SIDECAR: Leveraging Debugging Extensions in Commodity Processors to Secure Software

Konstantinos Kleftogiorgos[1], Patrick Zielinski[1], Shan Huang[1], Jun Xu[2], Georgios Portokalidis[3]
[1]Stevens Institute of Technology, Hoboken, NJ, USA
[2]University of Utah, Salt Lake City, UT, USA
[3]IMDEA Software Institute, Madrid, Spain

***Abstract*—The increased parallelism in modern processors has sparked interest in offloading security policy enforcement to processes or hardware operating in parallel with the main application. This approach can reduce application latency, enhance security, and improve compatibility. However, existing software solutions often incur high overheads and are susceptible to memory corruption attacks, while hardware solutions tend to be inflexible and require substantial modifications to the processor. In this paper, we present SIDECAR, a novel approach that offloads security checks to run concurrently with applications by leveraging the debugging infrastructure available in commodity processors. Specifically, we utilize software-driven logging (SDL) extensions in Intel and Arm processors to create secure, append-only channels between applications and security monitors. We build and evaluate a prototype of SIDECAR for the x86-64 and Aarch64 architectures. To demonstrate its utility, we adapt well-known security defenses within SIDECAR, providing control-flow integrity (CFI), shadow call stacks (SCS), and memory error checking (ASAN). Our evaluation shows that these extensions perform better on the Intel architecture. In terms of defenses, SIDECAR reduces the latency of CFI in the tested real-world applications by an average of 30%, offers enhanced security with similar overhead for SCS, and is versatile enough to support complex defenses like ASAN. Furthermore, our security monitor for CFI+SCS is 30 times more efficient compared to previous work.**

1. Introduction

Software often contains bugs resulting in undefined behavior [1]. Systems software, predominantly authored in memory-unsafe languages such as C and C++, specifically suffers memory-related errors [2], [3], such as overflows and use-after-free (UAF) bugs, leading to severe security issues such as arbitrary code execution [4], [5], [6], [7], [8], [9], [10], data leaks [11], and privilege escalation [12]. To effectively detect or mitigate the exploitation of these bugs, prior works [13], [14], [15], [16], [17], [18], [19], [20] have introduced a variety of approaches for introducing security checks into programs at compile or run time, unfortunately not without problems. First, they can impose high memory and performance overheads [14]. In other cases [20], storing runtime metadata in the same address space as the application allows the subversion of the defense. Finally, some approaches [21], [22], [23], [24] offer reduced compatibility, e.g., by requiring recompilation of *all* software components (e.g., libraries), leading to slow (or no) adoption.

A new paradigm for addressing these issues is to offload security operations to spare cores, which exploits the increasing parallelism available in multi-core processors. Software-only approaches [25], [26], [27], [28], [29] in this direction involve modestly instrumenting applications to relay runtime state to a parallel-running *monitor* process. Despite performance improvements, they still induce high overheads and face security issues of their own, as they employ shared memory for transferring data, which can be corrupted before being processed by the monitor.

On the other hand, hardware approaches [30], [31], [32], [33], [34], [35] offer strong isolation by introducing new processor designs that automatically log runtime data for use by separate monitor process(es) or core(s). However, they necessitate substantial microarchitectural alterations, which has obstructed their adoption by manufacturers. Works leveraging *existing* hardware features, like execution tracing [36], [37], [38], [39], [40], [41], [42] can be immediately deployed, but are not versatile in terms of policies they can support and require significant processing resources for the monitor due to the high volume of data produced by the processor.

This paper presents SIDECAR, a system that leverages a less explored functionality of processor debugging extensions to offload security operations from applications to spare cores. Specifically, we utilize *software-driven logging* (SDL) capabilities provided by Intel's Processor Trace (PT) and Arm's CoreSight architectures through the `PTWRITE` instruction and the System Trace Macrocell (STM). SDL enables programs to log data in an append-only manner, either to an operating system-managed memory queue or directly to the hardware input port of a co-processor. These extensions are *safer* than software-only approaches that rely on shared memory-based communication. Additionally, they are *versatile* due to their software-driven nature, allowing support for a broad range of security policies. Furthermore, they can be *efficient* as software controls what information is logged, thereby reducing the load on the security monitor.

To demonstrate the above, we implemented SIDECAR for x86-64 and Aarch64 Linux systems. SIDECAR provides

secure, append-only queues between C/C++ applications and monitors processes in an efficient manner. Our design accommodates both architectures, which differ in certain key aspects, like how data are logged and stored. To show the performance, security, and versatility of SIDECAR, we modify the designs of three well-established defenses that modify and harden applications at compile time, i.e., LLVM's CFI, SCS, and ASAN, to offload their operations to a monitor process. We also design and build monitors for each of these defenses.

We evaluate SIDECAR in terms of security and correctness, and performance. We use LLVM's testing environment [43] and the RIPE64 [44] security benchmark to establish that the offloaded versions of the defenses, coined SIDECFI, SIDESTACK, and SIDEASAN, operate correctly and capture the same attacks as their inlined versions. Note that SIDESTACK by design does not suffer the weaknesses of LLVM's SCS. We evaluate performance using SPEC CPU2006, SPEC CPU2017 and a set of real-world applications and servers (Apache, Bind, Lighttpd, memcached, and Chromium). We find that, on average, SIDECFI outperforms LLVM-CFI by 30%. Finally, we evaluate the resources required by the monitors and find that they are lightweight. Compared with prior work, GRIFFIN [37], which employed execution tracing extensions and required six cores to monitor a single application thread, SIDECAR requires approximately 20% of a core to monitor a single application thread (average over SPEC CPU2006). Running ASAN over SIDECAR, demonstrates that versatility of the approach, but it also revealed the bandwidth limits of SDL in current processors as it incurs higher overheads.

In summary, the main contributions of this paper are:

- We study two processor architectures with software-driven logging facilities: Intel PT and Arm CoreSight, used to create secure and efficient inter-process communication channels.
- We design and build SIDECAR on x86-64 and Aarch64 Linux, a system that leverages SDL to allow the offloading of security checks from applications to monitor processes running in parallel.
- We develop three defenses that apply security checks in parallel to the application, offering control-flow integrity (CFI), shadow call stacks (SCS), and memory error detection (ASAN). They consist of an LLVM compiler-based component that modifies programs to log the data necessary to apply the check in parallel, and a monitor process that receives the data and enforces the relevant security policies.
- We evaluate SDL on a 24-core Intel i9-13900K and a quad-core ARM Cortex A53, and find that the Intel processor offers significantly better performance.
- We evaluate SIDECAR and its defenses on the Intel architecture using the SPEC CPU2017 and SPEC CPU2006 integer benchmarks, along with five real-world applications and servers. The results are compared against their inlined counterparts, including LLVM's CFI [45], SCS [46], and ASAN [14], as well as prior works, FineIBT [47] and GRIFFIN [37].

The rest of the paper is organized as follows. §2 provides background information and highlights the advantages of decoupling security checks from applications using processor debugging extensions. The threat model for SIDECAR is in §3. Its design and implementation are presented in §4, while §5 describes the defenses we developed on top of it. The evaluation of SIDECAR is in §6.

2. Offloading Security Checks with SIDECAR

2.1. Overarching Concept

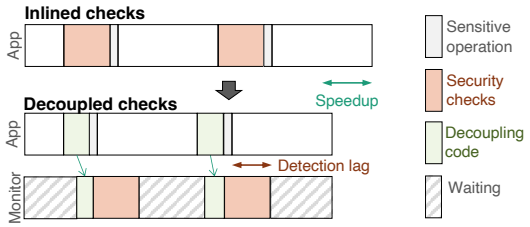

Figure 1: Decoupling security checks from execution.

Figure 1 illustrates decoupling security checks from execution. Instead of injecting checks before sensitive operations (e.g., an indirect call or memory access) into the application code, these checks are moved to a separate monitor process running in parallel. The application is instrumented with lightweight operations that transfer the necessary runtime data to the monitor. This can reduce runtime overhead if the decoupling code executes faster than the checks it replaces. However, this approach introduces a trade-off: security policy enforcement becomes asynchronous, creating a small detection lag between a violation and its detection. This lag can be eliminated by making the application wait for the monitor to complete all checks before proceeding with sensitive operations, like a system call, but this sacrifices some performance gains.

Prior work in this area can be classified in software [25], [26], [27], [28], [29] and hardware [34], [32], [35], [30], [31], [48], [33] approaches. Software solutions are frequently policy specific and transfer data through shared memory buffers, suffering the risk of tampering by an adversary before the checks are performed. Hardware approaches, on the other hand, face adoption hurdles in high-performance processors.

2.2. Using Processor Debugging Extensions

SIDECAR leverages processor debugging extensions to transfer data from the application to monitors. Such extensions are already available in modern processors and, hence, *do not require any hardware modifications*. Currently, both Intel and ARM provide them under the names Intel Processor Trace (PT) [49] and ARM CoreSight (CoreSight) [50], respectively. One of the facilities provided by these extensions is transparent execution tracing, where the processor automatically logs every non-deterministic control-flow

transfer. This capability has received ample attention by past works [36], [37], [38], [39], [40], [41], [42]. However, even though it imposes minimal overhead *(<4%)* on applications, the data produced are voluminous and require significant processing resources for analysis, while *it can only support policies related to control flow.*

SIDECAR instead leverages software-driven logging (SDL). It allows the programs being traced to inject custom data into the trace stream, even when execution tracing is not active. CoreSight utilizes the System Trace Macrocell (STM) and memory-mapped I/O (MMIO) ports for this purpose, where a program can write data to be appended in the stream, whereas PT introduces the `PTWRITE` instruction [51] for the same purpose. The logged data are stored in the same manner as with execution tracing, which, however, does not need to be active. The data written by each core are stored into a memory buffer that can be accessed by the operating system (OS) kernel. Alternatively, some systems (mainly Arm) allow you to directly route data to a hardware port for processing by a co-processor, like a Field Programmable Gate Array (FPGA). This essentially creates *a secure, append-only channel for sending data from the application to a monitor.*

TABLE 1: Comparison of related work with SIDECAR.

Approach	Availab.	Tamper Safe	Versatility
ShadowReplica [25]	✓	✗	✗
TaintPipe [26]	✓	✗	✗
StraightTaint [27]	✓	✗	✗
CAB [28]	✓	✗	✗
PIPA [29]	✓	✗	✗
LBA [30], [31]	✗	✓	✓
FlexCore [32], [33]	✗	✓	✓
GuardianCouncil [34]	✗	✓	✓
HERQULES [35]	✗	✓	✓
GRIFFIN [37]	✓	✓	✗
SIDECAR (this paper)	✓	✓	✓

The use of SDL extensions has been underexplored. In this paper, we explore the capabilities and challenges of using this feature to offload security checks and improve existing inlined defenses. *SDL systems and by extension* SIDECAR *exhibits the following key properties for decoupling security checks:*

Property 1: Availability. The extensions are already present in modern processors.

Property 2: Tamper Safe. The data are logged in append-only fashion outside the application's memory space and are accessible only by the kernel.

Property 3: Versatility. The software controls data logging, supporting the implementation of different types of policies.

Compared with prior work, SIDECAR is the only approach to concurrently have all these properties, as highlighted in Table 1.

2.3. Benefits

SIDECAR can generally reduce application latency by offloading security checks to a parallel monitor. However, there are other important benefits too. We examine three popular defenses, CFI, SCS, and ASAN to highlight them.

Benefit 1: Hiding Latency of Slow Checks. CFI [52], [53], [54], [55] is a popular defense that aims to ensure that the control flow of a program remains within its valid control flow graph (CFG). Forward-edge CFI [55], [53], specifically, focuses on indirect calls (icall) and jumps (ijmp) that use operands from (corruptible) memory. These correspond to calls using function pointers, virtual method calls in C++, and even jump tables, such as the ones generated by the compiler to implement `switch` statements. An efficient and popular implementation of forward CFI is currently offered by the Clang/LLVM toolchain [45] (LLVM-CFI), which can ensure that indirect calls target functions of the same type as the call site.

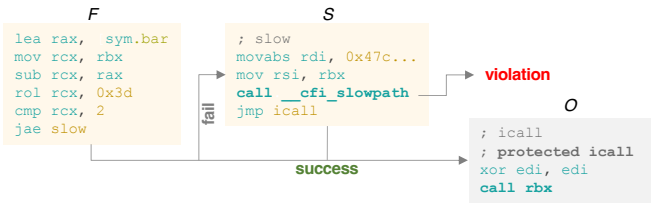

Figure 2: An indirect function call with LLVM-CFI checks.

Figure 2 depicts the code introduced for an icall (`call rbx`) on x86-64 in code block *O*. By carefully positioning functions of the same type together, LLVM-CFI simplifies the check to a range and alignment check, as shown in block *F*. Offloading such checks may deliver small gains, however, to support all transfers to dynamically shared objects (DSOs), it also introduces a slow path that performs a more thorough validation, as shown in block *S*, which is executed when the fast check fails. The cost of slow paths can vary depending on the valid function targets in a DSO, as confirmed by our micro-benchmarks comparing intra-DSO and cross-DSO function calls. Fast path checks averaged ∼130 cycles (∼1000 cycles cold), while slow path checks ranged from 170 to 500 cycles (up to 3000 cycles cold). With SIDECAR, we can offload all or *only slow path checks*, depending on the application, to *hide the latency of checks*.

Benefit 2: Improved Security. SCS [20], [56] is an established defense for detecting corruption of return addresses on the stack and enforcing the control-flow integrity of *backward-edges*. Multiple works have shown that it is a necessary compliment to CFI [6], [9], [52]. The basic idea is to maintain a shadow stack, where the return addresses are duplicated, and their integrity is checked upon function return. While there have been different implementations, the Clang/LLVM implementation [46] has the advantage that it can be deployed incrementally and works even when some

components (e.g., libraries) have not been compiled with SCS support.

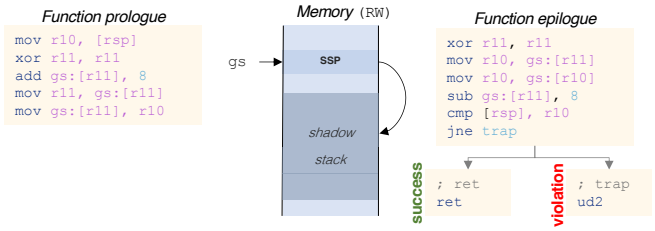

Figure 3: Operations added at function prologue and epilogue by LLVM-SCS to maintain a shadow stack.

Figure 3 shows the code introduced by LLVM-SCS at each function prologue and epilogue. At function entry, the shadow stack pointer (SSP) is accessed using the `gs` register to *push* a copy of the return address in the shadow stack. At function exit, the SSP is accessed again to *pop* and *check* the return address with the one stored in the program stack. Both the shadow stack and the SSP are, however, stored in the application's memory space, making them vulnerable to corruption attacks [57], [58]. With SIDECAR, SCS metadata are stored *outside* the application offering *increased security*.

Benefit 3: Incremental Deployment. Control-flow enforcement technology (CET) [59] by Intel supports CFI but is weaker than LLVM-CFI. Recent work [47] enhances CET to match LLVM-CFI. CET adds a new instruction, `endbr`, in the prologue of every function (Fig. 4), ensuring that the instruction following an indirect call or jump (`ijmp`) is an `endbr`. While this is efficient, if any function called indirectly has not been build with CET support, it causes a violation and program termination. To avoid this, loaders (e.g., Linux) disable CET if any module (e.g., shared library) doesn't support it.

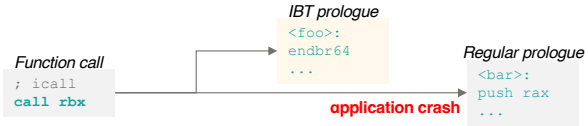

Figure 4: An indirect call targeting functions with and without Intel CET support.

CET's shadow stack support also faces compatibility issues with libraries not built with CET, particularly when exceptions or functions `setjmp/longjmp` are used. In contrast, SIDECAR supports *incremental deployment* of defenses, allowing the monitor to check module support for CFI or SCS and act accordingly (e.g., unwind the shadow stack or allow the call). Such actions are *prohibitively expensive to inline* in application code.

Benefit 4: Supports Complex Policies. Address sanitizer [14] (ASAN) is a popular sanitizer that tracks memory allocations during the program's lifetime using metadata stored in a large shadow memory area to detect invalid memory accesses. As shown in Fig. 5, when a heap memory

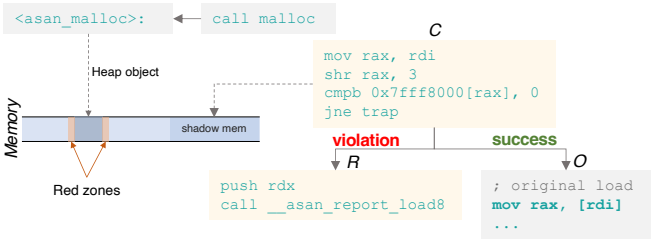

Figure 5: ASAN detecting invalid memory accesses.

object is allocated, red zones are also allocated around it and shadow memory is updated accordingly. Every memory access, such as the load in block *O*, is instrumented during compilation with a check, like the one in block *C*. It consults shadow memory to determine if a red zone is accessed, in which case it detects a violation and reports the error (block *R*). Stack and global memory objects are similarly modified and red zones added around them.

Because every memory access is instrumented, ASAN can be very slow, so currently it is mostly used in fuzzing [60]. Moreover, it includes a multitude of operations, as it performs many different types of operations to keep track of memory usage and update shadow memory, which cannot be decoupled by domain specific approaches [25], [37]. However, SIDECAR is *highly versatile* and allows us to design elaborate protocols for *supporting complex defenses and policies*.

3. Threat Model

SIDECAR provides a general-purpose framework for decoupling security from applications and running them in parallel. We assume a strong adversary capable of arbitrary memory reads and writes through application vulnerabilities, potentially leading to arbitrary or remote code execution (ACE/RCE) [5], [4], [7], [9], [10]. However, we assume the adversary cannot compromise the hardware or OS [61], as protecting these is beyond our scope. Between application compromise and detection by the monitor, the adversary may send bogus data to the monitor, but cannot tamper with already sent data due to the append-only communication channel, ensuring eventual attack detection.

Our defense mechanisms and security policies focus on C/C++ applications that suffer from one or more memory errors [3], [2]. We assume a contemporary OS supporting non-executable memory [62] and enforcing the W^X policy [63] (i.e., no memory page can be both writable & executable). This prevents existing code alteration and new code injection before the monitor can detect control by the adversary. While security mechanisms like ASLR [64], stack-smashing protection [65], and code randomization techniques [66], [67], [68] may be applied, they are not required for SIDECAR's operation.

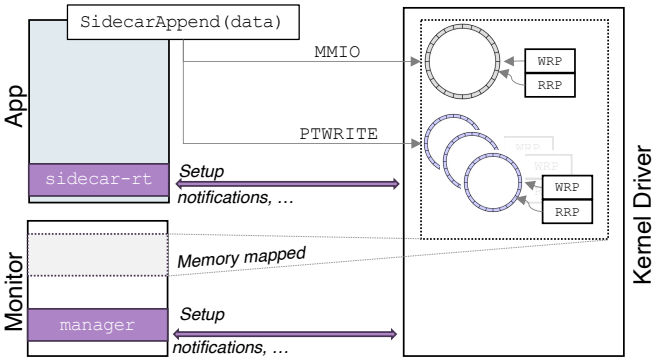

Figure 6: Overview of SIDECAR.

4. SIDECAR Design and Implementation

Figure 6 illustrates the high-level design of SIDECAR. Applications communicate with a driver to activate processor extensions and send data to the security monitor running in parallel. Data transmission is handled through a new primitive, `SidecarAppend`, which provides append-only logging, tamper-resistant logs, and support for logging arbitrary and thread-contextual data. Monitor processes communicate with the driver to access the logged data, send violation notifications, and perform management tasks like sequencing multithreaded operations or spawning threads when needed. We have implemented a prototype of SIDECAR for x86-64 and Aarch64, described in detail below.

4.1. Application Interface

An application that wishes to make use of a security monitor must first communicate with the driver. The setup serves two purposes: (i) enabling the debugging extensions and allocating the necessary memory in the kernel for storing the trace data, and (ii) enabling the `SidecarAppend` primitive for the application. To perform these two tasks, we introduce a runtime library, `sidecar-rt`, that applications can link against.

Setup is transparent to the application and essentially involves calling an initialization function that communicates with the driver using an IOCTL operation. However, depending on the architecture the `SidecarAppend` primitive is implemented differently. On Intel systems, it simply corresponds to the `PTWRITE` instruction, which can be used after completing setup. On the other hand, on Arm writing to the trace is done through memory-mapped I/O (MMIO) registers. Specifically, a number of *ports* are offered over MMIO. There are some minor differences between the ports available in different ARM systems, but they essentially offer the same functionality, appending the number of the port and the data written to it to the trace. So on Arm systems, the runtime also maps the corresponding STM's MMIO-mapped ports to the application's address space. `SidecarAppend` then corresponds to a a simple store operation: `*((uint64_t *)STIM_PORT_ADDR) = data;`

Finally, the runtime library is also responsible for notifying the driver when the application spawns a new thread or process. This is done by interposing on known APIs, like `pthread_create` of POSIX threads and `fork` and notifying the driver through an IOCTL operation.

4.2. Trace Collection

Both PT and CoreSight can be configured to store data in ring buffers, essentially, a sequence of physical pages that are treated as a contiguous ring buffer by the processor. These are allocated and configured by the SIDECAR driver, when an application requests its services. However, PT uses a separate ring buffer for each core, while CoreSight uses a single buffer, shared by all cores. Here we explain how SIDECAR handles these two disparate designs, as well as common aspects between the two.

Intel PT. When a *new* application is launched, SIDECAR allocates a new ring buffer and associates it with the core the application thread is currently running on. At that point, we need to either pin the application thread to that specific core or hook the scheduler to track when the thread migrates to another core, so that we disassociate the buffer from the old core and associate it with the new one. We have implemented the first option, but he second option could also be implemented with moderate engineering effort [69]. When additional threads or applications are launched, the driver is notified and a new ring buffer is allocated and associated with that thread and core. This way we have a one-to-one mapping between threads and ring buffers.

Arm CoreSight. When the *first* application is launched, SIDECAR allocates a single ring buffer and attaches it to the CPU. All cores use this same buffer for logging data. To differentiate the originating thread of data in the buffer, we use the *context ID* registers, which can be set per thread to a unique value identifying the thread. The processor adds a special data packet with the context ID of the thread that generated the data for each write. When additional threads or applications are launched, the driver is notified and the context ID register of that thread is set to a unique value. This way, data within the buffer can be associated with their originating thread.

Tracking Ring Buffer Status. Tracking buffer status is crucial for SIDECAR to prevent data loss when high system load risks the processor overwriting unprocessed data. SIDECAR uses two pointers: the read ring buffer pointer (RRP) and the write ring buffer pointer (RWP). Both architectures provide a hardware RWP (via MMIO on Arm and MSR on Intel), but SIDECAR maintains a software copy, updated by interrupts when a buffer page fills. When threads or applications exit, SIDECAR updates this copy and notifies the monitor. The RRP, managed in software, indicates the next byte to process. Monitor threads poll the RWP at intervals to process data between RRP and RWP. To keep the RWP from overtaking the RRP, the driver uses interrupts and temporarily halts threads if buffer space drops below two pages, resuming them when space becomes available.

4.3. Monitor Interface

Security monitors communicate with the driver to register and gain access to the data logged by applications. They do not need to run as privileged processes, after opening the driver. To facilitate the creation of monitors, we provide a Manager library for performing common tasks. At startup, the Manager communicates with the driver to notify it of its existence and waits for events, through a custom IOCTL operation. Once an application is launched, a message is received and the Manager spawns a new monitor thread to start processing its data. While monitor threads are running, the Manager still listens for other events from the driver. Besides the launch of a new application, this may include events about new threads or processes spawned by processes already monitored. An IOCTL is also used by monitor threads to report a security violation to the driver, which can then take appropriate action and terminate the application. If no violations are captured, the Manager is notified when the application exits through a signal from SIDECAR, after the latter ensures that all trace data has been consumed by the monitor threads.

The monitor Manager also needs to accommodate the architectural differences between Intel and Arm, which we present below.

Intel PT. Whenever a new application or thread is spawned, the Manager maps its ring buffer and RWP/RRP pointers into the monitor's address space. The RWP is mapped read-only and updated only by the kernel driver. Creating a new monitor thread for each application/thread depends on the specific monitor, but on Intel systems, it's straightforward to use a different thread for each buffer, even if they run on the same core. The monitor updates the RRP after processing chunks of data, not after each byte, to avoid performance degradation due to cache coherency protocols in multi-core processors.

Arm CoreSight. Since there is only one buffer, this needs to only be mapped once in the monitor's address space. Again, whether one or more monitor threads will be used depends on the monitor implementation. The SIDECAR Manager offers functionality to de-multiplex the data in the buffer based on the context ID of the thread that produced it and store it in separate buffers. This way, each monitor thread can operate on its own buffer, but memory copying can put a strain on the memory bus.

Trace Decoding. Both CoreSight and PT use distinct encodings for logged data. Although open-source decoding libraries exist [70], [71], we implemented our own decoders to enhance performance. The Manager library handles decoding, optimized for speed by focusing on specific trace packets and their operation codes. Our PT decoder, a streamlined, manually inlined version of Intel's official decoder, processes data in a single for-loop with switch cases, significantly speeding up the decoding process compared to Intel's function-based structure.

5. Redesigning Sanitizers for SIDECAR

This section outlines the security policies we developed for SIDECAR, integrated within the LLVM toolchain (v12) and comprising roughly 3,600 lines of code. Our prototype redesigns LLVM's runtime sanitizers by offloading security checks to a monitoring thread.

5.1. SIDECFI: Forward-Edge CFI

LLVM-CFI (Figure 2) enforces checks on indirect and virtual calls to ensure that the function invoked at runtime matches the expected type (i.e., has the same signature) as the call site. Each indirect call is assigned a Type ID based on its signature, and the same applies to each function referenced by a pointer. For C++ virtual calls, the Type ID corresponds to the dynamic type of the object used in the call. These checks ensure that the Type ID assigned to the call site aligns with the Type ID of the target function or object, thus enforcing type safety during execution.

SIDECFI builds on this concept by introducing a hashed version of the LLVM Type ID, compressed into 14 bits to fit within the 64-bit `SidecarAppend` primitive for efficient transfer. The hash is calculated through a series of bitwise operations that reduce the original 64-bit Type ID. This approach allows SIDECFI to replace LLVM's inline CFI checks with instrumentation that logs the hashed 14-bit Type ID along with the 48-bit target address of indirect and virtual calls. For indirect calls, the target is the function address, while for virtual calls, it is the vTable address corresponding to the dynamic type. `SidecarAppend` is inserted during Link-Time Optimization (LTO), effectively replacing the original LLVM-CFI check instrumentation and enabling a decoupled enforcement mechanism.

To enforce these checks, the SIDECFI monitor requires metadata about the allowed targets and their Type IDs. During compilation, the offset of each function or vTable is extracted, along with its 14-bit hashed Type ID, and stored in a typemap file associated with each binary object. A SHA-1 hash is generated for each typemap file to verify its validity. SIDECFI intercepts 'dlopen' and 'dlclose' system calls to identify the loading and unloading of DSOs and to extract their base addresses, which are needed to transform function offsets into runtime addresses. Whenever a new DSO is loaded at runtime, SIDECAR is notified and provided with the path to the binary and its base address. This information is then passed to the SIDECFI monitor, which uses it to locate the corresponding typemap file and calculate the runtime target addresses using the base address. The monitor loads this metadata into an internal hashtable. As the monitor receives runtime packets from the `SidecarAppend` primitive, it verifies that the target's Type ID matches the expected values stored in its hashtable. Any mismatches indicate a violation, which are reported to SIDECAR, prompting the application to be stopped. When a DSO is unloaded, the monitor is notified, and the corresponding entries are removed, disallowing those targets and their Type IDs.

5.2. SIDESTACK: Shadow Call Stack

SCS (Figure 3) provides backward-edge protection by maintaining a shadow stack that mirrors the application's stack. SIDESTACK decouples SCS by offloading shadow stack management to a monitoring thread, which requires runtime metadata for each function's return address. At each function prologue, the return address is logged using the 32-bit `SidecarAppend` primitive and sent to the monitor. Similarly, it is logged and verified at each epilogue. To reduce bus load, we compress the 48-bit return address by XORing the upper 18 bits with the lower 30 bits, resulting in a 30-bit hash. This `SidecarAppend` replaces the original SCS push and pop instructions during Link-Time Optimization (LTO). The monitor uses this runtime metadata to maintain its own shadow stack, ensuring return address integrity.

A key feature of SIDESTACK is its ability to handle discrepancies between the program's return address and the shadow stack's top hash. Such mismatches can occur during C++ exception handling, which requires stack unwinding. In these cases, the monitor searches the shadow stack for a matching hash to adjust the stack correctly. If no matching hash is found, SIDESTACK raises an alert to indicate a potential security violation, signaling that the return address may have been tampered with or altered unexpectedly.

To optimize performance, SIDESTACK uses a "hybrid" mode for leaf functions, where the return address is stored in a free register upon entry and compared with the stack's top address upon exit. Non-leaf functions and leaf functions lacking a free register use the original instrumentation.

5.3. SIDEASAN: Address Sanitizer

ASAN (Figure 5) detects memory errors by maintaining a shadow memory and instrumenting memory accesses to update it and perform checks. SIDEASAN decouples ASAN by offloading shadow memory operations to a monitoring thread. To facilitate the decoupling of ASAN's shadow memory, metadata including memory addresses, access sizes, operation types (read/write), details about memory allocation and deallocation, heap and stack poisoning, and other memory-related operations need to be transferred to the monitor. It is important to mention that red zones are maintained in the application's address space, but their poisoning and validation operations are offloaded to the monitor.

SIDEASAN modifies the ASAN LLVM pass and runtime library to offload shadow memory operations to a monitoring thread, encoding them into messages using `SidecarAppend`. Specialized messages handle frequent small memory accesses to reduce overhead. At runtime, the monitor processes these messages, updating shadow memory for heap and stack operations, poisoning and validating accesses, and maintaining red zone bits. Use-after-free errors are detected by marking freed memory as non-addressable. Violations trigger alerts to SIDECAR, which halts application execution.

To support multithreaded applications, SIDEASAN employs dedicated software monitor threads for each application thread. It is important to note that all monitor threads share a single shadow memory and thus special care must be taken to avoid false positives by ensuring the orderly execution of ASAN poisoning and validation checks. SIDEASAN achieves this by enabling timestamps in fixed intervals, that allow monitor thread operations to synchronize with each other using semaphores.

6. Evaluation

The evaluation aims to answer the following questions: *(i) How does SDL perform? (ii) How effective are the defenses implemented in* SIDECAR*? (iii) Does it reduce application latency caused by defenses?*

We have implemented SIDECAR for both Intel and Arm architectures and use the following three platforms in the evaluation: Two workstations: a 24-core Intel i9-13900K at 5.20 GHz with 128GB DDR4 RAM and a Samsung SSD 980 PRO 1TB (**intel**) and a 16-Core Intel i9-12900K at 5.20 GH with 128GB DDR4 RAM and 2 x 1.92 TB NVMe SSD (**intel2**), both running Debian 11 and with hyper-threading disabled. An Arm development board (db410c) with a 4-core ARM Cortex A53 at 1.2 GHz with 1GB LPDDR3 RAM and 8GB eMMC 4.5 (**Arm**), running Linaro Linux 5.10.7. The benchmarks we use are: the integer benchmarks in the SPEC CPU2017 and SPEC CPU2006 suites, real-world applications (Chromium v90.0.4396.0, Apache Httpd v2.4.58, Lighttpd v1.4.76, Memcached v1.6.9, and Bind v9.19.24), and the utilities pbzip2 v1.1.13 and pigz v2.8.0.

6.1. SDL on Intel vs. Arm

To answer the first question, we run a micro-benchmark measuring the average CPU cycles for a `SidecarAppend` operation on STM and PT, as we increase the number of consecutive operations. Cycle counts are collected via `pmccntr_el0` on Arm and `rdtsc` on Intel, comparing results to a standard memory-backed circular buffer. The tests are conducted on Intel 2 and Arm. Intel processors have *performance* (P-cores) and *efficiency* (E-cores) cores, where P-cores support hyper-threading and higher energy usage, and E-cores prioritize efficiency.

Results for Arm and Intel P-cores are shown in Figure 7. Initially (<16 writes), average duration decreases as the cycle counter overhead is spread across more operations.

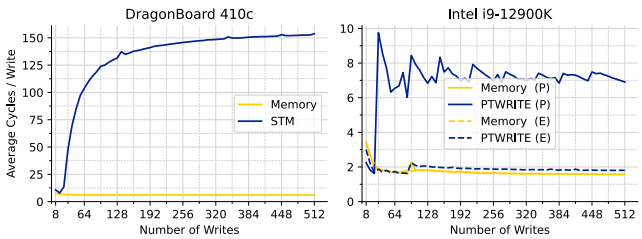

Figure 7: Comparison of SDL on Arm and Intel.

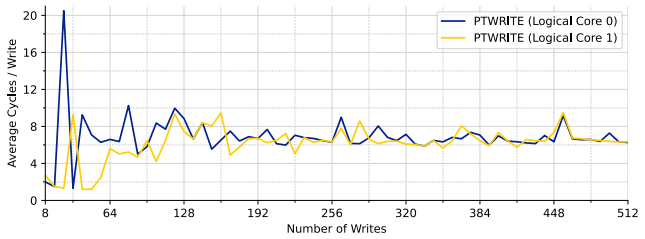

Figure 8: Logical cores sharing the same P-core performing simultaneous `PTWRITE` operations.

On Arm, the cost grows logarithmically with more writes, while on Intel P-cores, it stabilizes at ≈7 cycles after a jump at 32 writes. This behavior likely reflects the size of internal buffers, preventing data loss. On Intel E-cores, performance matches regular memory writes, but data loss occurs beyond 64 consecutive writes.

We also examined `PTWRITE` contention on two logical cores sharing a physical core, as shown in Figure 8. Throughput stabilized, but with higher spike magnitudes, indicating minor contention due to shared PT hardware. Given Arm 's high overhead and data loss on E-cores, we focus the rest of the evaluation on P-cores with hyper-threading disabled to minimize noise from shared PT hardware.

6.2. Security and Correctness

To evaluate the effectiveness of SIDECAR defenses, we need to show: (i) that *they retain detection capabilities*, and (ii) that *no new vulnerabilities are introduced*.

6.2.1. Detection Capabilities

To test the ability to detect attacks, we use the RIPE64 security benchmark [44] to test SIDECFI and SIDESTACK, and the LLVM Integrated Tester (LIT) [43] to test SIDEASAN.

RIPE64. The RIPE64 suite extends the original RIPE security suite [72] for evaluating security systems designed to mitigate buffer overflow exploits, with Address Space Layout Randomization (ASLR) [64] enabled. We tested the suite with SIDECFI, SIDESTACK, LLVM-CFI, and LLVM-SCS defenses to assess their effectiveness. Out of all the attacks in the RIPE64 suite, we specifically selected 1,350 forward-edge control flow attacks. LLVM-CFI successfully detected all of these attacks, and we used this as a baseline to confirm that SIDECFI effectively captured all 1,350 attacks as well, demonstrating the equivalent protection provided by SIDECFI. Additionally, LLVM-SCS and SIDESTACK both stopped 154 backward-edge attacks, such as Return-Oriented Programming (ROP) and return-to-libc.

LLVM Integrated Tester. LIT includes tests to verify ASAN functionality. We selected 150 tests focused on ASAN's core features, excluding tests related to other sanitizers like LSan [73] or features not related to bug detection. Since ASAN functionality now runs on the monitor, we adapted the tests to check for errors detected by the monitor, modifying the test files to have FileCheck [74] verify the monitor's output. Our results show that SIDEASAN captures all bugs detected by ASAN, demonstrating equivalent effectiveness in bug detection.

During these tests and throughout the performance evaluation (§6.3), no erroneous alerts (false positives) were reported.

6.2.2. Using Pointer Hashes

In contrast to LLVM-SCS, our approach relocates the shadow stack to a separate monitoring process, significantly enhancing security by isolating it from potential corruption by attackers. SIDESTACK transmits a hash of the return address to the monitor instead of the full address, which, in theory, could be susceptible to exploitation through hash collisions. To evaluate this risk, we used ROPgadget [75] to extract all gadgets from the SPEC CPU2017 benchmark binaries and executed all benchmarks with SIDESTACK, collecting the return address hashes. We then compared these hashes to identify any gadgets that shared a hash with legitimate return addresses. No collisions were found across all benchmarks, demonstrating the reliability and security of the chosen hash function.

6.2.3. Detection Lag

The asynchronous nature of SIDECAR allows a compromised application a brief period to perform malicious actions before termination by the monitor. To measure this window, we conduct a micro-benchmark on the Intel platform where an application repeatedly obtains the time using `gettimeofday` and sends it to the monitor with `SidecarAppend`. The monitor timestamps each received message using `gettimeofday` and compares it with the sent timestamp. We repeat this experiment with an increasing number of consecutive messages to assess the impact of bus congestion on the detection lag.

Figure 9 presents the results, showing the window ranges from 1.9 to 2.4 milliseconds. Although not insignificant, persistent changes to the system can be quickly rolled back. Alternatively, the application could wait for the monitor before executing risky operations, such as running a new program or writing to system files.

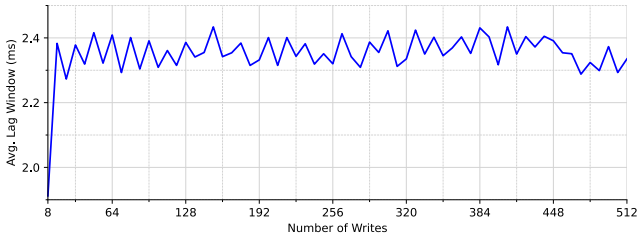

Figure 9: Average lag in message reception and processing by the monitor on the Intel platform.

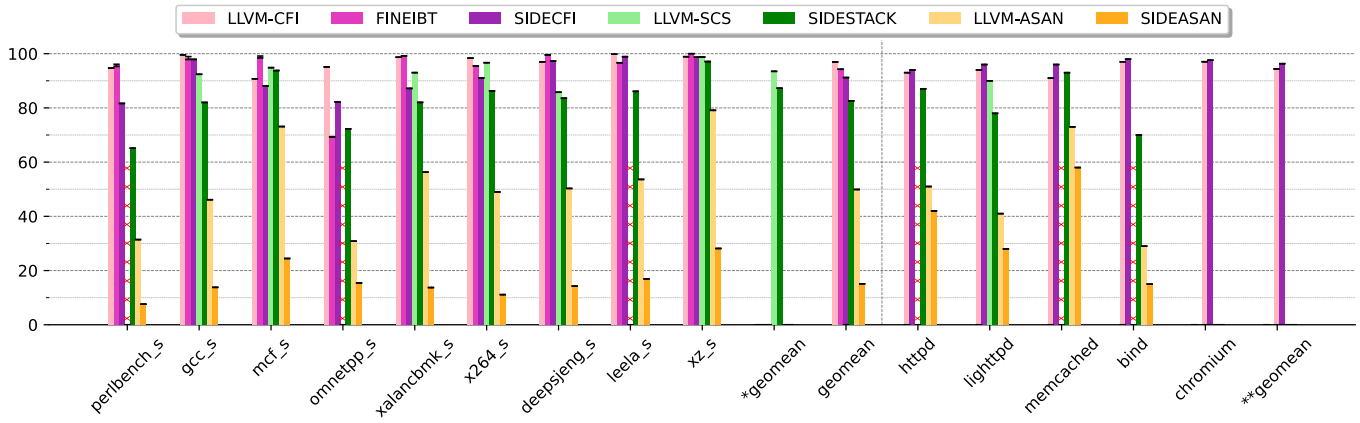

Figure 10: Relative (%) performance comparison under SIDECAR policies and related approaches.
*Geomean including only benchmarks that executed successfully for LLVM-SCS.
**Geomean of LLVM-CFI and SIDECFI for real-world applications.

6.3. Performance

To address the third question, we assess SIDECAR defenses on the Intel platform in terms of benchmark application runtime overhead. For CFI, we compare against LLVM-CFI and FineIBT [47], a recent approach utilizing Intel's CET processor extensions. FineIBT has demonstrated superior performance to LLVM-CFI, albeit lacking incremental deployment support, requiring recompilation of all software components. For SCS, we compare against LLVM-SCS[1], while we compare SIDEASAN against vanilla ASAN. We evaluate ASAN on multithreaded applications, using the `pbzip2` and `pigz` utilities. Finally, we evaluate against GRIFFIN [37], a prior approach utilizing another feature in Intel's debugging extensions, execution tracing. GRIFFIN combines CFI and SCS, so we concurrently apply both defenses as well with SIDECAR. We compare it both in terms of application latency and monitor resource consumption.

Setup. We utilized SPEC CPU2017 (SPECspeed Integer) with the "ref" workload to evaluate the runtime performance of SIDECAR. For Httpd, we used two `wrk` [76] threads with four connections each to generate HTTPS requests continuously for 60 seconds, serving 100KB files filled with random data. Similarly, for Lighttpd, we used two `wrk` threads with four connections for 60 seconds, serving 100KB files. For Memcached, we used two `memtier` [77] threads with four connections each to continuously send 32B GET requests over 60 seconds. For Bind, we used a single thread of `dnsperf` [78] to query a specific URL address, using a simple URL to minimize string processing time and increase the number of requests handled. Chromium officially supports enabling Clang's CFI checks on indirect and virtual calls for intra-DSO checks only. To support cross-DSO checks, we modified the GN [79] flags to pass custom build flags, including `-fsanitize-cfi-cross-dso`, to ninja [80], and executed the Dromaeo benchmark from the Telemetry [81] suite that ships with Chromium. All experiments were performed 20 times, with averages calculated over an uninstrumented baseline. Applications ran on the same host, connected via the loopback (lo) virtual network interface to minimize I/O latency. Each application was limited to a single core, and benchmarks were configured to ensure full core utilization to avoid I/O masking the overhead.

Figure 10 shows the relative performance against an unprotected baseline, with error bars indicating the standard deviations. The figure categorizes mechanisms visually with a color scheme: forward-edge mechanisms in shades of purple, backward-edge in green, and sanitizers in yellow.

6.3.1. Control-flow Integrity

On SPEC CPU2017, LLVM-CFI shows a geomean of 96.94%, with FineIBT at 94.26%, and SIDECFI at 91.20%. SIDECFI's performance is consistently on par with the others, demonstrating its effectiveness across a wide range of benchmarks. In real-world applications, SIDECFI outperforms LLVM-CFI in almost all tests, achieving a geomean performance of 96.31% compared to LLVM-CFI's 94.38%. This improvement is largely due to SIDECFI replacing the slow paths required for cross-DSO calls, which are more prominent in real-world applications. For example, in `memcached`, SIDECFI achieves 96.0%, significantly better than LLVM-CFI's 91.0%. In `httpd`, SIDECFI scores 94.0%, again outperforming LLVM-CFI's 93.0%.

6.3.2. Shadow Call Stack

For Shadow Call Stack mechanisms, benchmarks such as `perlbench_s`, `omnetpp_s`, and `leela_s` were excluded from the geomean calculation due to violations caused by unhandled optimizations like tail-calls. Additionally, real-world applications `httpd`, `memcached`, and `bind` were excluded from the geomean calculation for similar LLVM-SCS violations. SIDESTACK achieves a geomean of 87.27%, while LLVM-SCS reaches 91.20%. In the remaining real-

1. Using LLVM-8, because x86-64 support has been discontinued in more recent versions.

world applications, SIDESTACK shows competitive performance, with `httpd` achieving 87.0%, closely matching LLVM-SCS's 87.0%.

6.3.3. Address Sanitizer

ASAN shows a geomean performance of 49.91%, while SIDEASAN exhibits a reduced performance at 15.11%. This reduction is primarily due to the frequent message communication with the monitor, which imposes a heavy load on debugging extensions. The high message generation rates by ASAN cause hardware backpressure, resulting in packet loss. Despite these challenges, SIDEASAN still shows its strongest performance in `memcached`, where it reaches 58.0%. For `httpd`, SIDEASAN achieves 42.0%, while ASAN performs slightly better at 51.0%.

Additionally, we evaluated `pbzip2` and `pigz` with our multithreaded SIDEASAN implementation, chosen for their high single-threaded overhead. Despite assigning threads to separate cores, the overhead remained similar to single-threaded performance. `pbzip2` showed a consistent ≈9% performance compared to the baseline, regardless of the number of worker threads. Similarly, `pigz` maintained ≈15% performance, irrespective of the worker thread count. This suggests that while the message load per thread decreases, the cumulative increase in message frequency with more threads negates this advantage, maintaining high overhead relative to the baseline without security measures.

6.3.4. Comparison with GRIFFIN

The SIDEGUARD approach integrates both forward and backward-edge protections by concurrently employing SIDECFI and SIDESTACK. Figure 11 compares the performance of SIDEGUARD with GRIFFIN [37], another comprehensive CFI solution that protects both edges. Specifically, we compare against GRIFFIN's *combination* policy that enforces a fine-grained policy on forward edges and a shadow stack on backward edges, using SPEC CPU2006 Integer with the "train" workload as they did. While SIDEGUARD maintains a relative performance of 73%, it remains competitive with GRIFFIN's 86% (overhead figures from their paper were converted for comparison). GRIFFIN employs Intel PT and worker threads for control-flow reconstruction, resulting in high resource use in trace processing. In contrast, SIDEGUARD's trace only contains messages injected via instrumentation, eliminating the need for complex decoding of a large trace by worker threads.

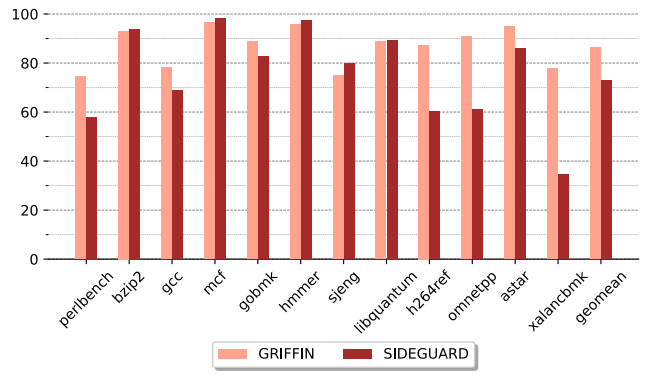

Figure 11: Relative (%) performance comparison under GRIFFIN and SIDEGUARD.

Additionally, we compare monitor resource consumption of SIDEGUARD and GRIFFIN by looking at the average CPU usage for our SIDECFI and SIDESTACK monitors running in parallel to the instrumented SPEC CPU2017 benchmarks. We utilized `getrusage` to obtain the user and system CPU time consumed by the monitor process and calculated utilization by dividing by the actual (wall) time of the monitor, obtained with `gettimeofday`. The results, as shown in Table 2, highlight the efficiency of SIDECFI and a reasonable overhead on SIDESTACK.

TABLE 2: CPU Usage for SIDECFI and SIDESTACK monitors in SPEC2006 benchmarks.

Benchmark	CPU Usage	
	SIDECFI	SIDESTACK
`perlbench`	97.02%	52.81%
`bzip2`	0.48%	6.31%
`gcc`	16.83%	49.85%
`mcf`	0.57%	1.53%
`gobmk`	16.42%	28.15%
`hmmer`	0.48%	0.50%
`sjeng`	18.14%	12.75%
`libquantum`	0.48%	4.66%
`h264ref`	44.96%	11.17%
`omnetpp`	99.93%	46.63%
`astar`	8.61%	8.15%
`xalancbmk`	98.96%	61.96%
`geomean`	8.36%	11.54%

As one would expect, CPU consumption depends on the number of messages sent. SIDECFI consumes significantly fewer CPU cycles than GRIFFIN, as the volume of messages, compared with execution tracing, is much smaller. SIDESTACK shows a geomean of 8.36% for SIDECFI and 11.54% for SIDESTACK. Assuming these are combined together to compare against GRIFFIN, the total CPU usage does not surpass 20%. Comparing this to GRIFFIN, which utilizes 6 CPU cores to decode and process the large amount of data, we estimate that we outperform them by approximately a factor of 30. This significant reduction in CPU usage highlights the efficiency of our SIDEGUARD approach compared to the more resource-intensive method employed by GRIFFIN.

7. Related Work

7.1. Parallelizing Security Checks

Multiple prior works have adopted decoupling while attempting to speed up security policy enforcement. We classify them into two categories, which we discuss below.

Software. Such approaches [25], [26], [27], [28], [29] instrument applications using dynamic binary instrumentation (DBI) frameworks like Pin [82] or DynamoRIO [83] to inject code that will log runtime data to shared buffers that can be accessed by separate monitor threads or processes. Some works have focused on expensive dynamic analyses like dynamic taint tracking [25], [26], [27], while others have aimed to provide a more general framework for dynamic analysis [28], [29], sacrificing performance for adaptability. Despite offering acceleration to applications, at least compared to inlined approaches [84], their overheads remain high for regular use in production *(>x2)*, which is due to the use of DBI and the volumes of data that need to be transferred to monitors. More importantly, the use of shared buffers to transfer data to the monitor poses the risk of tampering by an adversary before the checks are performed.
Hardware. These approaches [34], [32], [35], [85], [30], [31], [48], [86], [33] instead propose novel processor designs that automatically log data, while executing a process, to allow for security checks to be applied in parallel by specialized processing cores [34] or a monitor processes running in general-purpose cores [35], [30], [31], [48], [33]. Some of these approaches [34], [32], [30], [31], [33] are even configurable and can support different dynamic analyses. Support for multithreaded applications has also been explored in the past [87], [88], [89], with these works facing a number of challenges including inter-thread data dependencies, unmonitored operating system activity, synchronization overheads and effectively sharing communication channels between multiple threads. In general, hardware-based methods typically incur lower overheads without the risk of tampering, as data are logged into memory that is isolated from the application. However, their integration into high-performance processors is challenging and we have not witnessed their adoption by manufacturers yet.

7.2. Leveraging Debugging Extensions.

Execution Tracing. Prior research has harnessed execution tracing capabilities for offloading security tasks like control-flow integrity and attestation [36], [37], [38], [39], [40], [41], [42]. However, even though the processor can log these traces with minimal overhead *(<4%)*, the data produced are voluminous and require significant processing resources to analyze. As such, additional overhead is incurred because the core's execution needs to be stalled to allow for the trace to be processed, or multiple cores need to be reserved for protecting a single application thread. Hence, using execution tracing for security checks *is not practical yet*, unless dedicated hardware is introduced that can efficiently process the trace data. Moreover, *it can only support policies related to control flow.*
Software-Driven Logging. The use of the `PTWRITE` instruction has been underexplored, with only a few works utilizing it so far. DTrace [90] employs an emulated version of `PTWRITE` for fine-grained data integrity enforcement. Another notable use is found in the work by Lu and Hu [91], which employs `PTWRITE` to log function pointers for refining indirect-call targets through Multi-Layer Type Analysis (MLTA). Additionally, Execution Reconstruction (ER) [92] uses `PTWRITE` to record data values, identified by key data value selection, enabling the accurate reproduction of production failures. Despite these uses, the full capabilities and limits of this extension have not been thoroughly explored and challenged in existing research.

8. Conclusion

In this work, we introduce SIDECAR, leveraging and repurposing processor debugging extensions to create secure, append-only channels for offloading security policies in processors. SIDECAR supports a diverse array of policies, demonstrated through the development of forward-edge and backward-edge CFI implementations and an address sanitizer. We rigorously validated these implementations using the RIPE suite and LLVM tests, ensuring their robustness and effectiveness. Performance on SPEC CPU benchmarks indicates average overheads of 3-20% for our CFI policies, with SIDECAR-based CFI+SCS being 30 times more efficient than prior work.

Notably, in DSO-heavy applications like Apache Httpd, Bind, and Chromium SIDECAR's forward-edge CFI outperforms LLVM's implementation due to offloading the expensive slow path checks. This highlights the potential of utilizing existing debugging hardware for advanced security solutions and underscores the need for future enhancements in such extensions for more sophisticated security mechanisms. Overall, SIDECAR's innovative approach offers a compelling path forward for enhancing security with minimal performance overhead.

Availability

The prototype implementation of SIDECAR is available at: https://github.com/stevens-s3lab/sidecar

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was supported by DARPA, NSF, and CISCO under awards D21AP10116-00, CNS-2213727, and 71858473. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, DARPA, NSF, or CISCO.

References

[1] MITRE, "CWE top 25 most dangerous software weaknesses," https://cwe.mitre.org/top25/, 2023.

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *IEEE Symposium on Security and Privacy (S&P)*, 2013, pp. 48–62.

[3] V. Van der Veen, L. Cavallaro, and H. Bos, "Memory errors: The past, the present, and the future," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2012, pp. 86–106.

[4] N. Carlini and D. Wagner, "{ROP} is still dangerous: Breaking modern defenses," in *Proceedings of the USENIX Security Symposium*, 2014, pp. 385–399.

[5] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 552–561.

[6] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "{Control-Flow} bending: On the effectiveness of {Control-Flow} integrity," in *Proceedings of the USENIX Security Symposium*, 2015, pp. 161–176.

[7] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-Oriented Programming: A new class of code-reuse attack," in *Proceedings of the ACM Asia Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011, pp. 30–40.

[8] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *ACM Conference on Computer and Communications Security (CCS)*, 2010, pp. 559–572.

[9] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 575–589.

[10] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit Object-Oriented Programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2015, pp. 745–762.

[11] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proc. of the European Workshop on System Security (EUROSEC)*, 2009, p. 1–8.

[12] Y. Li, B. Dolan-Gavitt, S. Weber, and J. Cappos, "Lock-in-Pop: Securing privileged operating system kernels by keeping on the beaten path," in *USENIX Annual Technical Conference (ATC)*, 2017, pp. 1–13.

[13] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "TaintEraser: Protecting sensitive data leaks using application-level taint tracking," in *SIGOPS Oper. Syst. Rev.*, 2011.

[14] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *USENIX Annual Technical Conference*, Jun. 2012, pp. 309–318.

[15] I. Haller, J. Yuseok, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "TypeSan: Practical Type Confusion Detection," in *CCS*, Oct. 2016.

[16] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proc. of the USENIX Security Symposium*, 2014, pp. 941–955.

[17] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[18] N. Nethercote, "Dynamic binary analysis and instrumentation." [Online]. Available: http://valgrind.org

[19] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps, "ConSeq: Detecting concurrency bugs through sequential errors," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, p. 251–264.

[20] T. H. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proc. of the 10th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2015, p. 555–566.

[21] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 75–88.

[22] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis, "xMP: Selective memory protection for kernel and user space," in *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 563–577.

[23] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*, 1993, p. 203–216.

[24] Intel, "A technical look at Intel's control-flow enforcement technology," https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html, 2020.

[25] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis, "ShadowReplica: Efficient parallelization of dynamic data flow tracking," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2013, pp. 235–246.

[26] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "TaintPipe: Pipelined symbolic taint analysis," in *USENIX Security Symposium*. USENIX Association, Aug. 2015, pp. 65–80. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ming

[27] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu, "StraightTaint: Decoupled offline symbolic taint analysis," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 308–319.

[28] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley, "A concurrent dynamic analysis framework for multi-core hardware," in *Proc. of OOPSLA*, 2009.

[29] Q. Zhao, I. Cutcutache, and W. Wong, "PiPA: pipelined profiling and analysis on multi-core systems," in *Proc. of CGO*, 2008.

[30] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos, "Flexible hardware acceleration for instruction-grain program monitoring," in *2008 International Symposium on Computer Architecture*, 2008, pp. 377–388.

[31] O. Ruwase, S. Chen, P. B. Gibbons, and T. C. Mowry, "Decoupled lifeguards: Enabling path optimizations for dynamic correctness checking tools," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2010, pp. 25–35.

[32] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh, "Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric," in *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010, pp. 137–148.

[33] D. Lo, T. Chen, M. Ismail, and G. E. Suh, "Run-time monitoring with adjustable overhead using dataflow-guided filtering," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 662–674.

[34] S. Ainsworth and T. M. Jones, "The guardian council: Parallel programmable hardware security," in *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1277–1293.

[35] D. D. Chen, W. S. Lim, M. Bakhshalipour, P. B. Gibbons, J. C. Hoe, and B. Parno, "HerQules: Securing programs via hardware-enforced message queues," in *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021, p. 773–788.

[36] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "PT-CFI: Transparent backward-edge control flow violation detection using Intel Processor Trace," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 173–184.

[37] X. Ge, W. Cui, and T. Jaeger, "GRIFFIN: Guarding control flows using intel processor trace," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 585–598, 2017.

[38] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, "Transparent and efficient CFI enforcement with intel processor trace," in *2017 IEEE International Symposium on High performance computer architecture (HPCA)*. IEEE, 2017, pp. 529–540.

[39] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of Path-Sensitive control security," in *Proceedings of the USENIX Security Symposium*, 2017, pp. 131–148.

[40] O. Arias, D. Sullivan, H. Shan, and Y. Jin, "Lahel: Lightweight attestation hardening embedded devices using macrocells," in *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2020, pp. 305–315.

[41] M. A. Wahab, P. Cotret, M. N. Allah, G. Hiet, V. Lapotre, and G. Gogniat, "Armhex: A hardware extension for dift on arm-based socs," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–7.

[42] L. Feng, J. Huang, J. Hu, and A. Reddy, "Fastcfi: Real-time control-flow integrity using fpga without code instrumentation," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 5, pp. 1–39, 2021.

[43] "lit - LLVM Integrated Tester." https://llvm.org/docs/CommandGuide/lit.html.

[44] Hubert Rosier, "RIPE64. National University of Singapore." https://github.com/hrosier/ripe64, 2019.

[45] The Clang Team, "Control flow integrity," Clang 12 documentation – https://releases.llvm.org/12.0.0/tools/clang/docs/ControlFlowIntegrity.html.

[46] "Shadow Call Stack," https://clang.llvm.org/docs/ShadowCallStack.html.

[47] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis, "FineIBT: Fine-grain control-flow enforcement with indirect branch tracking," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Oct. 2023.

[48] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic, "Heapmon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection," *IBM Journal of Research and Development*, vol. 50, no. 2.3, pp. 261–275, 2006.

[49] James Reinders (Intel), "Processor tracing," Sep 2013, https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[50] ARM, "Better trace for better software," https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Better_Trace_for_Better_Software_-_CoreSight_STM_with_LTTng_-_19th_October_2010.pdf, 2010.

[51] B. C. Strong, J. W. Brandt, P. Lachner, A. Kleen, J. B. Crossland, and T. Opferman, "Software-initiated trace integrated with hardware trace," Dec. 29 2016, uS Patent App. 14/751,759.

[52] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005, p. 340–353.

[53] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the USENIX Security Symposium*, Aug. 2013, pp. 337–352.

[54] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, p. 577–587.

[55] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[56] P. Zieris and J. Horsch, "A leak-resilient dual stack scheme for backward-edge control-flow integrity," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 369–380.

[57] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining entropy-based information hiding (and what to do about it)," in *Proceedings of the USENIX Security Symposium*, August 2016, pp. 105–119.

[58] A. Oikonomopoulos, B. Kollenda, C. Giuffrida, E. Athanasopoulos, E. Göktaş, G. Portokalidis, H. Bos, and R. Gawlik, "Bypassing clang's safestack for fun and profit," in *Black Hat Europe*, November 2016. [Online]. Available: https://www.blackhat.com/eu-16/briefings/schedule/index.html#bypassing-clangs-safestack-for-fun-and-profit-4965

[59] I. Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2023.

[60] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *USENIX Workshop on Offensive Technologies (WOOT)*, Aug. 2020.

[61] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse," in *European Conference on Computer Systems (EuroSys)*, 2017, pp. 420–436.

[62] LWN.net, "x86 NX support," Jun. 2004. [Online]. Available: https://lwn.net/Articles/87814/

[63] OpenBSD, "i386 W^X," Apr. 2003. [Online]. Available: https://marc.info/?l=openbsd-misc&m=105056000801065

[64] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 1997, pp. 67–72.

[65] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of Buffer-Overflow attacks," in *Proceedings of the USENIX Security Symposium*, Jan. 1998.

[66] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted code randomization," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2018, pp. 461–477.

[67] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2014, pp. 276–291.

[68] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code Re-Randomization," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016, pp. 367–382.

[69] The kernel development community, "Perf ring buffer," https://docs.kernel.org/userspace-api/perf_ring_buffer.html.

[70] Linaro, "OpenCSD: An open source CoreSight trace decode library," https://github.com/Linaro/OpenCSD, 2023.

[71] "Intel Processor Trace Decoder Library," https://github.com/intel/libipt.

[72] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "Ripe: Runtime intrusion prevention evaluator," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011, pp. 41–50.

[73] "Leak Sanitizer," https://github.com/google/sanitizers/wiki/AddressSanitizerLeakSanitizer.

[74] "FileCheck - Flexible pattern matching file verifier." https://llvm.org/docs/CommandGuide/FileCheck.html.

[75] J. Salwan, "ROPgadget," https://github.com/JonathanSalwan/ROPgadget.

[76] "wrk - a HTTP benchmarking tool." https://github.com/wg/wrk, 2021.

[77] "Memtier Benchmark," https://github.com/RedisLabs/memtier_benchmark, 2020.

[78] "DNSPerf - DNS Performance Analytics and Comparison." https://www.dnsperf.com/.

[79] "GN - a meta-build system that generates build files for ninja." https://gn.googlesource.com/gn.

[80] "Ninja: a small build system with a focus on speed." https://github.com/ninja-build/ninja.

[81] "Telemetry: Chrome's performance testing framework." https://chromium.googlesource.com/catapult/+/HEAD/telemetry/README.md.

[82] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[83] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proceedings of the ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE)*, 2012, p. 133–144.

[84] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, March 2012, pp. 121–132.

[85] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin *et al.*, "Log-based architectures for general-purpose monitoring of deployed code," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, 2006, pp. 63–65.

[86] Q. Zeng, D. Wu, and P. Liu, "Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 367–377, 2011.

[87] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry, "Paralog: Enabling and accelerating online parallel monitoring of multithreaded applications," in *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*, 2010, pp. 271–284.

[88] M. L. Goodstein, E. Vlachos, S. Chen, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Butterfly analysis: Adapting dataflow analysis to dynamic parallel monitoring," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 257–270, 2010.

[89] M. L. Goodstein, S. Chen, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Chrysalis analysis: Incorporating synchronization arcs in dataflow-analysis-based parallel monitoring," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, 2012, pp. 201–212.

[90] X. Wang, F. Huang, and H. Chen, "Dtrace: Fine-grained and efficient data integrity checking with hardware instruction tracing," *Cybersecurity*, vol. 2, no. 1, pp. 1–15, 2019.

[91] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.

[92] G. Zuo, J. Ma, A. Quinn, P. Bhatotia, P. Fonseca, and B. Kasikci, "Execution reconstruction: Harnessing failure reoccurrences for failure reproduction," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 1155–1170.