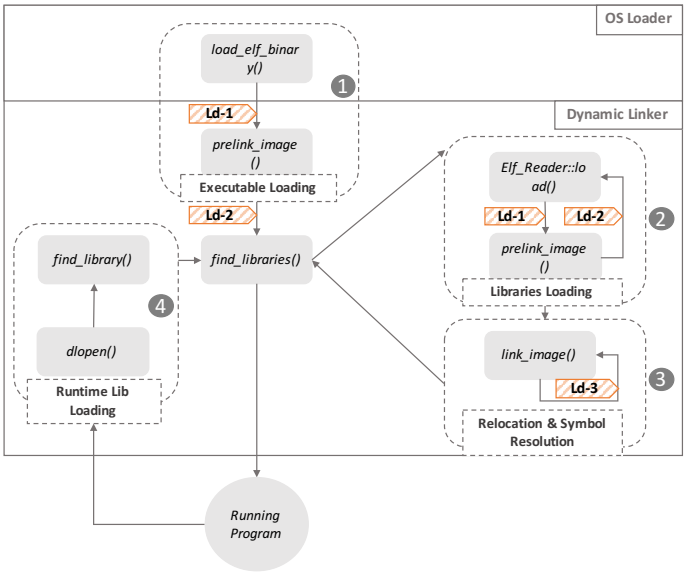


Figure 11: Bionic Linker's binary loading flow, NLoader operates in different binary preparing stages, including module loading, relocation and symbol resolution.

to be backward-compatible. This is because the ELF standard ignores anything that comes after the section header table. As a result, binaries transformed by NPatcher can run on devices without NORAX support installed. They can also be parsed and disassembled by standard ELF utilities such as *readelf* and *objdump*. Moreover, NORAX-patched binaries are compatible with other binary-level security enhancement techniques.

4.2.3 NLoader: Plugin for Stock Loader and Linker

Binaries rewritten by NPatcher remain recognizable by and compatible with the stock loader and linker. They can still function albeit without the XOM protection. New data sections added by NORAX, however, are transparent to the toolchain. They require NLoader's support to complete the binary loading and references updating process before their code can be mapped in XOM. Other than the ones prepared by NPatcher, there are several types of references to executable data which are related to the linker and only available at runtime. Built as a linker/loader plugin, NLoader adjusts these references in the following steps:

- **Ld-1:** It parses and loads NORAX header into memory, including information about the embedded data in *.text* and the stub code accessing embedded data. Then, it creates duplicated mappings for *.rodata* and the linker-referencing sections[3], which have been loaded by the stock linker/loader.
- **Ld-2:** It updates the *.dynamic* section to redirect linker to use the read-only copy of those relocated data sections.
- **Ld-3:** It collects the *.rodata* references from *.got* and *.data.rel.ro*, which are only populated after the relocation is done. It then adjusts all the collected data references in one pass. Eventually, the memory access level of the loaded module is adjusted to enforce the $\mathbf{R} \oplus \mathbf{X}$ policy.

The overall workflow of NLoader is shown in Figure 11. It starts with the executable loading, which is done by the OS ELF loader (Step ①). Then, the OS loader transfers the control to the dynamic linker, which in turns creates a book-keeping object for the just-loaded module. Meanwhile, **Ld-1** is performed to complete the binary loading. Next, the binary's corresponding book-keeping object is then populated with references to those ELF sections used by the linker to carry out relocation and symbol resolution in a later stage. **Ld-2** is then invoked to update these populated references. At this point, the preparation for the executable is done. The linker then starts preparing all the libraries (Step ②). This process is similar to the preparation of executable, thus **Ld-1** and **Ld-2** are called accordingly. When all the modules are loaded successfully in previous steps with their book-keeping objects populated, the linker walks through the book-keeping objects to perform relocation and symbol resolution (Step ③). In this step, **Ld-3** is called for each of the relocated modules to update all those collected references, including the ones from *.got* and *.data.rel.ro* to *.rodata*. This is feasible because the *.got* entries which reference to *.rodata* are populated upfront, same as those in *.data.rel.ro*.

During runtime, the program may dynamically load or unload new libraries (Step ④), as shown in Figure 11, which is also naturally handled by NLoader. To boost performance, once NLoader finishes updating the offline-updatable references, it caches the patched binary so that it can directly load the cached version without going through the whole references adjustment process again next time.

4.2.4 NMonitor: Runtime Enforcement and Safety-net

After being processed by the last three NORAX components, a patched binary that follows the $\mathbf{R} \oplus \mathbf{X}$ policy is ready to run, which is assisted by NMonitor. At runtime, the converted program could still be running with some unadjusted references to the executable data, which belong to the two following possible categories.

- **Missed references to embedded data**: Although in our evaluation we rarely see cases where an access violation is triggered by missed embedded data references, such situation, if mishandled, will cause a program crash. NDisassembler is unable to discover such cases due to the limitation of static analysis. These missed data references would trigger access violations. Note that references to *.rodata* from

[3]The linker-referencing sections include *.(gnu).hash, .dynsym, .dynstr, .gnu.version, .gnu.version_r, .rela.dyn, .rela.plt.*, etc.

.text do not have this problem, because whenever an address is calculated that happens to point at *.rodata* section, NDisassembler will mark it as a valid reference regardless of whether a corresponding memory load instruction is detected or not.

- **References to *.eh_frame_hdr* and *.eh_frame***: These sections provide auxiliary information such as the address range of functions, the stack content when a C++ exception is triggered, etc. The previous components are unable to update them because they are used neither by the converted module itself nor by the dynamic linker. Instead, we found that C++ runtime and debuggers such as gdb would reference and read into these two sections for exception handling or stack unwinding.

NMonitor dynamically handles both categories of unadjusted references. NMonitor responds to memory violations caused by any attempted read access to XOM. It checks the context and the data being accessed. If the context matches the two cases discussed above and the address being accessed does belong to the relocated data, NMonitor permits and facilitates the access; otherwise, it terminates the program. Specifically, NMonitor whitelists these two kinds of data and ensures legitimate accesses to them can go through while potential abuses by attackers cannot. For instance, NMonitor only allows C++ runtime module to access the *.eh_frame* sections (updatable through sysctl). For the *.text* embedded data, NMonitor only allows code from the over-approximated hosting function to read them. Note that while this design helps our system cope with those corner cases, the security of our system is barely undermined for two reasons: (*i*) the majority of the whitelisted data are indeed *real* data, which are not even decodable or surrounded by non-decodable data. (*ii*) Different data require code from different regions to access them; attackers cannot simply exploit one memory leak bug to read across all these embedded data.

4.3 Implementation

Norax is fully implemented based on two commercial mobile phones, Samsung Galaxy S6 and LG Nexus 5X. In this section, we present the implementation of Norax on LG Nexus 5X, which is equipped with Qualcomm Snapdragon 808 MSM8992 (4 x ARM Cortex-A53 & 2 x ARM Cortex-A57) and 2GB RAM. The phone is running Android OS v6.0.1 (Marshmallow) with Linux kernel v3.14 (64-bit).

4.3.1 Kernel Modification

We modified several OS subsystems in order to implement the design discussed in § 4.2. To start off, the memory management (MM) subsystem is modified to enable the execute-only memory configuration and securely handle the legitimate page fault triggered by data abort on reading the execute-only memory. Specifically, we intercept the page fault handler, the *do_page_fault()* function, to implement the design of *NMonitor*. Implementing the semantics for all kinds of memory load instructions is error-prone and requires non-trivial engineering effort, but there is one additional caveat, as page fault is one of the most versatile events in Linux kernel that has very diversified usages, such as copy-on-write (COW), demand paging and memory page swappings etc. Also, accessing the same virtual address could fault multiple times (e.g., First triggered by demand paging, and then by XOM access violation). If not carefully examined, irrelevant page fault events could be mistakenly treated as XOM-related ones, which may cause the entire system to be unstable or even crash. To precisely pinpoint the related page fault events, we devise a series of constraints to filter the irrelevant ones. when a page fault happens, the following checks are performed:

- Check if the faulting process contains Norax converted module, this is indicated by a flag set by *NLoader* when loading a converted binary. This flag will be propagated when the process *forks* a new child, and properly removed if the new child does an *exec* to run a new program.

- Check the exception syndrome register on exception level one (ESR_EL1[7]) for two fields: (*i*) Exception class and (*ii*) Data fault status code. This ensures the fault is triggered by the user space program, and it faults on the last level page table entry (we only enforce XOM at pte entries) because of permission violation.

- Check the VMA permission flags and only handle the case of reading an execute-only page. All these restrictions together ensure that we do not mistake other page fault events with ours.

To verify the integrity of a violation triggered by XOM, we extend the *task_struct* to maintain a list of access policies, one for each module. We also instrument the *set_pte* function to ensure the permission of a page must follow the $\mathbf{R} \oplus \mathbf{X}$ policy. This way, we prevent the attacker from tricking the OS to remap the execute-only memory through high-level interfaces. The modified kernel subsystems also include the file system (FS) and system calls where we instrument the executable loader and implement the design of *NLoader* plugins respectively.

4.3.2 Bionic Linker Modification

In a running program, all the libraries needed by the executable are loaded by the linker. In order to handle those converted libraries and make the code regions of the whole process execute-only, we directly modify the linker's source code to place hooks before the library loading and symbol resolution routines as described in § 4.2.3. One quirk of the Bionic linker is that when loading libraries, it places those modules right next to each other, leaving no space in-between. This causes problems from multiple perspectives. Firstly, it lowers the entropy of the address space randomness thus undermines the effectiveness of ASLR. Secondly, it also "squeezes" out the space for *NLoader* to load the Norax-related metadata. To resolve this issue, *NPatcher* encodes the size of the total metadata into the Norax header when it recomposes the binary, and we instrument the linker such that when it is loading a library it will leave a gap with the size of the sum of the encoded number (zero for the unconverted binaries) and a randomly generated nuance.

4.4 Evaluation

We evaluate three aspects of Norax: (i) whether it breaks the functioning of patched binaries? (ii) how accurate is its data analysis? and (iii) how much overhead it incurs?

Functioning of Transformed Binaries: For this test, we selected 20 core system binaries to transform, including both programs and libraries (Table 5). These binaries provide support for basic functionalities of an Android phone, such as making a phone call, installing apps, and playing videos. We obtain these binaries from a Nexus 5X phone that runs Android OS v6.0.1 (Marshmallow). These stock binaries are compiled with compiler optimization and without debugging metadata.

We tested the functionality of the transformed binaries using our own test cases as well as the Android Compatibility Test Suite (CTS) [1]. We modified the system bootstrapping scripts (**.rc* files), which direct Android to load the system binaries patched by Norax. Table 3 shows the specific tests we designed for each system executable and library. For example, *surfaceflinger* is the UI composer, which depends on two libraries: *libmedia.so* and *libstagefright.so*. Zygote (*app_process64*) is the template process from which all App processes are forked. It uses all of the patched binaries. While running our functionality tests, we observed an attempt by the linker to read the ELF header, which

is located in the pages marked executable-only. While this attempt was allowed and facilitated by NMonitor, our system can be optimized to handle this case during the patching stage instead.

We also ran the Android Compatibility Test Suite (CTS) on a system where our transformed binaries are installed. The suite contains around 127,000 test packages, and is mandatory test performed by OEM vendors to assess the compatibility of their modified Android systems. The test results are shown in Table 4. NORAX did not introduce any additional failure than those generated by the vendor customization on the testing devices. The results from both tests show that the functioning of patched binaries is not interrupted or broken by NORAX.

Correctness of Data Analysis: To thoroughly test the correctness of our embedded data identification algorithm, we ran the data analysis module of NDisassembler against a large test set consisting of all 313 Android system binaries, whose sizes span from 5.6KB (*libjnigraphics.so*) to 16.5MB (*liblog.so*), totaling 102MB. For these binaries, we compare the data identified by *NDisassembler* with the real embedded data. Our ground truth is obtained by compiling debugging sections (.debug_*)[6] into the binaries. We use an automatic script to collect bytes in file offsets that fall outside any function range and compare them with the analysis results from *NDisassembler*. For the bytes that are not used by any of the functions, we found that some of them are *NOP* instructions used purely for the padding purpose; whilst some are just "easter eggs", for instance, in the function *gcm_ghash_v8* of *libcrypto.so*, the developers left a string *"GHASH for ARMv8, CRYPTOGAMS by <appro@openssl.org>"*. These kinds of data were not collected by NORAX. Since there are no references to them, making them non-readable will not break any function.

For the tested binaries, *NDisassembler* correctly identified all the embedded data. Only for 28 out of the 313 binaries did NDisassembler reported false positives (i.e., code mistakenly identified as embedded data), due to the over-approximate approach we use. These rare false positive cases are expected by our design and are handled by NMonitor during runtime. Table 6 shows a subset of the results[4].

Table 3: Rewritten program functionality tests.

Module	Description	Experiment	Success
vold	Volume daemon	mount SDCard; umount	Yes
toybox	***115*** **nix utilities*	try all commands	Yes
toolbox	***22*** *core *nix utilities*	try all commands	Yes
dhcpcd	DHCP daemon	obtain dynamic IP address	Yes
logd	Logging daemon	collect system log for 1 hour	Yes
installd	APK install daemon	install 10 APKs	Yes
app_process64 (zygote)	Parent process for all applications	open 20 apps; close	Yes
qseecomd	Qualcomm's proprietary driver	boot up the phone	Yes
surfaceflinger	Compositing frame buffers for disply	Take 5 photos; play 30 min movie	Yes
rild	Baseband service daemon	Have 10 min phone call	Yes

Table 4: System compatibility evaluation, the converted zygote, qseecomd, installd, rild, logd, surfaceflinger, libc++, libstagefright are selected randomly to participate the test to see whether they can run transparently with other unmodified system components.

	Pass	Fail	Not Executed	Plan Name
CTS normal	126,457	552	0	CTS
CTS NORAX	126,457	552	0	CTS

Size Overhead: In our functionality test, the sizes of our selected binaries range from ≈14K to ≈7M, as shown in Table 5. After transformation, the binary sizes increased by an average of 3.91%. Note that *libm.so* is a outlier, as its file size increased much more than others. After manual inspection, we found that this math library has a lot of constant values hardcoded in various mathematical functions such as *casinh*(), *cacos*(). As an optimization, the compiler embeds this large set of constant data into the code section to fully exploit spatial locality, which translates to more metadata generated by NORAX during the patching stage.

Performance Overhead: We used Unixbench[85] to measure the performance of our system. The benchmark consists of two types of testing programs: (*i*) User-level CPU-bound programs; (*ii*) System benchmark programs that evaluate I/O, process creation, and system calls, etc. We ran the benchmark on both the stock and patched binaries, repeating three times in each round. We then derived the average runtime and space overhead, which are given in Figure 12.

For the runtime overhead, the average slowdown introduced by NORAX is 1.18%. The overhead mainly comes from the system benchmark programs, among which *Execl* shows the maximum slowdown. Investigating its source code, we found that this benchmark program keeps invoking the *exec* system call from the same process to execute itself over and over again, thus causing NLoader to repeatedly prepare new book-keeping structures and destroy old ones (§ 4.2.3). This process, in turn, leads to multiple locking and unlocking operations, hence the relatively higher overhead. Fortunately, we do not find this behavior common in normal programs. In addition, some simple optimizations are possible: (*i*) employing a more fine-grained locking mechanism; (*ii*) reusing the book-keeping structures when *exec* loads the same image.

Security Impact: Since NORAX duplicates the *embedded data* in code sections, they are in theory still reusable by adversaries. we conduct a gadget searching experiment in the duplicated *embedded data* appended at the end of the converted binaries. Table 6 shows the number of available gadgets we found in those data. As the result shows, available gadgets are actually very rare even in the binaries that have a lot of *embedded data* such as *libm.so*, we believe this is because the majority of those duplicated bytes are by themselves not decodable. Also note that the shown numbers are upper bounds of the available gadgets. Because, in the executable code section, where the original *embedded data* reside, the bytes that form the gadgets may not be placed next to each other.

5 Towards Bug-Driven and Verification-Based Hybrid Fuzzing

Defenses combating in-process memory abuse should be muti-dimensional, since there is no existing approaches provide guaranteed protection against all variants of in-process memory abuse. SHREDS [31] provides isolation with better granularity, but each shred still subjects to direct exploitation. NORAX [32] and existing fine-grained ASLR [69] drastically raise the bar of code-reuse attacks, but data-only attacks [64] still remains unsolved. Hence, to develop comprehensive defense, another important perspective is to stand in the shoes of an attacker. In this

[4]This subset was chosen to be consistent with the binaries used in the other tests in this section. The complete set of all 313 Android system binaries, which can be easily obtained, are not shown here due to the space limit.

Table 5: Binary transformation correctness test.

Module	Size (Stock)	Size (NORAX)	File Size Overhead	# of Rewrite Errors
vold	486,032	512,736	5.49%	0
toybox	310,800	322,888	3.89%	0
toolbox	148,184	154,632	4.35%	0
dhcpcd	112,736	116,120	3.00%	0
logd	83,904	86,256	2.80%	0
installd	72,152	76,896	6.58%	0
app_process64 (zygote)	22,456	23,016	2.49%	0
qseecomd	14,584	15,032	3.07%	0
surfaceflinger	14,208	14,448	1.69%	0
rild	14,216	14,784	4.00%	0
libart.so	7,512,272	7,772,520	3.46%	0
libstagefright.so	1,883,288	1,946,328	3.35%	0
libcrypto.so	1,137,280	1,157,816	1.81%	0
libmedia.so	1,058,616	1,071,712	1.24%	0
libc.so	1,032,392	1,051,312	1.83%	0
libc++.so	944,056	951,632	0.80%	0
libsqlite.so	791,176	805,784	1.85%	0
libbinder.so	325,416	327,072	0.51%	0
libm.so	235,544	293,744	24.71%	0
libandroid.so	96,032	97,208	1.22%	0
AVG.			3.91%	0

Table 6: Embedded data identification correctness, empirical experiment shows our analysis works well in AArch64 COTS ELFs, with **zero** false negative rate and very low false positive rate in terms of finding embedded data. The last column shows the negligible number of left-over gadgets in the duplicated embedded data set.

Module	#. of Real Inline Data (Byte)	#. of Inline Data Flagged by Norax (Byte)	#. of Gadgets found in extracted inline Data
vold	0	0	0
toybox	8	8	0
toolbox	20	20	0
dhcpcd	40	40	4
Logd	0	0	0
installd	0	0	0
app_process64 (zygote)	0	0	0
qseecomd	N/A	0	0
surfaceflinger	0	0	0
rild	0	0	0
libart.so	17716	17716	8
libstagefright.so	296	296	5
libcrypto.so	2472	2512	25
libmedia.so	3936	3936	0
libc.so	4836	4836	5
libc++.so	12	12	0
libsqlite.so	932	1004	13
libbinder.so	0	0	0
libm.so	20283	20291	48
libandroid.so	0	0	0
Total	50551	50671	108

section, I propose a novel automated bug-finding system aiming to decrease the number of low-hanging bugs that lead to further attacks such as code-reuse attack, or more generally, in-process memory abuse. As a result, raising the bar of in-process memory abuse.

5.1 Overview

Facilitated by efforts from both the academia and the industry, software testing techniques gain remarkable advancement in finding security vulnerabilities. In particular, people have deeply exploited fuzz testing [105, 122] and concolic execution [94, 76] to disclose a great amount of vulnerabilities every year. Nevertheless, the evolution of fuzz testing and concolic execution is gradually impeded by their intrinsic limitations. On one hand, fuzz testing quickly covers different code regions but hardly reaches compartments that are guarded by complex conditions. On the other hand, concolic execution excels at solving conditions and enumerating execution paths while it frequently sinks the exploration into code segments that contain a large volume of execution paths (*e.g.,* loop). Due to these limitations, using fuzz testing or concolic execution alone often ends with a substantial amount of uncovered code within the given time frame. To enlarge code coverage, recent researches explore the idea of hybrid testing which assembles both fuzz testing and concolic execution [121, 101, 65].

The insight of hybrid testing is to apply fuzz testing on the path exploration and leverage concolic execution to resolve complex conditions. A hybrid approach relies on fuzz testing to exhaustively explore code regions. Once fuzzing stops making progress, the hybrid approach switches to concolic execution which replays the seeds accumulated in fuzzing. While following the path pertaining to a given seed, concolic execution examines at each of the conditional statements to check whether the other branch remains uncovered. If so, it solves the constraints leading to the new branch. This enables concolic execution to generate a new seed which guides fuzz testing to deeper code space exploration.

As demonstrated in recent works [101, 121], hybrid testing creates new opportunities towards high code coverage. However, it still adopts the code-driven philosophy, which unfortunately falls short of the needs of vulnerability mining. First, hybrid testing treats all the branches indiscriminately. However, most of these branches might lead to code lack of vulnerabilities and the exploration on them involves expensive computation (*e.g.,* constraint solving and extra fuzzing). Consequently, hybrid testing often exhausts the assigned CPU cycles way before it could resolve the branches guarding vulnerabilities. Second, hybrid testing can fail to catch a vulnerability even if it reaches the vulnerable code region following a correct path. This is mainly because hybrid testing exercises new branch in the manner of random execution, which oftentimes has low chance to satisfy the subtle conditions to trigger the vulnerability.

In this section, I propose Savior (abbreviation for **S**peedy-**A**utomatic-**V**ulnerability-**I**ncentivized-**OR**acle), a hybrid testing tool that evolves towards being bug-driven and verification-based. To fulfill this goal, Savior incorporates two novel techniques as follows.

Bug-driven Prioritization: Instead of solving the unreached branches in an indiscriminate manner, Savior prioritizes those that have higher promise in leading to vulnerable code. Specifically, prior to the testing, Savior labels all the potential sites with vulnerability in the target program. As there lacks theories to obtain the ground truth of vulnerabilities, Savior follows the extant schemes [3, 46] to conservatively capture all the possibilities. In addition, Savior statically computes the code regions dominated by each branch. At the time of testing, Savior prioritizes concolic execution on branches that dominate code with more vulnerability labels. Since such branches guard higher volumes of vulnerabilities, advancing the schedule to reach them can significantly expedite the discovery of vulnerabilities.

Bug-guided Search: Going beyond accelerating vulnerability finding, Savior also comprehensively seeks for vulnerabilities along the paths explored by fuzzing. While Savior executes a seed with concolic execution, it revisits each of the vulnerability label that remain unconfirmed on the execution path. That is, Savior constructs the constraints to trigger such a vulnerability following the current path. If the synthesized constraints are satisfiable, Savior generates a test case as a proof of this vulnerability. Otherwise Savior verifies that the vulnerability is un-triggerable by arbitrary inputs going through this path.

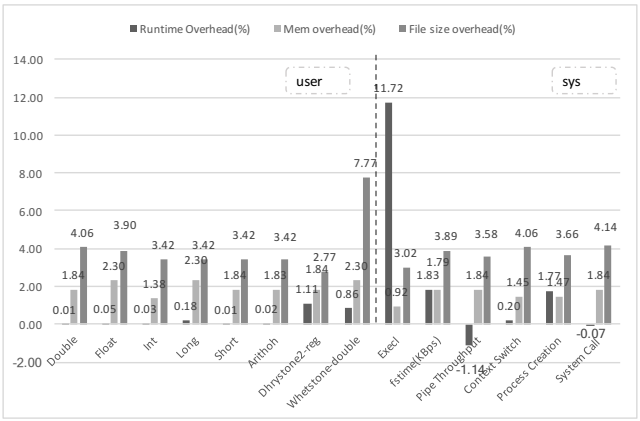


Figure 12: Unixbench performance overhead for unixbench binaries, including runtime, peak resident memory and file size overhead (left: user tests, right: system tests)

5.2 Methodology

Savior's hybrid testing demands the static information of vulnerability labels and the domination relations. We gather the required information via static analysis on the LLVM IR of the target program.

Vulnerability Labeling: Currently, Savior leverages existing tools to provide vulnerability information. We have summarized that, In general, any tool satisfying the following criteria can plug-and-play.

- As Savior loads the labels prior to testing, the tool should be able to work statically on the code.
- The tool should conduct sound analysis. In other words, we expect a complete set of labels which should mark all the potential vulnerabilities of interests. An unsound tool may miss a lot of true labels, which affects Savior in two ways. First, the weights of many non-covered branches are falsely lowered since the labels in their dominating code get reduced. This delays Savior to solve them and the defers the identification of vulnerabilities guarded by them. Second, Savior will lose guidance to those true labels and miss the opportunities to catch them during vulnerability verification. We understand that a such tool often produces many false postives. This may also reduce the effectiveness of Savior in prioritization. So we should employ additional analysis that can help reduce such false positives.
- An applicable tool has to summarize the triggering conditions once it labels a vulnerability. The reason is that Savior relies on such conditions as guidance in vulnerability verification. Going beyond that, we only accept conditions that have data dependency with objects in the target program (*e.g.,* conditions that checks status of variables in the program). Otherwise our concolic execution will not correlate the program execution with the vulnerability conditions and hence, unable to verify the existence of bug. For instance, the widely used AddressSanitizer [97] verifies the vulnerability based on the status of its own red-zone, which are not applicable to Savior.
- Engineering wise, we expect the tool to be compatible with LLVM IR, since Savior runs with LLVM IR. In addition, the tool needs to either marks the labels it plants or exports related information. This is mainly because Savior needs such information for prioritizing non-covered branches and vulnerability verification.

In our current design, Savior employs LLVM UBSan [3], which conservatively capture a wide spectrum of undefined behaviors. By default, Savior only enables certain components in UBSan to labels issues that have explicit security consequences, including signed/unsigned integer overflow, floating cast overflow, pointer arithmetic overflow and out-of-bounds array indexing. In addition, we patch the Clang front-end to attach UBSan labels to the generated IR. The vulnerability conditions created by UBSan mostly follow the comprehensive summaries in [113] and we omit their details.

Domination Analysis: This analysis proceeds with two phases which first construct the inter-procedure control flow graph (CFG) and then build the pre-domination tree on the CFG. We present their specifics in the following.

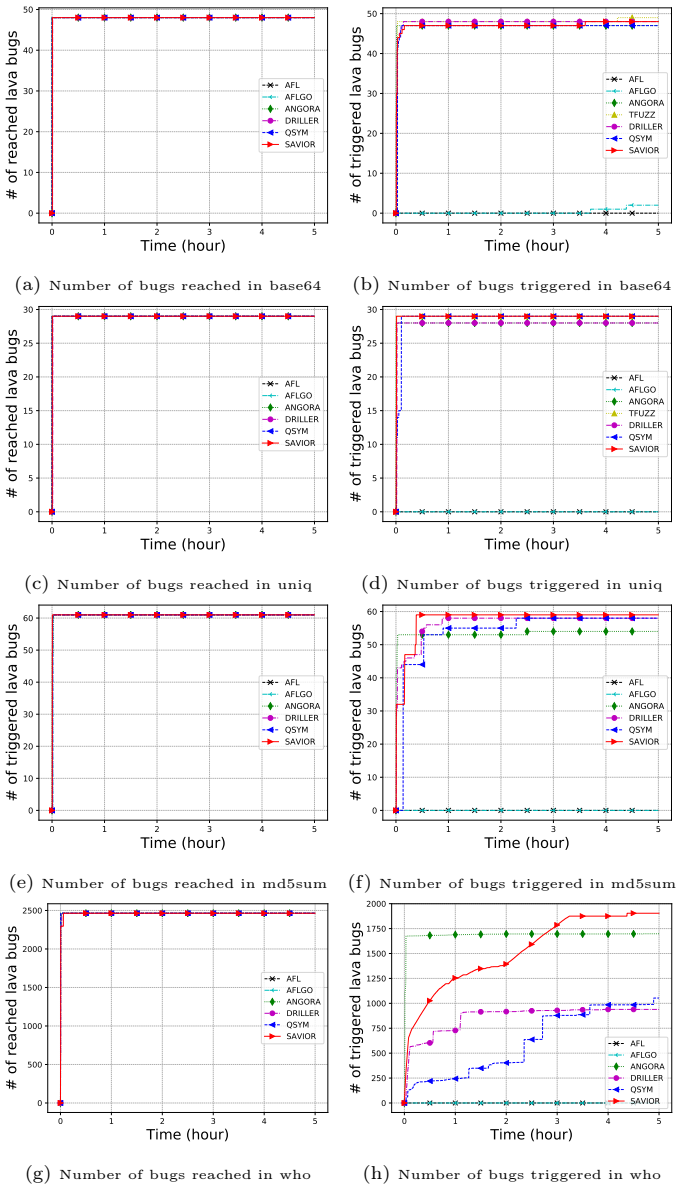
To generate the inter-procedure CFG, we use an algorithm similar to the one implemented in SVF [104]. The algorithm starts with building intra-procedure CFGs for individual functions and then connect them by the caller-callee relation. To resolve indirect calls, it iteratively performs point-to analysis and expand the targets of indirect calls. In Savior, we utilize the Andersen's algorithm in point-to analysis for better efficiency. Note that our design is fully compatible with other point-to analyses that have higher precision (*e.g.,* flow-sensitive analysis [60]).

Taking the above inter-procedure CFG as input, we run the algorithm proposed in [36] to construct the domination tree. This enables us to understand the code regions dominated by each of the basic blocks and count the number vulnerability labels enclosed. For the convenience of access at the time of testing, we encode the dominated bug number (metadata in LLVM IR) at the beginning of each basic block.

5.3 Preliminary Results and Analysis

5.3.1 Evaluation with LAVA-M

Experimental Setup In this evaluation, we run each of the fuzzers in Table 7 with four LAVA-M [47] programs and we use the seeds shipped with the benchmark. For consistency, we conduct all the experiments on r4.16xlarge amazon EC2 instances with Ubuntu 16.04 LTS and we sequentially run all the experiments to avoid interference. In addition, we affiliate each fuzzer with 3 free CPU cores to ensure fairness with computation resources. Following the previous works [121, 29, 90], we run each test for 5 hours. To minimize the effect of randomness in fuzzing, we repeat each test 5 times and report the average results. The settings specific to each fuzzer is summarized in Table 7, including how we distribute the 3 CPU cores and the actions we take to accommodate those fuzzers. In LAVA-M, each artificial vulnerability is enclosed in a



(a) Number of bugs reached in base64 (b) Number of bugs triggered in base64

(c) Number of bugs reached in uniq (d) Number of bugs triggered in uniq

(e) Number of bugs reached in md5sum (f) Number of bugs triggered in md5sum

(g) Number of bugs reached in who (h) Number of bugs triggered in who

Figure 13: Evaluation results with LAVA-M. The left column shows the number of reached lava-bugs by those fuzzers while the right column shows the number of lava-bug triggered by different fuzzers. For TFuzz, we only present the number of triggered bugs in `base64` and `uniq`, as the other resutls are not reliable due to defects in a third-party component. This has been confirmed with the developers of TFuzz.

function call to `lava_get` (in-lined in our evaluation). We use these calls as the targets to guide AFLGo and we mark them as vulnerability labels to enable bug-driven prioritization in Savior. In addition, as the vulnerability condition is hard-coded in the `lava_get` function, we naturally have support for bug-guided search. Finally, for Angora, we adopt the patches as suggested by the developers [9].

Evaluation Results In the left column of Figure 13, we show how many vulnerabilities are reached over time by different fuzzers. The results demonstrate that all the fuzzers can instantly cover the code with LAVA vulnerabilities. However, as presented in the right column of Figure 13, TFuzz, Angora, Driller, QSYM, and Savior are able to trigger most (or all) of the vulnerabilities while AFL and AFLGo can trigger few. The reason behind is that the triggering conditions of LAVA vulnerabilities are all in the form of 32-bit magic number matching. Mutation-based fuzzers, including AFL and AFLGo, can hardly satisfy those conditions while the other fuzzers are all featured with techniques to solve them.

Despite TFuzz, Angora, Driller, QSYM, and Savior all trigger large numbers of LAVA vulnerabilities, they differ in terms of comprehensiveness and efficiency. TFuzz quickly covers the listed vulnerabilities in `base64` and `uniq`. This is attributable to that (1) TFuzz can reach all the vulnerabilities with the initial several seeds and (2) TFuzz can transform the program to immediately trigger the encountered vulnerabilities. Note that we do not show the results of TFuzz on `md5sum` and `who`, because TFuzz gets interrupted by a defective third-party component[5]. For all the cases, Angora triggers most vulnerabilities immediately after its start. The main reason is that the "black-box function" pertaining to all LAVA vulnerabilities is `f(x) = x` and the triggering conditions are like `f(x) == CONSTANT`. Angora always starts evaluates such functions with `x = CONSTANT` and therefore, it can instantly generate seeds that satisfy the vulnerability conditions. In the case of `who`, Angora does not make all the vulnerabilities because of the incomplete dynamic taint analysis.

Regarding the three hybrid tools, they can trigger every vulnerability that their concolic executor encounters. In the cases of `base64`, `uniq`, and `md5sum`, their concolic executor can reach all the vulnerabilities with (arbitrary) initial seeds, which explains why the fuzzers all

[5]This has been confirmed with the developers.

Table 7: Fuzzer specific settings in evaluation with Lava-M.

Fuzzers	*Settings*		
	Source	Instances	Note
AFL	[10]	1 AFL master; 2 AFL slaves	N/A
AFLGo	[5]	1 AFLGo master; 2 AFLGo slaves	Use inlined `lava_get` as target locations of guided fuzzing
TFuzz	[6]	3 AFL jobs (adjust default argument to Fuzzer)	Use the docker environment prepared at [6] for evaluation
Angora	[11]	3 Angora threads (with option `"-j 3"`)	Patch Lava to support Angora, as suggested by the developers [12]
Driller	Self-developed	1 concolic executor; 1 AFL master; 1 AFL slave	Followed the native `Driller` in scheduling concolic execution [7]
QSYM	[8]	1 concolic executor; 1 AFL master; 1 AFL slave	N/A
Savior	Self-developed	1 concolic executor; 1 AFL master; 1 AFL slave	Use in-lined `lava_get` as lables of vulnerabilities

quickly trigger the listed vulnerabilities (regardless of their seed scheduling). But in the case of `who`, even though the fuzzing component quickly generate seeds to cover code containing vulnerabilities, it takes the concolic executor much longer to run those seeds. For instance, while executing the inputs from AFL, QSYM needs around 72 hours of continuous concolic execution to reach all the LAVA vulnerabilities in `who`. It's worth noting that Driller (with a random seed scheduling) moves faster than QSYM, because QSYM prioritizes concolic execution on small seeds, while reaching many vulnerabilities in `who` needs larger seeds.

Table 8: Bug triggered by different fuzzers in evaluation with Lava-M (with bug-guided search).

Fuzzers	*Lava-M Benchmark*			
	base64	uniq	md5sum	who
AFL	48 (100%)	29 (100%)	59 (100%)	2357 (96.3%)
AFLGo	48 (100%)	29 (100%)	59 (100%)	2357 (96.3%)
TFuzz	47* (100%)	29* (100%)	N/A	N/A
Angora	48 (100%)	29 (100%)	59 (100%)	2357 (96.3%)
Driller	48 (100%)	29 (100%)	59 (100%)	2357 (96.3%)
QSYM	48 (100%)	29 (100%)	59 (100%)	2357 (96.3%)
SAVIOR	48 (100%)	29 (100%)	59 (100%)	2357 (96.3%)
Listed	**44**	**28**	**57**	**2136**

Table 9: IDs of Lava bugs that are not listed but identified by bug-guided search.

Program	*Bugs unlisted by LAVA-M but exposed by bug-guided verification*
base64	274, 521, 526, 527
uniq	227
md5sum	281, 287
who	1007, 1026, 1034, 1038, 1049, 1054, 1071, 1072, 117, 12 125, 1329, 1334, 1339, 1345, 1350, 1355, 1361, 1377, 1382 1388, 1393, 1397, 1403, 1408, 1415, 1420, 1429, 1436, 1445 1450, 1456, 1461, 16, 165, 169, 1718, 1727, 1728, 173 1735, 1736, 1737, 1738, 1747, 1748, 1755, 1756, 177, 181 185, 189, 1891, 1892, 1893, 1894, 1903, 1904, 1911, 1912 1921, 1925, 193, 1935, 1936, 1943, 1944, 1949, 1953, 197 1993, 1995, 1996, 2, 20, 2000, 2004, 2008, 2012, 2014 2019, 2023, 2027, 2031, 2034, 2035, 2039, 2043, 2047, 2051 2055, 2061, 2065, 2069, 2073, 2077, 2079, 2081, 2083, 210 214, 2147, 218, 2181, 2189, 2194, 2198, 2219, 222, 2221 2222, 2223, 2225, 2229, 2231, 2235, 2236, 2240, 2244, 2246 2247, 2249, 2253, 2255, 2258, 226, 2262, 2266, 2268, 2269 2271, 2275, 2282, 2286, 2291, 2295, 2302, 2304, 24, 2462 2463, 2464, 2465, 2466, 2467, 2468, 2469, 2499, 2500, 2507 2508, 2521, 2522, 2529, 2681, 2682, 2703, 2704, 2723, 2724 2742, 2796, 2804, 2806, 2814, 2818, 2823, 2827, 2834, 2838 2843, 2847, 2854, 2856, 2919, 2920, 2921, 2922, 294, 2974 2975, 298, 2982, 2983, 2994, 2995, 3002, 3003, 3013, 3021 303, 307, 3082, 3083, 3099, 312, 316, 3189, 3190, 3191 3192, 3198, 3202, 3209, 321, 3213, 3218, 3222, 3237, 3238 3239, 3242, 3245, 3247, 3249, 325, 3252, 3256, 3257, 3260 3264, 3265, 3267, 3269, 327, 334, 336, 338, 3389, 3439 346, 3466, 3468, 3469, 3470, 3471, 3487, 3488, 3495, 3496 350, 3509, 3510, 3517, 3518, 3523, 3527, 355, 359, 3939 4, 4024, 4025, 4026, 4027, 4222, 4223, 4224, 4225, 4287 4295, 450, 454, 459, 463, 468, 472, 477, 481, 483 488, 492, 497, 501, 504, 506, 512, 514, 522, 526 531, 535, 55, 57, 59, 6, 61, 63, 73, 77 8, 81, 85, 89, 974, 975, 994, 995, 996

As mentioned before, all the fuzzers have generated seeds to reach all the vulnerabilities, which they cannot trigger all of them either because their incapability to satisfy the conditions or because they did not finish concolic execution on all the seeds. We extended a further experiment by performing bug-guided searching with the seeds from all the fuzzers. Simply speaking, we run each of the seeds from all the fuzzers with the concolic executor in Savior. In this experiment, we only do constraint solving when a LAVA-M vulnerability condition is encountered. This simulates the process of bug-guided search for all fuzzers. Enhanced with our bug-guided search, all the fuzzers can trigger not only the listed bugs in LAVA, as shown in Table 8, but also a group of unlisted bugs (Table 9).

6 Milestones

The plan for completing my research is presented in Table 10.

Table 10: Plan for completion

Task	**Completion Date**
Complete Implementation of Savior	March 2019
Evaluate Savior on large real-world programs	April 2019
Finalizing Savior for publication	June 2019
Explore Summarization Rules for UaF bugs	July 2019
Dissertation defense	September 2019

References

[1] Android compatibility test suite. https://source.android.com/compatibility/cts/index.html.

[2] Arm 32-bit sandbox. https://developer.chrome.com/native-client/reference/sandbox_internals/arm-32-bit-sandbox.

[3] Clang 8 documentation - undefined behavior sanitizer. http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks. (Accessed on 08/22/2018).

[4] clang: a c language family frontend for llvm. http://clang.llvm.org/.

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

[6] Dwarf standards. http://www.dwarfstd.org.

[7] Exception syndrome register(esr) interpretation. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500e/CIHFICFI.html.

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

[9] https://github.com/AngoraFuzzer/Angora/blob/master/docs/lava.md, 2019.

[10] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353. ACM, 2005.

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

[12] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries.

[13] ANSI/ISO. Iso/iec 9899:2018. https://www.iso.org/standard/74528.html, 2018.

[14] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*, 2011.

[15] Ahmed M Azab, Peng Ning, and Xiaolan 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*, pages 375–388. ACM, 2011.

[16] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1342–1353. ACM, 2014.

[17] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos 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)*, pages 335–348, 2012.

[18] Sandeep Bhatkar, Daniel C DuVarney, and R Sekar. Efficient techniques for comprehensive protection from memory error exploits. In *Usenix Security*, 2005.

[19] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely rerandomization for mitigating memory disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 268–279. ACM, 2015.

[20] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242. IEEE, 2014.

[21] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *NSDI*, volume 8, pages 309–322, 2008.

[22] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 30–40. ACM, 2011.

[23] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. Leakage-resilient layout randomization for mobile devices. In *Proceedings of the 2016 Network and Distributed System Security (NDSS) Symposium*, 2016.

[24] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, pages 57–72, 2004.

[25] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering, 15-19 September 2008, L'Aquila, Italy*, pages 443–446, 2008.

[26] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, 2015.

[27] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58. ACM, 2009.

[28] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 380–394, 2012.

[29] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. *arXiv preprint arXiv:1803.01307*, 2018.

[30] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 2–13. ACM, 2008.

[31] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71. IEEE, 2016.

[32] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. Norax: Enabling execute-only memory for cots binaries on aarch64. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 304–319. IEEE, 2017.

[33] Yue Chen, Zhi Wang, David Whalley, and Long Lu. Remix: On-demand live randomization. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 50–61. ACM, 2016.

[34] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–278. ACM, 2011.

[35] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. In *IEEE International Workshop on Program Comprehension*, 1999.

[36] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.

[37] Jonathan Corbet. Memory protection keys. https://lwn.net/Articles/643797/, May 2015.

[38] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *2015 IEEE Symposium on Security and Privacy*, pages 763–780. IEEE, 2015.

[39] Stephen J Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 243–255. ACM, 2015.

[40] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[41] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, 2014.

[42] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 299–310. ACM, 2013.

[43] Drew Davidson, Yaohui Chen, Franklin George, Long Lu, and Somesh Jha. Secure integration of web content and applications on commodity mobile operating systems. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 652–665. ACM, 2017.

[44] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[45] Leonardo Mendonça de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. volume 4590 of *Lecture Notes in Computer Science*, pages 20–36, 2007.

[46] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding integer overflow in c/c++. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE'12, pages 760–770, 2012.

[47] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[48] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. Xfi: Software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88. USENIX Association, 2006.

[49] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913. ACM, 2015.

[50] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*, pages 293–306. Boston, MA, 2008.

[51] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 660–677.

[52] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. volume 4590 of *Lecture Notes in Computer Science*, pages 519–531, 2007.

[53] Jason Gionta, William Enck, and Peng Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 325–336. ACM, 2015.

[54] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[55] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[56] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589. IEEE, 2014.

[57] Le Guan, Jingqiang Lin, Bo Luo, and Jiwu Jing. Copker: Computing with private keys without ram. In *21st ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.

[58] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 3–19, May 2015.

[59] Dave Hansen. [rfc] x86: Memory protection keys. https://lwn.net/Articles/643617/, May 2015.

[60] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 289–298. IEEE Computer Society, 2011.

[61] Keith Harrison and Shouhuai Xu. Protecting cryptographic keys from memory disclosure attacks. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 137–143. IEEE, 2007.

[62] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. Ilr: Where'd my gadgets go? In *2012 IEEE Symposium on Security and Privacy*, pages 571–585. IEEE, 2012.

[63] William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Trans. Software Eng.*, 3(4):266–278, 1977.

[64] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[65] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. jfuzz: A concolic whitebox fuzzer for java. In *First NASA Formal Methods Symposium - NFM 2009, Moffett Field, California, USA, April 6-8, 2009.*, pages 121–125, 2009.

[66] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *ACSAC*, volume 6, pages 339–348, 2006.

[67] Douglas Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284, 2003.

[68] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[69] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 461–477. IEEE, 2018.

[70] Abraham Peedikayil Kuruvila, Ioannis Zografopoulos, Kanad Basu, and Charalambos Konstantinou. Hardware-assisted detection of firmware attacks in inverter-based cyberphysical microgrids. *International Journal of Electrical Power & Energy Systems*, 132:107150, 2021. Publisher: Elsevier.

[71] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[72] Changming Liu, Yaohui Chen, and Long Lu. Kubo: Precise and scalable detection of user-triggerable undefined behavior bugs in os kernel. In *NDSS*, 2021.

[73] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, 2015.

[74] Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. Boomerang: Exploiting the semantic gap in trusted execution environments. In *Proceedings of the 2017 Network and Distributed System Security Symposium (NDSS)*, 2017.

[75] Federico Maggi, Marcello Pogliani, and P Milano. Attacks on Smart Manufacturing Systems. *Trend Micro Research: Shibuya, Japan*, pages 1–60, 2020.

[76] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 416–426. IEEE, 2007.

[77] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 941–951. ACM, 2015.

[78] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 143–158. IEEE, 2010.

[79] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for tcb minimization. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 315–328. ACM, 2008.

[80] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday 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*, pages 1–1. ACM, 2013.

[81] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015(S 91), 2015.

[82] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque control-flow integrity. In *NDSS*, 2015.

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

[84] Tilo Müller, Felix C Freiling, and Andreas Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, pages 17–17, 2011.

[85] DC Niemi. Unixbench 4.1. 0.

[86] Ben Niu and Gang Tan. Rockjit: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1317–1328. ACM, 2014.

[87] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 914–926. ACM, 2015.

[88] Brian S Pak. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *School of Computer Science Carnegie Mellon University*, 2012.

[89] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[90] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[91] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *USENIX Security*, volume 3, 2003.

[92] Rui Qiao and R. Sekar. Function interface analysis: A principled approach for function recognition in COTS binaries. In *The 47th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2017.

[93] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[94] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572. ACM, 2007.

[95] Koushik Sen and Gul Agha. CUTE and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification, 18th International Conference, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 419–423, 2006.

[96] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272, 2005.

[97] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[98] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *NEUZZ: Efficient Fuzzing with Neural Program Smoothing*, page 0. IEEE, 2018.

[99] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[100] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[101] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[102] Raoul Strackx and Frank 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*, pages 2–13. ACM, 2012.

[103] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 339. IEEE Computer Society, 2003.

[104] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 265–266. ACM, 2016.

[105] Ari Takanen, Jared D Demott, and Charles Miller. *Fuzzing for software security testing and quality assurance*. Artech House, 2008.

[106] PaX Team. Pax address space layout randomization (aslr). 2003.

[107] PaX Team. Pax rap. https://pax.grsecurity.net/docs/pax-future.txt, 2003.

[108] PaX Team. grsecurity: RAP is here. 2016.

[109] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, 2014.

[110] Arjan van de Ven and Ingo Molnar. Exec shield. *Retrieved March*, 1:2017, 2004.

[111] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Pixelvault: Using gpus for securing cryptographic operations. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1131–1142. ACM, 2014.

[112] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216. ACM, 1994.

[113] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 260–275. ACM, 2013.

[114] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.

[115] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *USENIX Security Symposium*, pages 29–46, 2010.

[116] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 20–37. IEEE, 2015.

[117] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 304–316, New York, NY, USA, 2002. ACM.

[118] Rafal Wojtczuk. The advanced return-into-lib (c) exploits: Pax case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.

[119] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

[120] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery*, page 0. IEEE.

[121] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761. USENIX Association, 2018.

[122] Michal Zalewski. American fuzzy lop.(2014). *URL http://lcamtuf.coredump.cx/afl*, 2014.

[123] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 559–573, Washington, DC, USA, 2013. IEEE Computer Society.

[124] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, 2013.

[125] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing.

[126] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 558–569. ACM, 2014.