Thread-Level Attack-Surface Reduction

Florian Rommel
rommel@sra.uni-hannover.de
Leibniz Universität Hannover
Germany

Christian Dietrich
christian.dietrich@tuhh.de
Hamburg University of Technology
Germany

Andreas Ziegler
ziegler@cs.fau.de
Friedrich-Alexander-Universität
Erlangen-Nürnberg
Germany

Illia Ostapyshyn
ostapyshyn@sra.uni-hannover.de
Leibniz Universität Hannover
Germany

Daniel Lohmann
lohmann@sra.uni-hannover.de
Leibniz Universität Hannover
Germany

Abstract

Existing debloating techniques designed to prevent buffer-overflow exploits through return-oriented programming do not differentiate roles within a process or binary, allowing all threads access to the full program functionality. For example, a worker thread that handles client connections (highest attack exposure) still has access to all the code that the management thread needs (highest potential fallout).

We introduce thread-level attack-surface reduction (TLASR), a dynamic, context-aware approach that eliminates unused code on a thread level. For this, we (permanently or temporarily) eliminate parts of the text segment (both in shared libraries and the main binary) and use the `mmview` Linux extension to support multiple text-segment views in a single process. We reduce the executable code visible from a single thread in MariaDB, Memcached, OpenSSH, and Bash by 84 to 98.4 percent. As a result, the number of ROP gadgets decreases significantly (78–97 %), with TLASR rendering an auto-ROP utility ineffective in all investigated benchmarks and eliminating *all* CVE-related functions *ever* reported for glibc in 97 percent of the cases.

CCS Concepts: • **Security and privacy → Software security engineering**.

Keywords: debloating, binary tailoring, return-oriented programming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LCTES '23, June 18, 2023, Orlando, FL, USA

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0174-0/23/06.
https://doi.org/10.1145/3589610.3596281

ACM Reference Format:
Florian Rommel, Christian Dietrich, Andreas Ziegler, Illia Ostapyshyn, and Daniel Lohmann. 2023. Thread-Level Attack-Surface Reduction. In *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3589610.3596281

1 Introduction

Automatic elimination of unneeded program text ("debloating") has become a broadly explored technique for attack-surface reduction [22], with a strong focus on superfluous code imported from shared libraries [5, 16, 25, 28, 39]: Quach and colleagues found that only five percent of `libc` is actually used on average by the programs from the Ubuntu Desktop environment [28]. Ziegler and associates could eliminate 70 percent of all library functions on the (already size-optimized) embedded OpenWRT distribution [39]. The unused portion is bloat that can negatively impact software defenses by unnecessarily inflating their overheads or increasing the attack surface at run time [5].[1]

Text elimination is essentially a subsetting problem: The goal is to find a minimal (but sound) subset of functions that are required at run time and remove everything else. The definition of "minimally required" depends on the *sharing context*, which in the case of shared libraries, typically is a single program [28] or a set of programs [5, 39]. Within the given context, most approaches basically mimic the dead-code elimination of a static linker: They perform a reachability analysis, starting with the program-specific library entry points and cut off everything else. This can be done statically from the library binaries [5, 39], at load time [28], or even dynamically by restricting and expanding the set at run time only to include the library functions that are currently reachable from an entry function on the call stack [25]. In all cases, however, the sharing context is, at least, the whole

[1] Note that the term "attack surface" here actually means "exploit surface", as it refers to the gadgets and tools exploitable by an attacker *after* an initial breach into the system (e.g., by a buffer overrun) has been found, that is, in the second stage. We stick with this notion for the sake of convention.

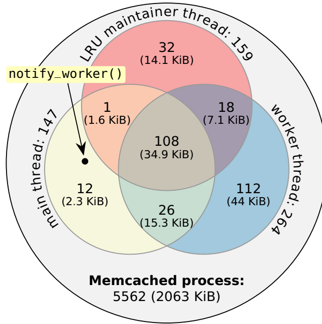

Figure 1. Intersection of available and required functions for thread roles in Memcached when serving Memtier 1.3.0.

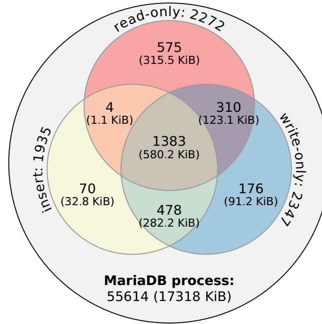

Figure 2. Intersection of available and required functions for three MariaDB *workers* serving different OLTP benchmarks.

process: In a multithreaded server application, each thread has access to *all* functions accessed by *any* other thread.

About this Paper In this paper, we argue for shifting the sharing context from the *whole program/process* to the *single thread*: Today's server software employs a multitude of threads that fulfill different "roles" at run time, for which they only need access to a fraction of the available text. Fig. 1 shows this on the example of Memcached, where we have, among others, the roles *main thread*, *LRU maintainer thread*, and *worker thread*. Out of the 5 562 functions loaded into the process, a *main*/*LRU*/*worker* thread, however, only needs 147/159/264 functions, (that is, only 3–5 percent). With *thread-level sharing*, all other functions could be eliminated for these threads. So even if an attacker succeeds in hijacking a *worker*, she finds only a drastically reduced function set and, for instance, still cannot control other threads by invoking `notify_worker` (e.g., using return- or jump-oriented programming (ROP/JOP) [11, 31, 35]), as this function is only available in the *main* thread's context.

Our Contribution We pioneer the idea of thread-specific text subsetting for attack-surface reduction, which significantly reduces the attacker-accessible code in multithreaded applications. In this realm, we present *thread-level attack-surface reduction* (TLASR), a new approach for dynamic text elimination that is based on two pillars: **(1)** The concept of *context-based text elimination* (CTE), a new library and toolset that provides easy-to-apply means for function-level text elimination – either permanently (i.e., *kill* functions so that they could never be called) or temporarily (i.e., *wipe* functions until they get called by a legitimate caller). **(2)** The application of *memory views* (`mmviews`, originally developed to speed up live patching [32]) for attack-surface reduction.

Like other debloating approaches (e.g., [25, 28]), we eliminate individual functions that are not required in the current context. Unlike with existing approaches, such contexts (a) also exclude functions from the application binary itself and (b) can be applied in a thread-specific way, resulting in extremely high reduction results.

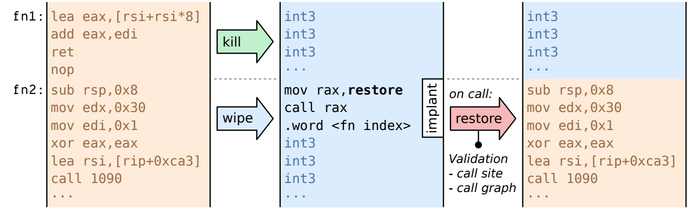

Figure 3. Killing, wiping, and restoring with libCTE.

2 TLASR Approach and Attacker Model

We assume that the target application is a TLASR-enabled multithreaded server process, provided as native machine code in ELF format (i.e., not interpreted or JIT'ed) that runs on a system with uncompromised hardware. Furthermore, we expect the standard W^X (write xor execute) model for code pages. For simplicity, we currently exclude loading additional code after the user-definable point of initialization (i.e., `dlopen`), but this is not a fundamental requirement. Note that throughout this paper, *user* refers to the developer/devops maintainer of the application, who applies TLASR.

The attacker sits on the client side and is connected to some thread of the server. We assume that the attacker is in possession of the binary code of all loaded objects in the targeted process and can exploit some existing vulnerability (e.g., a buffer overflow) that allows the manipulation of thread-local data (especially the stack) in this thread. Our goal is not to prevent this initial attack but to reduce its fallout by providing constructive means for hindering its further exploitation in the second stage by common exploit techniques, such as ROP/JOP [11, 31, 35].

2.1 (Thread-Specific) Text Contexts

The fundamental difference between TLASR and other text elimination approaches is the support of *multiple contexts* for the available text elements at run time. Technically, a context is realized and represented by an in-process address-space object (`mmview`) that shares everything but the text segment with other `mmviews` [32]. In TLASR, an `mmview` can implement an arbitrary context, for instance, a context that covers only functions needed within a specific part of the program's call graph, which, for instance, would mimic the dynamic get-what-you-want approach of BlankIt [24, 25], but without being restricted to library entry points. Within a context's `mmview`, TLASR disables/enables code on function granularity, either temporarily (*wipe*, as in BlankIt), or forever, (*kill*), which effectively mimics static debloating and is comparable to the idea of piece-wise loading [28], again without being restricted to library code.

TLASR contexts provide an easy and natural way to isolate the code required by groups of threads or even individual threads from each other. This can drastically reduce the attack surface in multithreaded server applications. We have already seen this on the example of thread roles in Memcached in Fig. 1: In a role-specific context for the most attack-prone

worker threads, 95 percent of the available functions were not loaded. However, depending on the architecture of the server application, even more fine-grained contexts, such as *per-client*, *per-connection*, or *per-transaction* should be considered: Fig. 2 depicts the function-set intersections of three *worker* threads in MariaDB serving different clients: All three thread instances share the same role, for which they need a total of 2 996 functions out of the 55 614 functions available in a MariaDB process (5 percent) in their *role-specific set*. However, actually all three threads share only 1 383 functions (2 percent). Restricted to a *connection-specific* function set, between 70 and 575 functions are exclusive for every single thread. For instance, all write functionality is *wiped* from the worker thread serving the (typically lesser-privileged) read-only client. Such dynamic disabling/enabling of functions facilitates very tight contexts but also requires some extra effort to validate legit *unwipe* attempts at run time.

2.2 The Role of the Binary

Existing approaches [5, 16, 25, 28, 39] target shared libraries only and mostly ignore the attack surface from the application binary – which is also not included in the baseline of their reported reduction rate. The focus on shared libraries is commonly justified by considering them as *the* major source of bloat. While this is true for smaller applications (UNIX *core utils* are a common evaluation target), we consider this assumption questionable for larger multithreaded applications: For instance, from the 55 614 loaded functions in a MariaDB process, 33 413 (60 percent!) actually stem from the `mysqld` binary – and only 2 613 of them (8 %) are needed to serve an `oltp_read_write` client. Hence, for larger applications, the binary can even be the *most significant* source of bloat.

In contrast to the existing approaches, TLASR *is* able to eliminate functions from the application's binary. Hence, all reported numbers in this paper relate to the functions loaded into the process/thread, independently from their origin.

2.3 TLASR Application and Limitations

TLASR is provided as a user library (libCTE) plus an analysis tool (CTEmeta) to extract caller–callee relations from program binaries. Unlike other debloating approaches, TLASR currently requires active integration into the source code and, thus, a recompilation of the application (with the standard compiler). The code modifications are straightforward and minimal. While an automated application of TLASR to existing binaries would be possible (we shall discuss this briefly in Sec. 5), we decided against it at this stage, as the most useful utilization of additional text contexts – a main feature of TLASR – requires some knowledge about the application anyway. In this sense, TLASR could also be seen as a code-only, easy-to-apply, and fine-grained alternative to approaches for explicit compartmentalization, such as Wedge [10], lightweight execution contexts [21], or manual splitting of the application logic into multiple processes [30].

The multi-context facility in TLASR requires kernel support for sub-process memory isolation. Our current implementation of contexts is based on our published [2] implementation of *address space (AS) views*, originally suggested as *AS generations* for wait-free binary patching in multithreaded applications [32]. TLASR's `mmviews` are just AS views that share everything, but the text segment. We are well aware that the dependence on a modified kernel would prevent the application of TLASR's multi-context feature in most production settings. We see TLASR as a research vehicle for understanding and improving hardening of multithreaded applications, which have been underrepresented in existing debloating attempts. Given the rising relevance of threads and thread roles in server software, we are also optimistic that, in the longer term, concepts for sub-process memory isolation and virtualization will become available in commodity kernels. The integration of *memory protection keys* in Linux can be seen as a step in this direction.

Despite the benefits of multiple contexts, libCTE and CTEmeta could nevertheless be applied productively without the kernel extension. In this case, only a single context (provided by the process's standard address space) is available, from which developers could wipe unneeded functionality. This is comparable to other dynamic approaches [25] but still has the advantage of also targeting the application's binary. We shall present examples for this in Sec. 4.

3 Implementation

We implemented CTE as *libCTE*, a run-time library, and *CTEmeta*, a supplementary binary analysis tool. libCTE allows for the elimination and restoration of code at the function-level granularity within a running process. CTEmeta performs ahead-of-time analysis of ELFs to provide static information on a per-binary/library level, which is then utilized by libCTE. CTE does not have specific requirements for binaries and libraries, except for position-independent code and the presence of all function symbols in the ELF file (or dedicated symbol files, e.g., Debian's dbgsym packages). We do not rely on source code or debugging information. Our focus is on x86-64, but the approach can be generalized.

LibCTE allows for fine-grained control of the used elimination strategy. The user can eliminate functions within a context specified by a *wiping rule set*, which determines for each function in the process (binary+libraries) whether it should be loaded, permanently removed *(killed)*, or temporarily removed and restored on demand *(wiped)*. LibCTE provides functions to create such rule sets and to manipulate them according to the CTEmeta-supplied static call-graph information. For example, the user can mark all functions that are reachable from a thread's entry point as to be wiped and mark the unreachable ones as to be killed. We also provide manipulators to set the policy of address-taken functions (which are potential targets of function pointers).

3.1 Function Elimination and Restoration

On initialization (usually performed during the program startup), libCTE scans the program binary and all loaded libraries for function symbols with `libelf`. For each function, libCTE creates a record that stores a copy of the body in a non-executable, non-writable memory region. We exclude PLT entries, un-typed assembler code, and non-instruction data embedded in the text segment, but these make up only a small fraction of the executable bytes (2 % in MariaDB).

When calling the elimination function, CTE overwrites the to-be-removed function bodies (according to the user-defined policy) with debug trap instructions (x86: int3), ensuring a recognizable error condition upon illegal access (see Fig. 3). For the functions marked as to be wiped (not killed), libCTE installs a short trampoline (x86: 16 bytes) at the function entry consisting of a call instruction to a restore handler followed by the index of the function record.

When the regular control flow calls a wiped function, the trampoline calls the restore handler, which saves the current context, validates the call, and copies the saved function body back to the original location. We can locate the function record in constant time via the index in the trampoline.

3.2 Multiple Text Segments per Process

As already touched (Sec. 2.1), to provide per-thread memory views (`mmviews`), we use a Linux kernel extension, that we originally developed for dynamic software updates [32]. Since it is the technical substrate of our CTE approach, we want to describe it briefly and explain the libCTE integration.

With `mmviews`, each process can have multiple address spaces that are structurally equivalent (have the same list of mappings) but have private user data in explicitly-marked mappings. Technically, this is achieved by having a separate page-table tree per `mmview`, which is (lazily) synchronized with the process's other `mmviews`; mapping operations are executed on all views simultaneously. Threads switch between the `mmviews` of a process via a system call (`mmview_migrate`), which boils down to exchanging the CPU's page directory.

For integration with CTE, we mark the text segments of the loaded ELFs as `mmview`-private, which results in kill, wipe, and restore operations only influencing the currently active `mmview`. The user can instantiate a new CTE context by creating a new `mmview`, migrating the current thread to it, and performing the function elimination. A thread may live in a specific context for its whole lifetime or only for the duration of a function call (component-level context).

3.3 Load-Time Validation

Unlike *killed* functions, which are permanently removed, *wiped* functions can be restored on demand by the trampoline-invoked restore handler. In order to harden this process and to hinder the exploitation of CTE itself as an attack vector, we validate the load decision before restoring.

Call-Site Inspection With call-site inspection, we ensure that the handler was invoked from an actual call site by inspecting the instruction that precedes the current return location. With this, ROP chains cannot provoke a function restore, as they do not originate from a valid call site.

For architectures with variable-length instructions (x86), backward instruction decoding is impossible in the general case. However, we can detect almost all invalid call sites by matching the respective instruction patterns against the bytes preceding the return address.

Caller–Callee Validation To further harden the load decision, we additionally verify that the invocation is covered by the regular program behavior. For this, libCTE uitilizes the static call-graph information pre-calculated by CTEmeta.

With *Zydis* [8], CTEmeta disassembles the function bodies to extract caller–callee information from direct-call sites and collect functions that contain indirect calls. It also captures calls to external functions through the program-linkage table (PLT). Additionally, CTEmeta calculates address-taken information by scanning the code for PC-relative addressing and by parsing the relocations to capture function pointers. CTEmeta is able to process even large applications efficiently (MariaDB+libraries: < 2 min run time; < 100 MiB RAM usage).

At initialization, libCTE combines CTEmeta's per-ELF information into a single call graph and consolidates the PLT entries. Currently, we require that the dynamic linker resolves all external symbols at load time (no deferred loading; enforced via the `LD_BIND_NOW` environment variable). As a side effect, dynamically-resolved functions (GNU ifuncs) do not require special treatment. All in all, CTE is able to handle all libraries for our benchmarks without modifications.

At restore time, libCTE consults the caller's callee list to validate direct calls. For indirect calls it uses the address-taken information and the list of functions that contain indirect calls to ensure the expected behavior.

4 Evaluation

In the evaluation, we will quantify the achieved attack-surface reduction in single- and multithreaded programs, as well as the induced overhead for function (un-)loading and multiple memory views. We apply our methodology to two single-threaded programs (Bash, OpenSSH), perform component-specific code wiping, and apply the full approach to two multithreaded server applications (Memcached, MariaDB). To quantify the attack surface, we use the executable bytes, the loaded-function count, and the number of ROP gadgets. We identify gadgets with the ROPgadget tool [34], and use its *auto-ROP generator* (AR) to conduct automatic ROP chain attacks that try to execute the `execve()` system call. We performed all measurements on a quad-core Intel i5-6400 @ 2.70 GHz machine with 32 GiB RAM. It runs Gentoo with an `mmview`-enabled Linux 5.15 kernel.

4.1 CTE: Single-Threaded Programs

First, we apply CTE to Bash 5.1 and the OpenSSH 8.8p1 daemon to reduce not only the amount of executable code within the libraries (as done by [5, 16, 25, 28, 39]) but also within the main binary itself. As both programs are single-threaded, there is no requirement for thread-specific code wiping. Thus, the `mmview` kernel extension is not necessary.

We eliminate all functions, excluding `main` and the functions necessary for CTE's re-loading procedure. We let CTE install the load trampoline in all functions reachable from `main`, as well as address-taken functions. The compilation of both projects uses their default configurations. As benchmarks, we run Bash's own `configure` script and execute a single public-key-authenticated login process in OpenSSH.

In Tab. 1, we quantify three different attack surfaces in terms of code size, function count, and number of gadgets: (1) the attack surface before wiping, (2) the maximal attack surface if all functions with a trampoline had been loaded, (3) the actual attack surface after executing the chosen benchmark. We also distinguish between the main executable, the loaded shared libraries, and the required CTE runtime. With this separation, we not only disclose the CTE-induced attack surface but also demonstrate that CTE is able to eliminate functions from the executable and the CTE runtime itself. It is worth mentioning that CTE also killed (permanently removed) functions from the binaries, despite the expectation that they would be reachable from `main`. Manual inspection confirmed that it was dead code surviving the build process.

For Bash, we reduced the code size by 84 percent and wiped 87 percent of all functions (35 % permanently). Thereby, we reduced the number of gadgets by 85 percent. Similarly, for OpenSSH, we reduced the code size by 84 percent and wiped 90 percent of all functions (38 % permanently), and removed 78 percent of all gadgets. For both benchmarks, CTE removed enough gadgets to let the AR fail even though we consider the inserted trampolines as possible gadgets.

We also measured the run-time overhead: For Bash, we initialized CTE for 16.6 ms, wiped functions for 2.4 ms, and loaded them again for a total of 4.9 ms (695 restores). For OpenSSH, init/wipe/restore took 35.8 ms/4 ms/5.9 ms (1 048 restores). Since we wipe exactly once in these scenarios, these numbers represent the complete run-time penalties.

4.2 Function Isolation with TLASR

For single-threaded programs, CTE not only significantly reduces loaded code, but also limits the maximally loadable functions. However, we can go further by utilizing `mmview`s to contextualize the function-load decision for individual (manually selected) code components, such as a single function. We create a separate `mmview` for the context, eliminate all non-necessary code, and switch the calling thread to the new `mmview` during the execution of the component. As `mmview`s share their data and stack segments, parameter passing (even by reference) requires no copies but only a wrapper function. For the function call, the wrapper migrates the calling thread to a wiped `mmview` that only contains the necessary functions. The attack surface of the "isolated" function is determined only by itself and its callees.

Table 1. Attack surface of single-threaded programs. For each column, the triple (init/max/at exit) gives the surface for the program before wiping, when the maximal reachable set would have been loaded, and the actual attack surface after the benchmark finished. The CTE row shows the attack surface that we introduce.

	Code Size [KiB]	Functions [#]	Gadgets [#]
bash	(Benchmark: `bash ./configure`)		
Exec.	779/ 755/ 284	2 252/ 2 002/ 507	9 423/ 8 724/ 3 159
Libs. (7)	1 601/ 1 094/ 110	3 926/ 2 050/ 300	24 724/ 16 045/ 1 995
CTE	86/ 43/ 11	262/ 138/ 27	1 743/ 1 132/ 322
sshd	(Benchmark: Pub-key login until auth. succeeds)		
Exec.	700/ 605/ 206	1 293/ 1 059/ 309	2 857/ 2 515/ 909
Libs. (13)	3 537/ 2 716/ 477	11 116/ 6 624/ 893	55 801/ 41 995/11 781
CTE	86/ 43/ 11	262/ 138/ 27	1 743/ 1 132/ 322

Table 2. Function isolation metrics for three scenarios. The number of functions includes the binary and libraries.

		Empty	SSL	JPEG
Text Segment	[KiB]	1 575	3 521	3 558
TLASR	[KiB]	18	282	96
Functions	[#]	3 823	11 534	10 273
TLASR	[#]	43	319	235
Gadgets (AR)	[#]	24 513 (✓)	56 718 (✓)	50 845 (✓)
TLASR	[#]	538 (×)	8 599 (×)	1 752 (×)
Run Time		1.48±0.73 ns	33.63±0.09 us	321.40±5.80 ms
TLASR		1 045.65 ns	35.52 us	322.88 ms

To show the principal applicability of this technique, we use three synthetic benchmarks: (1) *Empty* only invokes a single, non-inlined function returning nothing; it depends solely on the libC. This allows us to measure the base overhead for the two `mmview` migrations that are required per invocation. (2) *SSL* uses OpenSSL 1.1.1l (`libssl`, `libcrypto`) to calculate the fingerprint of a predefined certificate and compares the result to a fixed fingerprint (certificate pinning); the function only returns a boolean value to its caller. (3) *JPEG* loads and decodes an 9.6 MiB image (3288 px × 4384 px) with libjpeg-turbo 2.1.1-r2, converts it to grayscale, and saves the result as a JPEG to a `tempfs` file. We isolate the image decoding procedure in a wiped `mmview`.

For the three benchmarks, we compare the baseline variant without code elimination to a TLASR-protected execution context that uses a pre-wiped `mmview` for the specific scenario. We did *not* measure the CTE run-time overheads here, but only accounted for the `mmview` costs. Tab. 2 compares the attack surface of the baseline variant to the isolated context, and shows the per-invocation run-time costs. For Empty/SSL, we batched $1 \cdot 10^6 / 1 \cdot 10^4$ invocations to minimize the measurement's influence. We repeated each (batched) measurement 100 times.

Table 3. Loaded Attack Surface (MariaDB, Sysbench SQL).

Benchmark	Code Size	Functions	Gadgets (AR)
MariaDB	16.91 MiB	55 614	186 788 (✓)
bulk_insert	7.03 %	1942 (3.49 %)	9.36 % (×)
oltp_delete	6.81 %	2027 (3.64 %)	9.38 % (×)
oltp_insert	6.89 %	1935 (3.48 %)	9.29 % (×)
oltp_point_select	6.45 %	1817 (3.27 %)	8.70 % (×)
oltp_read_only	7.61 %	2272 (4.09 %)	9.64 % (×)
oltp_read_write	9.76 %	2926 (5.26 %)	11.34 % (×)
oltp_update_index	7.55 %	2240 (4.03 %)	10.00 % (×)
oltp_update_non_index	6.94 %	2089 (3.76 %)	9.53 % (×)
oltp_write_only	7.93 %	2347 (4.22 %)	10.20 % (×)
select_random_points	7.09 %	1983 (3.57 %)	9.14 % (×)
select_random_ranges	7.17 %	2015 (3.62 %)	9.19 % (×)

TLASR is able to eliminate significant parts of the text segment (92 %–99 %) and the loaded functions (97 %–99 %). Thereby, TLASR reduced the number of gadgets between 85 percent (SSL) and 98 percent (Empty) and successfully prohibited all AR ROP attacks.

From Empty's and SSL's run times, we see that switching the `mmview` of the calling thread costs around 560 ns per `mmview_migrate` (Empty). Therefore, we expected that TLASR's overhead diminishes for the long-running JPEG benchmark. However, we observed a measurable overhead (+0.46 %). An inspection of the benchmark's behavior revealed the problem: For the decompressed image data, libjpeg allocates a 41 MiB buffer, which glibc's `malloc` directly forwards to `mmap/munmap`. As this buffer is used in both `mmviews`, the overhead is caused by the page-table synchronization.

4.3 Multithreaded Applications

In the third part of our evaluation, we apply TLASR to two multithreaded server programs (MariaDB 10.8, Memcached 1.6.12). We chose these programs because they are widely used and have different worker-thread models: MariaDB's SQL server uses one worker thread per client connection by default, which executes the received SQL statements sequentially. Memcached, in contrast, is event-based, and each worker thread handles multiple connections simultaneously without blocking on IO. The two different models allow us to investigate both associated code-usage patterns: While we expect that a MariaDB worker requires vastly different function sets depending on the SQL command, each Memcached worker should use more or less the same functions. However, for Memcached, we also considered background threads, which differ significantly from the workers.

For MariaDB, we applied TLASR to the client-facing worker threads with the possibility to re-wipe `mmviews` periodically. Thereby, we can provide a clean `mmview` for each connection, each (Nth) transaction, or each SQL command. For the measurements, we used all SQL benchmarks from Sysbench 1.0.20 with a varying number of clients, acting on a single database table with 10 000 rows. Per default, each transaction consists of 20 SQL queries, and Sysbench opens one connection per client, each corresponding to one MariaDB worker thread. Except for end-to-end latency measurements

Table 4. Per-library breakdown for MariaDB and the benchmark `oltp_read_write`. TLASR introduces the `libcte.so` and `libelf.so.1` libraries (gray rows).

ELF/Library			Loaded [%]	
	Size [KiB]	Funcs. [#]	Size	Funcs.
mysqld	9 021.34	33 413	14.68	7.82
libcrypto.so.1.1	1 581.96	6 227	11.08	0.19
libc.so.6	1 289.18	3 130	4.56	3.83
libstdc++.so.6	1 013.15	4 836	1.76	0.62
libzstd.so.1	974.91	726	0.22	0.83
libgcrypt.so.20	831.69	1 337	7.88	1.65
libm.so.6	608.34	898	0.11	0.67
libsystemd.so.0	548.86	1 572	1.28	0.19
libpcre2-8.so.0	428.93	258	0.14	0.78
libssl.so.1.1	306.97	1 196	2.25	0
ld-linux-x86-64.so.2	142.01	263	0.63	1.90
liblz4.so.1	107.41	148	1.05	2.03
libgpg-error.so.0	83.93	411	2.21	0.49
libcrypt.so.2	82.88	128	0.02	0
libgcc_s.so.1	69.55	204	1.68	20.10
libelf.so.1	63.31	205	1.28	0
libpthread.so.0	56.27	288	16.37	10.42
libz.so.1	52.61	133	1.38	0
libcte.so	22.32	57	44.21	47.37
libcap.so.2	13.47	62	30.86	4.84
librt.so.1	13.24	72	7.26	0
libdl.so.2	4.05	29	6.39	3.45
libaio.so.1	1.49	21	8.48	0
Total	17 318	55 614	9.76	5.26

(see Sec. 4.3.2), we execute Sysbench on the same host as the database. The baseline is an unmodified MariaDB (v10.8).

For Memcached, we applied TLASR to all threads, with each thread living in its own `mmview`. The main thread listens for connections and assigns them to multiple workers. In addition, there are threads for asynchronous LRU maintenance and other background threads. To generate load and measure the performance, we use Memtier Benchmark 1.3.0 with its default settings. In this configuration, Memtier issues access operations to randomized keys in the database, maintaining a read/write ratio of 10/1. The number of benchmark threads matches the CPU count, with each thread creating 50 clients. Deviating from the default settings, we let the Memtier clients reconnect to Memcached after a fixed number of 100 000 requests in order to also put some load on the Memcached listener thread. As in the MariaDB benchmark, we run Memtier on the same host as the server, except for the latency measurements. For the baseline, we used the unmodified Memcached 1.6.12 server.

4.3.1 Quantitative Attack-Surface Analysis

First, we quantify the attack surface that each benchmark provokes on the server side. For MariaDB, we instruct TLASR to wipe the `mmview` of the per-connection workers once before handling the connection. We execute the benchmark with a single client for 120 seconds and dump the worker's function-load set and CTE's run-time statistics when the client closes the connection. Afterwards, we compare the attack surface against the unmodified MariaDB server.

In Tab. 3, we report the numbers for the whole MariaDB process broken down per Sysbench benchmark, where CTE

Table 5. TLASR attack surface for Memcached.

Thread	Code Size	Functions	Gadgets (AR)
Memcached	2 063 KiB	5 562	28 341 (✓)
`main`	68 KiB (3.3 %)	147 (2.6 %)	4.9 % (×)
`LRU`	72 KiB (3.5 %)	159 (2.9 %)	5.0 % (×)
`logger`	48 KiB (2.3 %)	119 (2.1 %)	3.4 % (×)
`crawler`	54 KiB (2.6 %)	134 (2.4 %)	3.9 % (×)
`maintenance`	47 KiB (2.3 %)	111 (2.0 %)	3.3 % (×)
`slabs`	46 KiB (2.2 %)	111 (2.0 %)	3.3 % (×)
`worker` (4 threads)	115 KiB (5.6 %)	264 (4.7 %)	7.8 % (×)

removes between 90 and 94 percent of the executable bytes from the worker's attack surface. The number of loaded functions varies between 1 817 and 2 926, indicating significant differences in the utilized functionality across the benchmarks. The variation mostly stems from the application binary (1 519–2 613 funcions). In contrast, the variation in the library-function usage (271–292 functions) is relatively small, which underlines the importance of removing functions from the binary. From the CTE runtime, all benchmarks load the same 27 functions already observed in Tab. 1.

Tab. 4 shows a per-library breakdown of the attack surface for `oltp_read_write`, which has the largest attack surface of the performed benchmarks (see Tab. 3). Here, TLASR performs well for all loaded libraries and the main executable. For 6 libraries, including CTE's dependency `libelf` (needed during the initialization), we eliminated all functions. The reason for the remaining code is the presence of executable data that is not captured by CTE (e.g., the program linkage table, PLT entries).

Fig. 2 shows an exemplary breakdown of the overlap in the attack surface between the three benchmarks: Naturally, the two benchmarks that modify database rows have many functions in common (478), while the read-only benchmark has the most distinctive function set (575) not used in the other two. We consider this a strong indicator that context-specific code elimination is beneficial in a per-connection threading model if clients require different services.

The code-size and function-count reductions lead to a gadget reduction between 89 and 91 percent for the workers compared to the original MariaDB (see Tab. 3). In all cases, the workers were not vulnerable to auto-generated ROP chains, unlike the original program.

For Memcached, Tab. 5 shows reductions broken down per thread, and Fig. 1 gives a more detailed view of the function-load–set overlap for three threads. Overall, TLASR is able to reduce the code size and function count by at least 95 percent (up to 98 %) within each `mmview`. This significantly reduces the available gadgets and also disarms the AR in all cases. A total of 101 functions (2 %) with 30 KiB (1.5 %) are loaded in all `mmviews`. Without `mmviews`, each thread would suffer from an attack surface of 6.9 percent of the code.

4.3.2 Run-Time and Memory Overheads In this section, we measure the runtime costs of applying TLASR to multithreaded server processes. These costs include the time

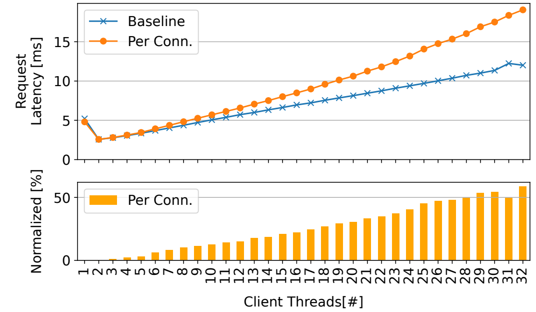

Figure 4. Sysbench latencies (oltp_read_only) for MariaDB.

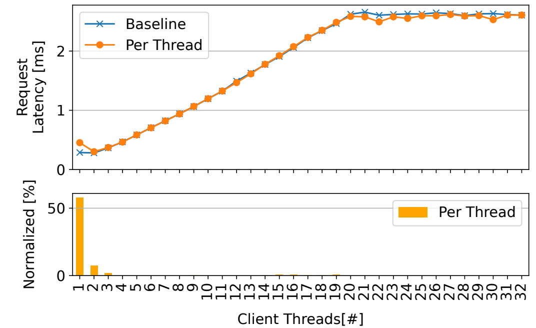

Figure 5. Memtier Benchmark latencies for Memcached.

incurred by CTE and the overhead introduced by the use of `mmviews`, which are reflected in the end-to-end request latencies experienced by the clients. We also examine the impact of different (periodic) elimination strategies at varying levels of concurrency.

First, we look at the CTE overheads regarding run-time and memory consumption for MariaDB, our largest benchmark program. CTE requires on average 204 ms to initialize its data structures. The whole elimination process (~56K functions) takes around 20 ms for all functions. Restoring functions is more expensive: While wiping a single function takes 0.31 us on average, restoring it again takes 4.59 us. This asymmetry is caused by the two `mprotect` calls to make the text-bodies (non-)writable. We can batch those calls for wiping, but for restoration, we have to do it on a per-function basis. However, for 10 of the Sysbench benchmarks, the restore activity comes close to quiescence (95 % required functions are loaded) in less than 25 ms. Only `bulk_insert` did not reach the 95 percent mark before 176 ms. At this point, we know that the main source of overhead stems from `mmviews`.

Apart from run time, TLASR also entails a space overhead. On initialization, CTE reads the CTEmeta files for all ELFs in the process, which take up 4 MiB of disk space for MariaDB (largest benchmark). At run time, CTE requires memory to save the decoded meta information and the non-executable copies of the function bodies. For MariaDB, this takes up around 23 MiB of non-executable heap memory. Furthermore, each `mmview` involves duplicating the page-table tree and the text segments. The per-`mmview` page tables are in the range of ~500 KiB for MariaDB and ~100 KiB for

Table 6. MariaDB request latency (`oltp_read_only`) with periodic elimination. Normalized to the MariaDB original.

	Threads					
[× baseline]	1	2	4	8	16	32
Per Connection	0.92	1.00	1.02	1.10	1.22	1.59
Per Transaction	3.14	7.35	8.90	19.92	25.73	30.02
Per 50 Transactions	0.82	1.23	1.15	1.32	1.46	1.73
Per 100 Transactions	0.88	0.99	1.00	1.09	1.22	1.60
Per 500 Transactions	0.93	1.00	1.01	1.10	1.21	1.58
Per 1000 Transactions	0.97	0.99	1.00	1.09	1.22	1.57

Memcached [32]. Note that all user-data pages are physically shared between `mmviews`.

Next, we look at the end-to-end latency that the clients experience when we employ TLASR in the server process. The servers are benchmarked with Sysbench (for MariaDB) and Memtier Benchmark (for Memcached) for 300 seconds from a remote host (quad-core Intel i5-7400 @ 3.00 GHz, 32 GiB RAM) connected to the server host by a gigabit ethernet link in a local-area network. We record the average of the experienced latencies. For MariaDB, we chose Sysbench's read-only benchmark as it does not suffer from transaction aborts with an increased client count.

Our elimination approach is the least intrusive when we invoke CTE precisely once per thread, with `mmview` becoming the primary source of overhead. Since a Sysbench client uses the same connection for all commands, per-thread elimination is equivalent to per-connection elimination for MariaDB.

In Fig. 4, we show the request latency for an increasing number of MariaDB clients. Here, we observe that TLASR has no or only minor influence (< 5 %) on the request latency as long as the active clients do not significantly surpass the number of CPU cores. Because MariaDB creates a worker thread for each client, an increasing number of clients causes frequent reschedules, which results in many `mmview` switches with associated costs (TLB invalidation). We measured a relative disadvantage of up to 59 percent for 32 active clients.

For Memcached, Fig. 5 shows the end-to-end latency when each thread is placed in its own `mmview` and wiped once at startup. Since Memcached's thread setup is static, a growing number of clients does not affect the number of threads and `mmviews`. As a result, the impact of TLASR on Memcached, using a once-per-thread wiping approach, remains minimal as the number of clients grows.

Next, we investigate the costs for periodic `mmview` re-wiping. We chose MariaDB for this analysis since the function-load set of a database worker heavily depends on the workload. Tab. 6 displays the resulting request latencies, normalized to the baseline. When wiping after each transaction, the clients experience request latencies between 3 and 30 times as high as the baseline. This discrepancy becomes understandable when comparing the baseline latency to the wipe/restore times: With one client, a transaction takes 4.8 ms, but the restorations of wiped functions alone take

Table 7. MariaDB request latency (`oltp_read_only`) for increasingly-sized transactions (20 × QM commands).

		Query Multiplier (QM)				
#Threads	TLASR Mode	1	5	20	50	100
1	Baseline [ms]	5.21	20.07	68.96	159.96	306.91
	Per Conn. [×]	0.92	1.02	1.03	1.01	1.12
	Per Trans. [×]	3.14	1.50	1.06	1.11	1.07
4	Baseline [ms]	3.05	14.61	57.36	144.08	286.43
	Per Conn. [×]	1.02	1.00	1.01	1.00	1.01
	Per Trans. [×]	8.90	2.19	1.22	1.09	1.06
32	Baseline [ms]	12.01	59.15	239.28	606.05	1 218.54
	Per Conn. [×]	1.59	1.59	1.54	1.55	1.52
	Per Trans. [×]	30.02	6.62	1.83	1.57	1.58

around 10 ms for the `oltp_read_only` benchmark. Thus, per-transaction wiping is clearly too expensive in the presented scenarios. The overhead could be mitigated by periodic wiping (re-wipe only every Nth transaction, see Tab. 6) or by increasing the amount of work performed within a single transaction (see Tab. 7). With decreasing frequency, the costs of the periodic strategy quickly approaches the overhead of per-connection re-wiping: If we re-wipe the `mmview` every 100 transactions, periodic re-wiping is mostly on par with the per-connection approach. In the end, it remains a trade-off between performance and attack-surface reduction. As long as it is done with a low frequency (every few seconds), periodic re-wiping should always be a viable option.

4.4 Qualitative Attack-Surface Analysis

While more reachable gadgets ease a second-stage exploit, also vulnerable functions can be an exploit-chain component as a return-to-libc attack. Therefore, removing functions with a history of vulnerability is a standard indicator for successful binary debloating [6, 19, 25–28, 36]. To this end, we analyzed *all* 127 documented CVEs of the GNU C library, which is a prime target for attackers [35, 37].

We manually identified 114 CVEs that mentioned a function name, which translates to 130 unique names. We could correlate all but 4 names to an ELF symbol, with the remaining being two historical symbols, a symbol from the name service cache daemon, and a variable name. On average, each of the 109 CVEs was related to 2.36 symbols (max=19).

We compare 109 CVEs with the observed function-load sets from the quantitative attack-surface experiments. For each CVE–benchmark combination (109 × 22), we calculate the fraction of historically vulnerable functions that was still present in the benchmark. As glibc loads some secondary libraries only on demand, the CVE-related functions were not present in 181 cases. From the remaining 2 217 cases, TLASR removed *all* CVE-related functions in 2 141 cases (97 %), and in 70 further cases at least one. Only in 6 cases we could not eliminate any of the mentioned functions. In four of those cases (all OpenSSH), the remaining vulnerable function was `getaddrinfo`, which the OpenSSH server uses

Table 8. Attack-Surface Reduction with LibraryTrader.

	Code Size	Functions	Gadgets (AR)
MariaDB	35 %	33 %	29 % (✓ → ✓)
Exec.	0 %	0 %	0 % (✓ → ✓)
Libs.	74 %	82 %	48 % (✓ → ✓)
Memcached	47 %	65 %	43 % (✓ → ✓)
Exec.	0 %	0 %	0 % (× → ×)
Libs.	52 %	70 %	46 % (✓ → ✓)

to perform reverse DNS lookups. In total, TLASR removes most functions with a history of vulnerabilities.

4.5 Comparison with Static Code Debloating

We also compare TLASR with LibraryTrader [39], a state-of-the-art static binary debloating technique that performs ahead-of-time ELF rewriting and uses dynamic tracing information to refine the set of required functions. Due to its static nature, LibraryTrader is unaware of threads and keeps all main-binary functions, removing code only from the libraries. We make this comparison, as it also works on the binary level, and it can handle complex libraries with handwritten assembler parts (e.g., `libcrypto`). In performance benchmarks, the shrunk binaries show (as expected) the same characteristics as the originals.

However, if we look at the attack surface for MariaDB and Memcached (see Tab. 8), we see that LibraryTrader is far less effective: For Memcached, it can remove at most 47 percent of the code segment. For MariaDB, where the main binary makes up more than half of the attack surface, the focus on libraries leads to a mere code-size reduction of only 35 percent, although the approach can reduce most of the library code (74 %). In comparison, a TLASR-wiped worker has, at most (`oltp_read_write`), 4 percent of its library code loaded, and the main executable is reduced by 85 percent.

We also quantified the effectiveness of LibraryTrader in removing CVE-related functions (see Sec. 4.4) for MariaDB and Memcached. For MariaDB, LibraryTrader could eliminate all related functions for 38 CVEs and at least some for 24 CVEs, while 41 CVEs were not addressed at all. For Memcached, the result is even worse: only 20/15 CVEs were full/partially addressed, while 63 CVEs were not addressed.

5 Discussion

Attack-Surface Reduction By (1) covering the main binary and (2) fine-grained thread-level killing and wiping, we are able to reduce the code size between 84 and 98.4 percent and shrink the gadget count by 78 to 96.7 percent. In contrast to related work that targets only libraries (e.g., [25, 28]), these numbers include all text segments, the non-wipeable parts of CTE (~5 KiB), and the injected trampolines.

Nevertheless, Brown et al. [12] argue that counting bytes and gadgets is insufficient, and a qualitative analysis of the attack surface is required. We meet this demand by (1) our auto-ROPer experiments for TLASR-wiped benchmarks, and (2) by our qualitative CVE-function analysis, where we removed almost all glibc functions with a history of vulnerabilities. Even if it may still be possible to construct ROP chains and access vulnerable functions, it becomes more difficult if every thread has a different (and possibly changing) function and gadget set that is unknown to the attacker. We might also mitigate blind ROP techniques [9], which use program crashes to probe automatically restarting servers, as TLASR can distinguish between a normal crash and a jump to an eliminated code region. Here, TLASR can be combined with restart delays or IP-address blocking.

With our load-time validation, we also defuse the restore handler as an additional attack vector and avoid loosing the CTE-achieved robustness improvements. Our two mechanisms are related to context-insensitive *control-flow integrity (CFI)* techniques [4]; however, in contrast to the usual usage pattern, we only apply them at load time and not continuously. Although we currently only validate forward edges, which has been considered easy to circumvent [38], we still argue that this nevertheless improves on the restore-handler attack vector: As we narrow contexts to individual threads, we shrink the caller/callee sets (and, thereby, call gadgets and return targets), which allows other CFI-mechanisms to become stricter. Furthermore, CTE could use a more sophisticated CFI mechanism to validate restores.

As a measure against ROP attacks, shadow stacks are a well-established defense technique, protecting the backward edges of the control flow. Unfortunately, software-based implementations come with significant performance overheads [15] and are often susceptible to modifications to the shadow stack itself [4, 13, 40, 41]. With Intel's *Control-Flow Enforcement Technology (CET)* [3, 17], the CPU manages a shadow stack with minimal performance overhead and branch-target instructions provide forward-edge verification. However, it still allows calling most functions through indirect branches, especially vulnerable library functions (Sec. 4.4) or other dangerous functions (e.g., `exec` or `system`).

In contrast to CFI, TLASR thwarts ROP attacks by restoring only necessary functions and entirely forbidding others. For indirect calls, TLASR currently consolidates process-wide address-taken information and performs call-site inspection, which is stricter than Intel's Indirect Branch Tracking and could further be improved with better static analysis.

With *Pointer Authentication Code (PAC)* [7], ARM provides a CFI technique based on cryptographic pointer authentication, which can be used to protect function returns. It has similar shortcomings regarding indirect branches as Intel CET. We see TLASR either as an additional safeguard that can be applied in conjunction to CFI or as a lightweight alternative when the fast but new [17, 29] hardware implementations are not available or supported by the OS [14].

Run-Time Impact Since `mmviews` are technically different address spaces, their usage increases the TLB pressure if they are concurrently active on the same core: In most scenarios,

we can attribute the largest part of the run-time overhead to `mmviews` (see Figures 4 & 5). This becomes especially prevalent when an application uses more CPU-bound threads than cores (which should be avoided anyway). Only with periodic elimination, libCTE itself induces continuous run-time overheads (see Tab. 6). From our results, we conclude that high-frequent periodic re-wiping (e.g., per-transaction wiping) is too expensive, but practical tradeoffs exist. Costs could be further reduced by employing a predictor, as suggested by BlankIt [25], to avoid wiping frequently-restored functions.

Applicability In contrast to other debloating mechanisms, the TLASR approach has two hurdles to its applicability: Firstly, the requirement of a kernel extension (see Sec. 2.3) and, secondly, our requirement for manual integration with the target program. To some extent, the first limitation could be eased by using multiple processes (or other protection-domain techniques [10, 21] – which, however, also require a kernel extension) that use shared mappings for their data. However, this would require actively synchronizing mapping operations to avoid diverging memory views, which we avoid with `mmviews`. Furthermore, splitting an existing application into multiple processes requires significant development efforts, whereas TLASR is very easy to apply.

However, in contrast to other debloating attempts, TLASR has to be applied manually. We argue that this is not necessarily a disadvantage, as instead of delegating debloating to the end user, it provides developers with a means to integrate fitted attack-surface reduction measures directly into their application. Nevertheless, CTE could easily be extended to support automatic application modes: A simple mode would hook the thread creation in the C library, create an `mmview` for the new thread, kill all functions that are not reachable from the thread entry, and wipe the others.

6 Related Work

The main advantage of TLASR over existing techniques is the much finer control over the function-load set, which no longer has to be the superset required by any thread in a multithreaded application, but can be adopted over time and context: Nevertheless, some aspects of those techniques could be employed in future work to improve TLASR further: *Piece-wise loading* [28], for instance, uses compiler-supplied information to eliminate unreachable functions permanently for all threads. For this, they employ a sophisticated code-pointer analysis, which CTE could make use of to increase the number of killable functions. *BlankIt* [24, 25] pioneered the idea of *dynamic* text elimination for shared libraries (similar to TLASR's function isolation), but currently does not support multithreaded programs. However, CTE could benefit from their advanced predictor to reduce restoration overheads. *Slimium* [27] targets the Chromium browser and combines static and dynamic analyses but only eliminates unneeded features from the application binary. In some broader sense, *site isolation* [30] in browsers (one process per origin) is related to TLASR's `mmview` contexts.

Razor [26] and *Chisel* [19] analyze test-case traces to remove basic blocks or even individual statements; *Trimmer* [6, 36] removes code only invoked by unused command-line arguments and config flags. Neither of them considers multiple threads. However, we could restrict the function set even further with their static analysis results.

Davidsson et al. [16] use build-time whole-system optimization, driven by a global function dependency graph like [28], to remove code from binary and libraries.

Shredder [23] specializes API calls and their allowed parameters to a given application, blocking the execution of unexpected invocations. *Ghavamni et al.* [18] disable system calls according to the current *program phase* (e.g., initialization, serving), which is also context-specific attack-surface reduction, but only on the time axis.

Similar to our memory views, *MultiK* [20] uses specialized kernel binaries per application to reduce the kernel's attack surface. It does not support dynamic elimination and operates on the process-level context only.

7 Conclusion

We presented TLASR (*thread-level attack-surface reduction*), a new approach for dynamic text elimination that reduces the second-stage attack surface in multithreaded server applications. TLASR, which we provide as a user-library plus kernel extension, eliminates code permanently or temporarily on the function granularity. Unlike existing techniques, TLASR (1) eliminates code not only from shared libraries but also from the application binary itself and (2) supports thread-specific elimination by multiple *contexts* — lightweight in-process address spaces that share all memory but the text segment. Thereby, a thread's attack surface shrinks to those functions that are actually required in its control flow. The ROP gadgets in MariaDB's worker threads can be reduced by 89–91 percent, while Memcached's code size and function count shrinks by 95–98 percent per thread. In all cases, TLASR-eliminated threads were protected against auto-generated ROP chains. In a study of all documented glibc CVEs, we found that TLASR was able to remove all CVE-related functions for 97 percent of the 109 CVEs that mention a function.

> *Please refer to the published artifact to verify and repeat the experiments [33]. The CTE source code and the Linux mmview extension are also available separately [1, 2].*

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and dedicated efforts in helping us improve this paper. TLASR was funded by the *Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)* – 468988364, 501887536.

References

[1] [n. d.]. Context-Based Text Elimination. https://github.com/luhsra/cte.

[2] [n. d.]. Linux with mmview extensions. https://github.com/luhsra/linux-mmview.

[3] 2022. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4.* https://cdrdv2.intel.com/v1/dl/getContent/671200

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity: Principles, Implementations, and Applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. ACM Press, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[5] Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2020. Large-Scale Debloating of Binary Shared Libraries. *Digital Threats: Research and Practice* 1, 4, Article 19 (Dec. 2020), 28 pages. https://doi.org/10.1145/3414997

[6] Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Junaid Haroon Siddiqui, and Fareed M Zaffar. 2021. TRIMMER: An Automated System for Configuration-based Software Debloating. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3095716

[7] Arm Limited. 2022. *Arm® Architecture Reference Manual for A-Profile Architecture.* Cambridge, England. DDI 0487H.a.

[8] Florian Bernd. [n. d.]. Zydis: Fast and lightweight x86/x86-64 disassembler and code generation library. https://github.com/zyantific/zydis – accessed on 2023-01-06.

[9] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, USA, 227–242. https://doi.org/10.1109/SP.2014.22

[10] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)* (San Francisco, California). USENIX Association, USA, 309–322.

[11] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)* (Hong Kong, China) *(ASIACCS '11)*. Association for Computing Machinery, New York, NY, USA, 30–40. https://doi.org/10.1145/1966913.1966919

[12] Michael D. Brown and Santosh Pande. 2019. Is Less Really More? Towards Better Metrics for Measuring Security Improvements Realized through Software Debloating. In *Proceedings of the 12th USENIX Conference on Cyber Security Experimentation and Test (CSET '19)* (Santa Clara, CA, USA) *(CSET'19)*. USENIX, USA, 5.

[13] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 985–999. https://doi.org/10.1109/SP.2019.00076

[14] Jonathan Corbet. 2022. Shadow stacks for user space. https://lwn.net/Articles/885220/ https://lwn.net/Articles/885220/.

[15] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS '15)*. ACM, Singapore Republic of Singapore, 555–566. https://doi.org/10.1145/2714576.2714635

[16] Nicolai Davidsson, Andre Pawlowski, and Thorsten Holz. 2019. Towards Automated Application-Specific Software Stacks. In *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11736)*, Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan (Eds.). Springer, 88–109. https://doi.org/10.1007/978-3-030-29962-0_5

[17] Tom Garrison. 2020. Intel CET Answers Call to Protect Against Common Malware Threats. https://www.intel.com/content/www/us/en/newsroom/opinion/intel-cet-answers-call-protect-common-malware-threats.html.

[18] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. 2020. Temporal System Call Specialization for Attack Surface Reduction. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1749–1766. https://www.usenix.org/conference/usenixsecurity20/presentation/ghavamnia

[19] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, ON, Canada) *(CCS '18)*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). Association for Computing Machinery, New York, NY, USA, 380–394. https://doi.org/10.1145/3243734.3243838

[20] Hsuan-Chi Kuo, Akshith Gunasekaran, Yeongjin Jang, Sibin Mohan, Rakesh B Bobba, David Lie, and Jesse Walker. 2019. MultiK: A Framework for Orchestrating Multiple Specialized Kernels. *arXiv preprint arXiv:1903.06889* (2019).

[21] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 49–64. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton

[22] Pratyusa K. Manadhata and Jeannette M. Wing. 2011. An Attack Surface Metric. *IEEE Transactions on Software Engineering* 37, 3 (2011), 371–386. https://doi.org/10.1109/TSE.2010.60

[23] Shachee Mishra and Michalis Polychronakis. 2018. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3274694.3274703

[24] Chris Porter, Sharjeel Khan, and Santosh Pande. 2021. On-the-fly Code Activation for Attack Surface Reduction. *CoRR* abs/2110.09557 (Oct. 2021). arXiv:2110.09557 https://arxiv.org/abs/2110.09557

[25] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt Library Debloating: Getting What You Want Instead of Cutting What You Don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI '20)*. Association for Computing Machinery, New York, NY, USA, 164–180. https://doi.org/10.1145/3385412.3386017

[26] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium* (Santa Clara, CA, USA) *(USENIX Security '19)*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, Berkeley, CA, USA, 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

[27] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event) *(CCS '20)*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). Association for Computing Machinery, New York, NY, USA, 461–476. https://doi.org/10.1145/3372297.3417866

[28] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium (USENIX Security '18)*. 869–886.

[29] Qualcomm Technologies, Inc. 2017. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions.* Technical Report. San Diego, CA, USA.

[30] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site Isolation: Process Separation for Web Sites within the Browser. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1661–1678. https://www.usenix.org/conference/usenixsecurity19/presentation/reis

[31] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 2 (mar 2012), 34 pages. https://doi.org/10.1145/2133375.2133377

[32] Florian Rommel, Christian Dietrich, Daniel Friesel, Marcel Köppen, Christoph Borchert, Michael Müller, Olaf Spinczyk, and Daniel Lohmann. 2020. From Global to Local Quiescence: Wait-Free Code Patching of Multi-Threaded Processes. In *14th Symposium on Operating System Design and Implementation (OSDI '20)*. 651–666.

[33] Florian Rommel, Christian Dietrich, Andreas Ziegler, Illia Ostapyshyn, and Daniel Lohmann. 2023. Thread-Level Attack-Surface Reduction - Artifact. https://doi.org/10.5281/zenodo.7939291

[34] Jonathan Salwan. [n. d.]. ROPgadget: Gadgets finder and auto-roper. http://shell-storm.org/project/ROPgadget/ – accessed on 2022-02-01.

[35] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[36] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/3238147.3238160

[37] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. 2011. On the Expressiveness of Return-into-libc Attacks. In *Recent Advances in Intrusion Detection*. Springer, Berlin, Heidelberg, 121–141.

[38] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proceedings of the 2015 ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. ACM, 927–940. https://doi.org/10.1145/2810103.2813673

[39] Andreas Ziegler, Julian Geus, Bernhard Heinloth, Timo Hönig, and Daniel Lohmann. 2019. Honey, I Shrunk the ELFs: Lightweight Binary Tailoring of Shared Libraries. *ACM Transactions on Embedded Computing Systems* 18, 5s, Article 102 (Oct. 2019), 23 pages. https://doi.org/10.1145/3358222

[40] Changwei Zou, Yaoqing Gao, and Jingling Xue. 2022. Practical Software-Based Shadow Stacks on x86-64. *ACM Transactions on Architecture and Code Optimization (TACO '22)* 19, 4, 1–26. https://doi.org/10.1145/3556977

[41] Changwei Zou, Xudong Wang, Yaoqing Gao, and Jingling Xue. 2022. Buddy Stacks: Protecting Return Addresses with Efficient Thread-Local Storage and Runtime Re-Randomization. *ACM Transactions on Software Engineering and Methodology (TOSEM '22)* 31, 2, 1–37. https://doi.org/10.1145/3494516

Received 2023-03-16; accepted 2023-04-21