BROL: Cache-Only Execution for Software Protection

Ruben Mechelinck
DistriNet, KU Leuven, Belgium
ruben.mechelinck@kuleuven.be

Stijn Volckaert
DistriNet, KU Leuven, Belgium
stijn.volckaert@kuleuven.be

Abstract

Industrial-scale reverse engineering is a growing problem for manufacturers of specialized equipment and machines. Both software and hardware intellectual property form the foundation of these manufacturer's competitive advantage and revenue, making them attractive targets for malicious competitors. The produced systems typically have limited resources and lack built-in protection mechanisms for the software it runs, leaving the software vulnerable to unauthorized duplication and reverse engineering. We present BROL, a technique that protects software against reverse engineering and piracy by binding it to a specific machine and hiding its code in the CPU's instruction cache. BROL loads the protected code from disk, decrypts it with a machine-specific key, and uses physical memory aliasing and targeted cache eviction to make the code unavailable in any level of the memory hierarchy except for the instruction cache. We implemented BROL for x86 and ARMv7 platforms and show that it reliably protects code without relying on dedicated security hardware to achieve maximum security. However, our evaluation also shows that BROL has non-trivial constraints that limit its applicability.

1 Introduction

In today's industry, many companies develop integrated hardware-software systems to deliver specialized functionality, such as in the mechanical and plant engineering sectors. Counterfeiters increasingly target these systems by replicating the hardware and copying the software to produce unauthorized clones. These activities cause significant economic harm by diverting revenue, violating intellectual property rights, and undermining the competitive advantage gained through R&D investments [51, 66]. While preventing the creation of hardware with equivalent specifications is often infeasible, especially since production is commonly outsourced, protecting the software against copying and reverse engineering remains a critical and more tractable challenge. This is particularly pressing because malicious actors can duplicate software with minimal effort, and the copy will execute identically on any cloned hardware with matching specifications. Vendors face a daunting task in defending their intellectual property, as once a genuine machine is acquired, attackers gain unrestricted access to both hardware and software components.

Existing defenses against software tampering and reverse engineering, such as obfuscation [18, 21, 49], trusted execution environments [41, 42, 63], remote attestation [42], and hardware-software binding using Physically Unclonable Functions (PUFs) [23, 24, 26, 47] or dongles [54], offer partial solutions to the piracy problem. However, these approaches often require specialized and expensive hardware, incur significant performance costs, or lack universal applicability across the diverse platforms used in modern industrial settings.

In this paper, we propose a new technique that hardens software against unauthorized deployment and reverse engineering. Our idea is to bind each software instance to its underlying hardware by encrypting the binary image using PUF-based encryption. At load time, we leverage physical memory aliasing to intentionally break cache coherency and erase the decrypted software from RAM while leaving it in the cache. This makes reverse engineering and code extraction extremely difficult, even without dedicated security components. We achieve the strongest protection guarantees by additionally creating a discrepancy between the L1 cache and other levels of the cache hierarchy. This way, the software resides exclusively in the CPU's L1 instruction cache at run time.

We implemented this idea in a proof-of-concept implementation called BROL (Basically Runs On L1) and evaluated it on two representative systems. We thoroughly evaluated the real-world practicality and constraints of the technique and discuss various design options and trade-offs in this paper. We conclude that our technique, if fully implemented, is a promising, inexpensive, and high-performance method that developers could use to defend against reverse engineering and piracy. However, it has non-trivial constraints that they should be aware of. Our current implementation provides full protection on systems with one cache level against powerful person-at-the-end attackers who have full system access. We additionally discuss an alternative real-world threat model where reduced protection is sufficient for systems with multiple cache levels.

2 Background

In this section, we describe the necessary background used further in this paper.

2.1 CPU Caches

Most CPUs employ high-speed caches of limited size within the CPU die to offset the high access latency to main memory. The caches store recently accessed data or prefetch data that might be used in the near future. Many architectures have multiple cache levels. Modern x86 CPUs, for example, have three cache levels [4, 9]. The lowest cache level, i.e., L1, is located closest to the CPU core and has the lowest access latency. Due to physical constraints, they are the smallest of the three levels. The higher levels gradually increase in size and access latency. On all x86 and most general-purpose ARM CPUs, the L1 cache is private to a single core and implements a Harvard architecture, meaning that there is a division between the instruction cache, i.e., L1i, and the data cache, i.e., L1d [4, 8, 9]. Other levels of the cache might be shared between multiple cores and do not separate instructions from data. We therefore refer to these caches as unified caches. In a setup like this, the CPU's instruction fetch unit is connected only to the L1i cache. This cache can therefore only serve instruction fetches and not data fetches.

ARM and x86 caches are organized in cache lines and operate in a set-associative manner [4, 8, 9]. Each memory location maps to a set of multiple cache lines. Each such line is called a cache way, and the associativity of the cache denotes the number of lines within the set. When the CPU issues a request for a memory location, it selects the cache set using a bit slice in the requested memory address. On Virtually Indexed Physically Tagged (VIPT) caches, it uses the requested virtual address, while it uses the requested physical address on Physically Indexed Physically Tagged (PIPT) caches. The cache replacement policy selects the specific way within the set. This policy determines which older lines are evicted to make way for the new data. Most vendors do not disclose the details of their replacement policies. However, past research has shown that at least some microarchitectures use adaptive replacement techniques that dynamically switch policies based on the access pattern [68].

Cache Inclusion. CPUs with multiple cache levels implement a cache inclusion policy. This policy determines whether a higher cache level holds copies of all lines in its lower levels. The L3 cache on older Intel Core CPUs, for example, is inclusive of L2 and L1, which means L3 contains copies of all data cached in L2 and L1 [9]. We verified, using `cpuid` on Intel's 2nd to 8th generation Core CPUs, that L3 is inclusive of its lower cache levels and L2 is non-inclusive of L1 [9]. Intel currently uses a non-inclusive cache across its Xeon series and its modern Core series starting with the twelfth generation, and its mobile eleventh-generation CPUs [13, 14, 70].

Cache Coherence. Due to the shared nature of data between cores and caches, the CPU runs a cache coherence protocol to track modifications to serve each memory request with the most recent data. The most widely used cache-coherence protocol, MESI, assigns one of the following states to each cache line: Modified (M), Exclusive (E), Shared (S), or Invalid (I). AMD and many ARM CPUs use a derivative called MOESI with an extra Owned (O) state and Intel uses MESIF with an extra Forward (F) state [3, 4, 8, 73]. This protocol allows cache-to-cache transfers when one core requests data that is already cached in another core. This reduces the number of requests forwarded to main memory.

On most architectures, however, the L1i cache is not part of the coherence protocol and the snoop control unit does not maintain instruction coherence between cores [3, 5, 8, 9, 10, 11]. It does not implement a modified state and requires special considerations for software containing self-modifying code. On x86, the CPU guarantees instruction coherence at the architectural level, not by using the cache coherence protocol, but by invalidating the cache lines of the modified code from all cache levels [4, 9]. The only difference on ARM is that self-modifying code needs to flush those cache lines itself [5, 10].

Memory-Cache Consistency. The operating system has some degree of control over the CPU's cache allocation and coherence policy [4, 5, 9]. By default, user-space pages allocated by the OS use the write-back and write-allocate policies. This means that (i) loads and stores on these pages operate on the cache, bringing data into the cache if necessary, and (ii) the CPU only writes data back to memory when it evicts a cache line that is in a modified state. The OS can also mark pages as uncached. In that case, loads and stores on these pages operate directly on memory.

2.2 Physically Unclonable Functions

A Physically Unclonable Function (PUF) is a hardware primitive that returns a response to a given challenge [29, 60]. They construct these responses based on physical hardware characteristics resulting from random variations in the manufacturing process. This makes it infeasible to clone a PUF's behavior in other hardware. Many kinds of PUFs, exploiting different unclonable hardware characteristics, have been proposed in the literature. For example, SRAM PUFs use the start-up values of the memory cells as a source of unclonable randomness [27, 30, 31, 72], while DRAM PUFs use the start-up values, access-time failures, capacitor retention, or latency failures [28, 35, 36, 62]. Common use cases for PUFs include identification and authentication, cryptographic key storage, and hardware-software binding [40, 43, 53, 58, 60, 69].

2.3 Physical Memory Aliasing: BadRAM

In BadRAM, De Meulemeester et al. attack trusted execution environments, like AMD SEV-SNP and Intel SGX, using physical memory aliasing within the memory module [22]. Each memory module contains a Serial Presence Detect (SPD) chip that encodes information about the module's storage configuration, e.g., the number of DRAM rows. During boot, the BIOS uses this data to initialize the memory controller and constructs the flat physical memory space presented to the OS. The authors increment the number of row bits in the SPD chip, effectively doubling the amount of memory reported to the system. The extra ghost bit, however, is not connected to any physical address line, so any use of this bit is ignored. Therefore, any pair of physical addresses that only differ in the ghost bit aliases to the same storage location on the module. This allows an attacker to modify critical data, oblivious to the OS and the CPU. While BadRAM presents itself as an attack primitive, we use physical memory aliasing for a benign use case. Henceforth, we will call the area where the ghost bit is set the aliased memory area, and the area where the ghost bit is unset the non-aliased memory area.

3 Threat Model

The techniques we propose protect industrial assembly-line software from attackers seeking to steal Intellectual Property (IP), including its binary code and information about its run-time behavior. This software is tightly coupled with specific hardware to form a functional machine. Vendors compile and ship the software-hardware pair to clients themselves, thus enabling them to use hardware-specific protection mechanisms if necessary.

We assume the attacker's goal is to produce functional machine clones. To do so, they may reverse-engineer the software, deploy unauthorized copies on cloned hardware, analyze run-time behavior, or tamper with the software. We assume attackers have access to exact hardware clones and that the victim software is free of vulnerabilities.

The hardware consists of a general-purpose computing stack with at least a multicore CPU, main memory, and persistent storage. The CPU must use a Harvard architecture in its L1 cache with separate instruction and data caches, where the instruction cache is not part of the cache coherence protocol. The system does not contain any trusted computing components (e.g., Trusted Platform Modules (TPM) and Trusted Execution Environments (TEE)) to support software protection mechanisms. Physical memory aliasing (e.g., via BadRAM) must be feasible. We consider two adversarial scenarios where cache requirements may vary.

Scenario 1: On-site deployment Here, the victim software runs on-site at a legitimate manufacturing facility. Attackers may gain physical or remote access with full user-space privileges (including root), but they cannot reboot the machine or modify the kernel, as such events are easily detectable. The system runs a trusted Linux kernel that enforces kernel module signing through the `CONFIG_MODULE_SIG_FORCE` build parameter or the `module.sig_enforce=1` command-line parameter. Consequently, the root user cannot load arbitrary modules into the kernel. With physical access, attackers can inspect the hardware. They can also swap out hot-pluggable hardware, such as storage devices and peripherals, but cannot change the CPU or main memory, since this would require a reboot. These capabilities empower the attacker to extract the software binaries and analyze or manipulate running processes.

Scenario 2: Attacker-owned system Here, attackers acquire a legitimate hardware-software pair, gaining full system access, including to the kernel and the main memory. We do, however, assume that the CPU has been physically tamper-proofed so that it cannot be modified to allow for direct inspection of the L1 instruction cache contents. In this scenario, the attacker can monitor and manipulate kernel-level operations.

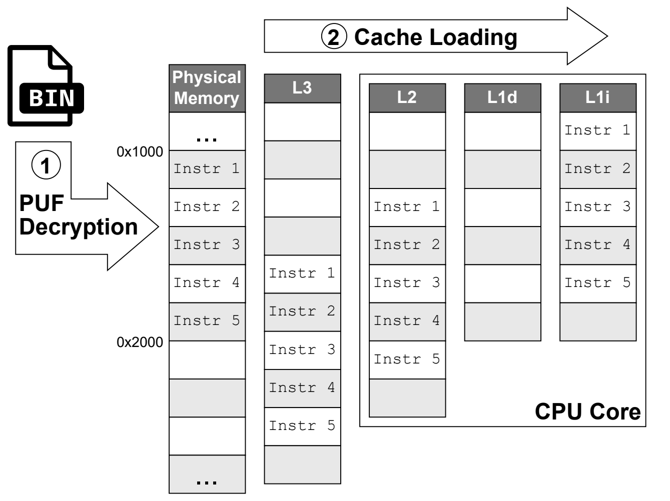
4 Design & Protection

We propose BROL, an approach that protects software against malicious reverse engineering, tampering, and unauthorized copying. BROL leverages (i) the unique properties of CPU instruction caches to store protected code safely at run time, and (ii) physical memory aliasing to invalidate code stored in memory while leaving its cached version intact.

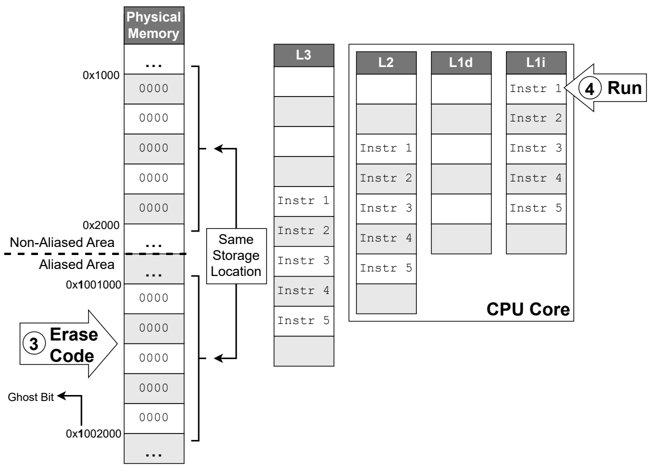
In this section, we first explain BROL's base operations, along with the protections it provides in adversarial scenario 1. Afterward, we describe an extension to BROL that additionally protects the protected software in adversarial scenario 2. Table 1 summarizes all discussed protections against attackers in both adversarial scenarios.

Attacker's Capabilities	BROL Countermeasures
Static Image: analyze, deploy	The binary is encrypted with a PUF-derived key.
Memory Image: analyze, dump, tamper	The protected code is erased from memory.
Caches: read, write-back, tamper, cache-to-cache transfer	The protected code is evicted from data and unified caches.

Table 1: Summarizing table showing all targets containing the protected code and how BROL defends them. The gray row represent attacks that are only possible in adversarial scenario 2.



(a) BROL decrypts and loads the protected code in memory ①. Afterward, it loads the code in the cache hierarchy by running it on dummy data ②.



(b) BROL erases the memory representation of the code through physical memory aliasing ③. The caches remain intact. Afterward, the protected code runs from the caches ④.

Figure 1: Overview of BROL.

4.1 Base Operations

BROL uses a preloader that sets up the environment for the protected software. This preloader operates in multiple stages, as shown in Figure 1. Here, we explain each step along with its protective purpose. We discuss measures to reduce interferences in Section 5.

In the first stage, BROL loads the protected software into memory ①. The software's binary image is preinstalled on disk in encrypted form to prevent static analysis. We use a PUF-derived encryption key to bind the shipped software image to its associated hardware securely. This prevents unauthorized deployments of copied images on cloned hardware. On the associated machine, the preloader decrypts the image and loads the plain code into memory.

The plain-memory representation of the protected code is a prime target for attackers because a copy of this decrypted representation can be deployed on cloned hardware. BROL's next step is to erase this memory representation. The preloader runs the protected code from memory on dummy data to load it into the CPU's caches ②. The code is now present in the L1 instruction cache and, depending on the cache architecture, possibly in other unified cache levels. Any later execution of the protected code will be served from these caches, rendering the memory representation redundant.

BROL then uses physical memory aliasing to erase the code in memory while keeping the cached version intact ③. By overwriting the code in memory using an aliasing physical address, both the OS and CPU are unaware that the protected code in the caches becomes stale, so they will not instigate any cache coherence operations. At this stage, only the internal CPU caches hold the valid protected code. After preloading is finished, the software can run from the caches on its real-world input data ④.

Having the correct PUF responses and having physical memory aliasing enabled are both crucial to BROL's correctness and security guarantees. On a machine with incorrect PUF responses, BROL would decrypt the software into incorrect and, likely, invalid code. With the expected PUF responses, but without aliasing, BROL would correctly decrypt the binary code but not erase its representation in memory, as explained in Section 7. To combat this threat, BROL uses a physical memory PUF (i.e., a DRAM PUF), which it queries in part through the aliased area of the physical memory space. If an attacker disables aliasing, these queries will either crash because BROL reads from invalid memory addresses, or generate incorrect responses if the attacker has replaced the aliased area with additional physical memory.

4.2 Protections: Static Image and Memory

With the design we described so far, attackers in both adversarial scenarios cannot deploy copies of the static binary image of the protected software on hardware clones due to the PUF-based encryption. The design also prevents them from performing static analysis on these images for the same reason. Tampering with the code in memory or creating a software image from it is, furthermore, not possible due to the erased code in memory.

However, attackers can try to perform dynamic analysis on the associated hardware, after BROL has loaded the code into the cache. Their options include dynamic binary analysis tools (debuggers, tracers, and processor tracers like Intel PT) and dynamic binary translation or instrumentation frameworks (e.g., Valgrind, Intel PIN) to add instrumentation code or monitor features such as execution traces, data dependencies, etc. Fortunately, all aforementioned tools and frameworks require at least a correct memory representation of the code, either as the input for analysis or, in the case of Intel PT, to map captured instruction pointers to their corresponding code [1, 9]. This intact memory representation is absent after preloading completes, rendering all the aforementioned dynamic analyses insufficient for extracting implementation details from the protected software.

Without a code image, an attacker can still use a processor tracing mechanism, such as Intel PT, to observe the length of each executed instruction and reconstruct the program's control-flow structure, including branch targets, loops, and function calls. However, in the general case, we argue that to understand the actual operations performed by a code sequence, our adversary additionally requires knowledge of the instruction semantics and the data flow through the code sequence, particularly the dependencies between operands. The attacker can obtain this information only by mapping the captured instruction pointers back to the instructions of a code image, which is absent in BROL [1, 9]. Other CPU-based tracing features, such as performance counters, provide an attacker insight into the instruction types of a code sequence (e.g., conditional or unconditional branch, arithmetic, vector, or memory instructions), along with their microarchitectural events such as mispredictions, elapsed clock cycles, pipeline clears, etc. The attacker can obtain similar information through CPU side channels based on port contention, branch prediction analysis, etc. [17, 55]. However, for the same reasons stated above, we believe this information is insufficient for an adversary to reverse engineer the protected program fully.

4.3 Protections: Caches

With these techniques neutralized, the attacker might instead focus on the protected code inside the caches. They can read the code from the cache using load instructions, tamper with it using store instructions, or write it back to main memory by evicting it while in a modified state.

Reading the protected code or tampering with it while it is loaded in the caches requires the attacker to issue memory requests that map to the cache lines containing the protected code. The kernel in adversarial scenario 1 is trusted and the protected software does not contain vulnerabilities, so the attacker cannot make these memory requests from within the protected process context. To issue them from another context on systems with VIPT cache levels, the attacker needs access to a mapping with the same virtual and physical address pair as the protected code in the protected process. On a PIPT system, they only need a mapping with the same physical address. However, since the kernel is trusted, the attacker cannot alter page tables directly, and therefore cannot map any physical page of the protected process into their own attacker process. They could, however, access the protected process' virtual address space using APIs such as ptrace. When running a trusted kernel, we can easily block ptrace attachment using `PT_DENY_ATTACH`. The aforementioned restrictions and techniques defeat an attacker in adversarial scenario 1.

In adversarial scenario 2, the attacker can compromise or replace the kernel to bypass the restrictions mentioned above. They have complete control over the protected process's page tables and address space. They can also run arbitrary code in the context of the protected process, for example, by adding new code pages and overwriting the instruction pointer. The attacker needs to mark those new code pages as uncachable to prevent the new code from evicting some of the protected code lines preloaded in the caches. This allows our attacker to perform read and write operations on physical memory assigned to the protected process, thus allowing them to access the protected process' cache. Alternatively, on a PIPT system, the attacker can also gain access to the protected process' caches by mapping the protected process's physical page frames into the virtual address space of a malicious process. On a VIPT system, they additionally have to map these page frames at the same virtual addresses as in the protected process so they can access the caches through the same virtual and physical address pair.

However, even with these additional capabilities, extracting code from the cache may be subject to non-trivial limitations. We describe the limitations of all feasible extraction mechanisms below.

Read The attacker could simply inject code into the protected process to read the protected code from the caches using standard load instructions. Since load instructions are data operations, the CPU serves them from the data or unified caches, i.e., L1d, L2, and L3. Thus, code stored in these caches can be readily extracted. If the code is only present in L1i, however, this mechanism will fail.

Cache-to-Cache Transfer Similarly, the attacker could issue the same load operations from another process running on another CPU core. The protected process' core will transfer the requested data to the attacker's core. The snoop control unit, however, only tracks data in the data and unified caches [3, 5, 8, 9, 10, 11]. Therefore, this mechanism only works if the code is present in L1d or higher-level caches. If the protected code is only present in the instruction cache, this technique fails because the attacker's core will fetch a fresh copy of the data from memory and store it in its local caches.

Write-back The CPU writes a modified cache line back to memory when it is evicted due to a collision or explicit flush operation (e.g., on x86, with `clflush` or `clwb` from user space, or `wbinvd` from kernel space). For efficiency, unmodified cache lines are simply evicted without writing them back to memory. Hence, to write back protected code, attackers must first bring the corresponding cache line into a modified state. They can easily do this without overwriting the contents of the cache line, for example by executing an `and 0xff` instruction on a single byte of code. A subsequent flush instruction of the cache line should now replace the erased code in memory with the valid protected code from cache.

Intel CPUs, however, detect these modifications to cached code and will first invalidate the corresponding lines from all caches before making the modification on a fresh data fetch from memory [9]. This effectively neutralizes the attack because BROL invalidated the in-memory version of the code in Step (3) (Section 4). ARM CPUs, on the other hand, perform the modification on code cached in the data and unified caches while keeping the unmodified code version in its instruction caches until the software explicitly flushes it [5, 10]. Therefore, this attack only applies to the code currently cached in the data and unified caches.

All mechanisms described above fail if the code we wish to protect is not present in the data or unified caches (i.e., L1d, L2, and L3). We therefore designed BROL to evict code from these caches, leaving the only functional copy in the L1i cache. On systems with multiple cache levels, this is a critical step to guarantee BROL's effectiveness in adversarial scenario 2. As illustrated in Figure 2, we use a pattern of data-read operations on junk data that is mapped onto the same cache sets as our protected code (3.5). After eviction, the CPU executes the protected code directly from the L1i cache (4). BROL's eviction mechanism works on systems in which all cache levels are non-inclusive or exclusive of L1i, as in modern Intel CPUs [70]. In Section 6, however, we show that it is not trivial to perform these targeted evictions from the unified and data caches on these CPUs. Some systems using low-end ARM Cortex A and M CPUs have only one cache level or can be configured to use only one active cache level with a separate data and instruction cache [2, 3, 6, 16]. On these systems, eviction is not necessary at all since the preloading step ((2) and (3) in Figure 1) suffices to reach the desired state where only L1i holds the protected code.

The aforementioned restrictions and techniques protect the code even in the absence of a trusted kernel. This defeats an attacker in adversarial scenario 2.

5 Implementation

We implemented two prototypes of BROL, one for x86 and one for ARMv7 [1]. For both prototypes, we implemented a user-space preloader that loads the protected code into the caches as described in Section 4, and a kernel module that grants us fine-grained control over the CPU and physical memory.

[1] BROL's source code is available at: https://github.com/COMET-DEPS/BROL-cache-only-execution

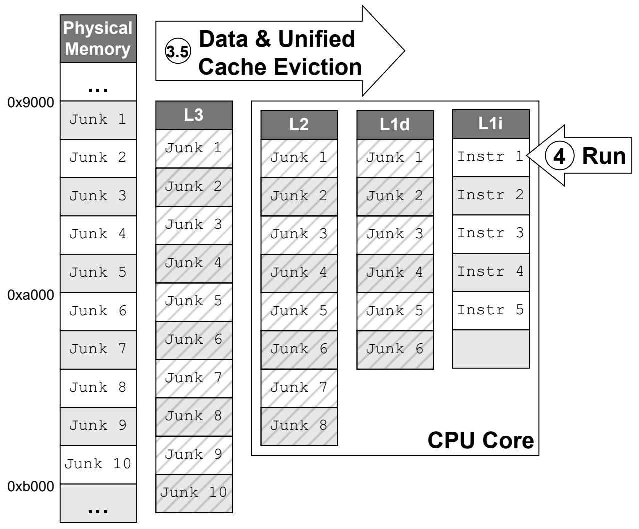


Figure 2: BROL evicts the data and unified caches by loading junk data (3.5). The protected code in the instruction cache stays intact on systems with a non-inclusive or exclusive cache hierarchy. Afterward, the protected code runs from the instruction cache alone (4).

Both implementations use physical memory aliasing to erase the protected code from RAM while leaving it in the cache. Our x86 prototype uses the BadRAM method that enables physical memory aliasing by reprogramming the SPD chips of the memory modules [22]. At boot time, the BIOS initializes the memory controller using the configuration information stored in these chips. Our ARMv7 system does not use an SPD chip to store the memory configuration, so we implemented physical memory aliasing by changing the BIOS itself to initialize the memory controller with a larger memory area. SPD-less memory is common in embedded systems.

To avoid the system from using memory in the aliasing region, we use the `memmap` kernel command-line parameter to reserve this region on x86 machines. On ARMv7, we marked the aliasing region as reserved in the flattened device tree. In both cases, the kernel will not assign memory from this region to any user-space process, nor allocate it for itself. The kernel is still aware of this region, allowing us to make on-demand allocation requests for it through our kernel module. While bringing the system into a cache-incoherent state, we mark allocations in the aliasing region as uncached, so any alteration goes directly to memory instead of to the caches first.

Right before loading the protected code into cache, BROL flushes all caches on the system. This allows us to utilize every cache way of a set because the replacement policy will always select an unused cache way, i.e., in an invalid MESI state, when bringing new data into a cache set [15].

One major practicality challenge for BROL is to keep preloaded protected code intact while it resides in the caches. If the CPU evicts any cache line holding protected code, it will refetch the erased representation from memory when that code is due, resulting in unexpected behavior. To avoid these unwanted evictions, we prevent the caches from being used by non-BROL code, e.g., from other user-space processes or kernel threads. We accomplish this by isolating one CPU core, i.e., the protected core, and only use it to run the protected code. BROL does not run any of its own code on this core, except when it preloads the protected code (2 and 4 in Figure 1). Using the appropriate kernel command-line arguments, we also isolate the core from the process scheduler, kernel threads, interrupt handlers, and RCU callbacks. This measure prevents any other user-space or kernel code to ever run on the core we assigned to the protected code.

Our prototypes fully implement the cache preloading and eviction protection described in Section 4. However, we did not implement the proposed PUF-based decryption. We believe this is a fair omission because (i) our focus in this paper is the cache-only execution mechanism, (ii) PUF-based encryption is a well-established technique that is not fundamentally difficult to implement [40, 43, 53, 58, 60, 69], (iii) its absence does not affect our evaluation or conclusions, and (iv) DRAM PUFs pair very well with our design *and* are compatible with our design goals (inexpensive and universally available). We elaborate on the potential use of a Rowhammer-based DRAM PUF in Section 7.4.

6 Evaluation

The biggest challenge to the practicality of BROL is to keep the protected code in the caches throughout the execution of the protected program. If the CPU evicts any cache line containing protected code, it re-fetches erased code from memory upon execution, leading to unexpected behavior. We evaluated the real-world feasibility of BROL by investigating the extent to which the CPU inadvertently evicts the protected code from the cache.

In our experiments, we preloaded the protected code into the caches and erased it from memory as described in Section 4. We gradually increased the complexity of the protected code to study the effect of complex control flow on cache evictions. We initially do not evict the protected code from the data and unified caches (3.5 in Figure 2).

Experimental Setup Our first test system contains an x86-based Intel Core i7-13700K CPU with 8 GiB of non-aliased physical memory and runs Fedora 41 with Linux kernel version 6.12. We implemented physical memory aliasing on the memory DIMM itself using the BadRAM method. At boot time, the system reports 16 GiB of physical memory to the OS. We used the caches of one P-core to preload the protected code into and disabled hyperthreading to prevent these caches from being shared with another logical core. Additionally, we disabled all prefetchers on this core to reduce background cache activity. This CPU has a core-private L2 (2 MiB per core) and a shared L3 (30 MiB) cache, which are both non-inclusive of the core-private L1i (32 KiB per core) and L1d (48 KiB per core) caches. This should, theoretically, allow us to evict the cache lines containing protected code from the unified L2 and L3 caches while keeping them in the L1i cache. This system uses 64-byte cache lines in all its cache levels.

Our second test system is a Banana Pi M2 Ultra. This single-board computer contains an Allwinner A40i SoC with four ARM Cortex A7 CPUs implementing the 32-bit ARMv7 architecture with support for the Large Physical Address Extension (LPAE) [3, 7]. It contains 2 GiB of non-aliased physical memory and it runs Armbian Community 25.11. The CPU implements a shared L2 (512 KiB) cache and a core-private L1i (32 KiB per core) and L1d (32 KiB per core) cache. This system uses 64-byte cache lines in all its cache levels, except for L1i, where it uses 32-byte cache lines. The inclusion policy of these caches is undocumented. The CPU allows the kernel to disable its data and unified caches, i.e., L1d and L2, while keeping L1i up. The system uses a u-boot bootloader, which also fulfills the role of BIOS. We implemented physical memory aliasing by modifying u-boot to initialize the memory controller with an extra row bit. The booted system now reports 4 GiB of physical memory. Because, in addition to physical memory, memory-mapped I/O also requires physical addresses, the physical address range cannot be represented with 32-bit pointers anymore. We therefore use a plain Debian Linux kernel version 6.12 with LPAE support.

Experiment 1 The goal of our first experiment on x86 was to find out how much code we could run from the caches without triggering unwanted evictions. We preloaded a region containing straight-line code into the caches and, subsequently, overwrote it in memory with different code. As shown in Listing 1, each cache line starts with one 3-byte `INC RAX` instruction, followed by `NOP` instructions to fill the line. Henceforth, we reference a whole cache line by the single instruction at the start of the line. The code region was allocated at a cache-line boundary. The data dependencies between the instructions prevent out-of-order execution and the lack of control flow changes eliminates the need for speculation. We ran the code once to preload it in the cache and recorded the result in `RAX` for later comparison. Afterward, we overwrote every `INC` instruction in the memory version with a 3-byte `DEC RAX` instruction through our BadRAM-enabled physical memory aliases. We ran the code again to check if the result in `RAX` matched the recorded result of the preloading round. If the resulting value was smaller, the difference indicates how many cache lines were evicted from the cache and replaced by refetches of the overwritten code in memory. Note that this number does not show evictions of cache lines containing code that had already been executed.

We increased the region size in steps of 256 cache lines and ran six iterations for each size. Figure 3 shows the average, minimum, and maximum number of evicted cache lines for a variable code region size. We did not encounter any unwanted evictions for regions smaller than 10,000 cache lines. The figure shows that the code runs reliably from the caches until the size reaches 10,000 cache lines (625 KiB). Within 10,000 cache lines (625 KiB) and 32,000 cache lines (2000 KiB), we encountered unwanted evictions while at least one run finished without evictions. Starting at 32,000 cache lines (2000 KiB), we found program sizes that failed in every iteration.

These unwanted evictions may arise from a variety of sources. Branch mispredictions, for example, might trigger speculative instruction fetches of uncached code, which in turn might evict some of the preloaded code from the caches. External processes could also create interferences but we try to reduce these to a minimum as discussed in Section 5.

We conclude that, on this particular microarchitecture, BROL reliably runs a straight-line code region below 10,000 cache lines (625 KiB) without any unwanted evictions, and until 32,000 cache lines (2000 KiB) with at least one successful run.

```
# cached version             # memory version
0x1200: inc rax; nops        0x1200: dec rax; nops
0x1240: inc rax; nops        0x1240: dec rax; nops
0x1280: inc rax; nops        0x1280: dec rax; nops
0x12c0: inc rax; nops        0x12c0: dec rax; nops
```

Listing 1: The straight-line code used in experiment 1 to preload the caches (left) and to overwrite memory with (right). Highlighted lines indicate a difference.

Experiment 2 In the next experiment, we added control-flow changes to the code setup from experiment 1 by inserting loops with conditional backward jumps. Listing 2 and Listing 3 show our test code using direct and indirect backward jumps, respectively. Each loop runs five iterations containing five `INC EDX` instructions. We varied the size of the code region by concatenating more loops, each separated by nine `INC EAX` instructions. We used a different register to count the evicted cache lines inside and outside loops because there might be a difference in the eviction pattern for that executed code more frequently. To erase the code in memory, we changed all `INC` instructions, both inside and outside the loops, to `DEC` instructions.

This pattern defines two data-dependence regions: one comprising all instructions inside the loops and the other comprising all instructions outside the loops which, furthermore, form a straight-line code sequence. Instructions within each region are data-dependent on one another, but not on instructions from the other region. This allows the Out-of-Order (OoO) CPU backend to interleave the code from these regions at run time.

Figure 4 shows the results for eleven iterations of this code pattern, both for direct and indirect branches, and for a varying number of concatenated loops. We found that this pattern runs reliably until the size reaches 32,002 cache lines (1.95 MiB), and until 48,002 lines (2.93 MiB) with a least one successful run. These limits are higher compared to the straight-line code in experiment 1, likely because of this pattern's OoO capabilities. If the CPU frontend mispredicts a branch target, the OoO backend can always continue on the instructions outside the loops while the mispredicted instructions are cleared. The replacement policy based on LRU will successively mark instructions outside the loops as the next eviction candidate for a set, because these are least-recently used. The associativity of the caches on our test CPU lies between 8 and 16. Due to OoO execution, it is likely that the next 8 to 16 instructions, i.e., cache lines, outside the loops have already finished executing.

There is no notable difference when using direct or indirect jumps, except for the large difference at 40,002 cache lines which we assume is a transitional phenomenon.

We conclude that this control flow pattern allows BROL to run larger code regions from cache, likely due to limited OoO execution between the two code regions.

```
# cached version             # memory version
0x1180: inc eax; nops        0x1180: dec eax; nops
0x11c0: mov rcx, 5; nops     0x11c0: mov rcx, 5; nops
L1:                          L1:
0x1200: inc edx; nops        0x1200: dec edx; nops
0x1240: inc edx; nops        0x1240: dec edx; nops
0x1280: inc edx; nops        0x1280: dec edx; nops
0x12c0: inc edx; nops        0x12c0: dec edx; nops
0x1300: inc edx; nops        0x1300: dec edx; nops
0x1340: dec rcx;             0x1340: dec rcx;
        jnz L1; nops                 jnz L1; nops
0x1380: inc eax; nops        0x1380: dec eax; nops
```

Listing 2: The code with direct conditional backward jumps used in experiment 2 to preload the caches (left) and to overwrite memory with (right). Highlighted lines indicate a difference.

Experiment 3 Next, we ran a collection of small real-world applications with BROL. We chose the Suckless sbase collection of light-weight Unix tools[2]. These programs are written in C and focus on implementation simplicity. They are, furthermore, small in size, making them a good starting point for validating BROL.

Real applications typically use code from external libraries. Because caches are small, we only preloaded the internal code, i.e., the code from the sbase source. However, when the

[2]https://core.suckless.org/sbase/

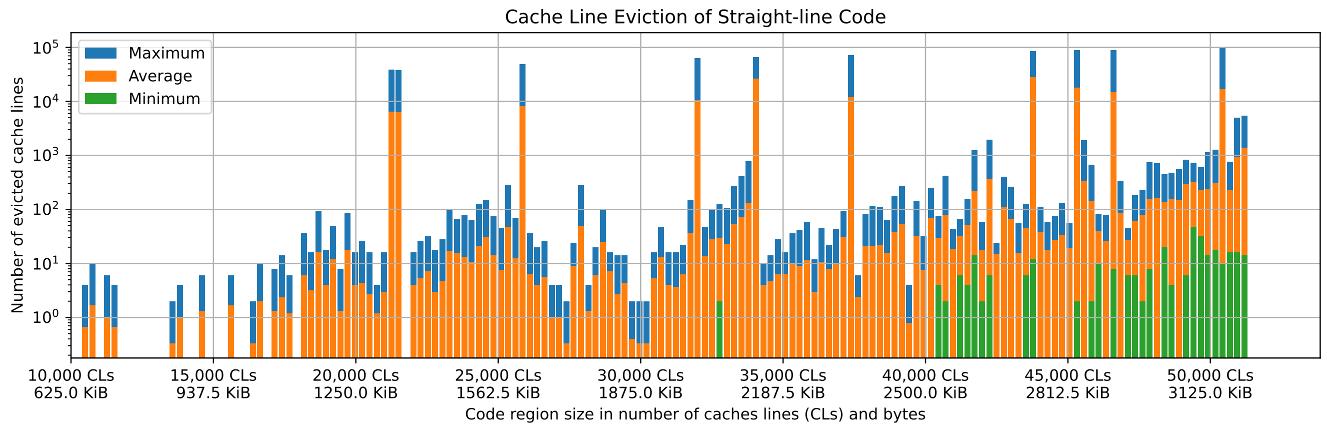


Figure 3: Number of evicted cache lines for a variable code region size containing straight-line code.

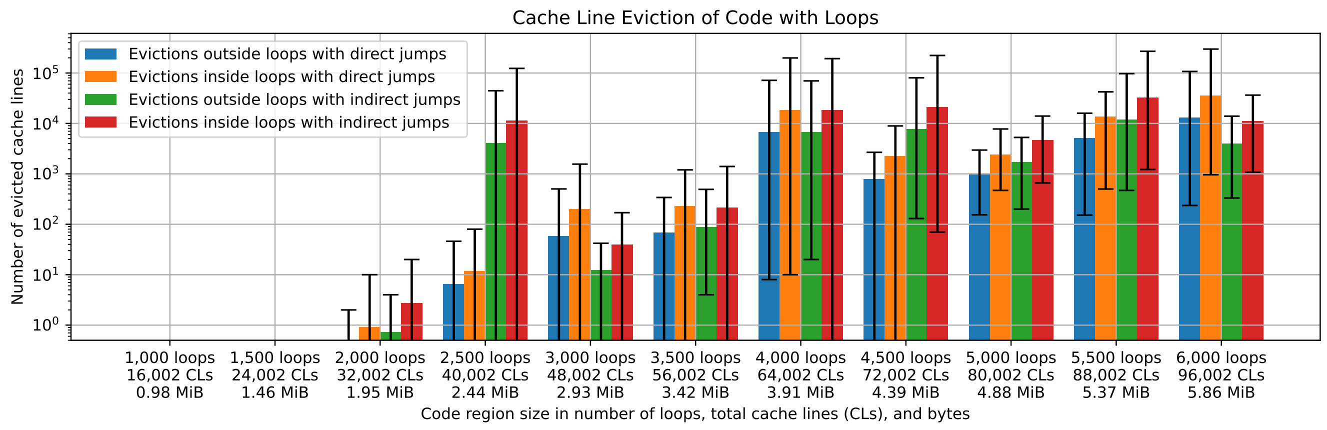


Figure 4: Number of evicted cache lines for a code region with a variable number of loops. The error bars represent the minimum and maximum encountered evictions.

program runs, the external code might interfere and cause unwanted evictions of the preloaded code. Therefore, we wrote an LLVM compiler pass that instruments the code to switch CPU affinity to another core when calling an external function. After the function returns, we switch back to the core with the preloaded code in its caches.

We used Intel PT on our instrumented binaries to find all internal code locations covered by the program for a given set of arguments. BROL preloads the internal code in cache by running it once with the same arguments. Afterward, we erased the memory version by overwriting the entire code region with breakpoint interrupt instructions, i.e., `INT3`. Then we reran the code with the same arguments while recording whether the program encounters an interrupt instruction.

We ran 50 iterations of each selected program and report their success rate in Table 2, along with the size of the code region covered. We only count a run as successful if it finishes with a zero exit value, without hitting an `INT3` instruction. The results show that all tested programs have at least one successful run from the cached code. For larger programs, only a portion of the executions were successful because the CPU evicted at least one cache line containing code that was still required. Programs with a covered code size below 1 KiB finished successfully in all iterations. This result is multiple orders of magnitude smaller compared to the results in experiment 1 and 2. We explain this using the added complexity of real-world programs. Various microarchitectural mechanisms, such as out-of-order execution with complex data dependencies, branch prediction, and speculative execution, increase cache traffic which interferes with BROL's preloaded code.

To get more insight into the failed runs, we ran the program while monitoring performance counters. We cannot use any conventional program analysis tool as BROL renders them ineffective. Using the `INST_RETIRED.ANY` counter, we can estimate at which point the program crashed. We found that in nearly every case when a program crashed, it did so while

```
# cached version              # memory version
0x1180: inc eax; nops         0x1180: dec eax; nops
0x11c0: mov rcx, 5;           0x11c0: mov rcx, 5;
        lea r11, [rip]; nops          lea r11, [rip]; nops
0x1200: inc edx; nops         0x1200: dec edx; nops
0x1240: inc edx; nops         0x1240: dec edx; nops
0x1280: inc edx; nops         0x1280: dec edx; nops
0x12c0: inc edx; nops         0x12c0: dec edx; nops
0x1300: inc edx; nops         0x1300: dec edx; nops
0x1340: dec rcx;              0x1340: dec rcx;
        jz L2; jmp r11; nops          jz L2; jmp r11; nops
L2:                           L2:
0x1380: inc eax; nops         0x1380: dec eax; nops
```

Listing 3: The code with indirect conditional backward jumps used in experiment 2 to preload the caches (left) and to overwrite memory with (right). Highlighted lines indicate a difference.

Program Invocation	Covered Code Size	Success Rate
`hostname`	0.32 KiB	100%
`dirname ./dirname.c`	0.32 KiB	100%
`echo 50377189`	0.42 KiB	100%
`pwd`	0.53 KiB	100%
`head ./sbase/head.c`	0.82 KiB	100%
`seq 0 10`	1.13 KiB	86%
`uuencode ./wc.c base64`	1.19 KiB	48%
`sha256sum ./sha256sum.c`	1.48 KiB	82%
`wc ./wc.c`	1.67 KiB	50%
`strings ./strings.c`	1.91 KiB	58%
`ls -lah`	3.44 KiB	48%
`expand ./expand.c`	1.66 KiB	24%
`sort ./sort.c`	2.12 KiB	34%

Table 2: The success rate when running a set of sbase programs with BROL.

executing the first few assembly instructions, suggesting that the unwanted evictions occur during the preloading run of the code ((2) in Figure 1). We recorded other performance counters during the preloading run and watched for sporadic events that might explain occasional unwanted evictions. Concretely, we monitored the following performance events across 50 runs while taking counter multiplexing into account: `BR_INST_RETIRED.{ALL_BRANCHES,COND}`, `BR_MISP_RETIRED.{ALL_BRANCHES,COND,INDIRECT,RET}`, `BACLEARS.ANY`, `OFFCORE_REQUESTS.DEMAND_CODE_RD`, `MACHINE_CLEARS.COUNT`, `MACHINE_CLEARS.SMC`, `FRONTEND_RETIRED.ITLB_MISS`, `L2_LINES_OUT.SILENT`, `L2_LINES_OUT.USELESS_HWPF`, `L2_RQSTS.ALL_CODE_RD`, `L2_RQSTS.CODE_RD_HIT`, `L2_RQSTS.CODE_RD_MISS`. We found that none of these counters exhibited irregularities that consistently led to failure when the binary was subsequently run from the cache ((4) in Figure 1). However, we did notice that the `L2_LINES_OUT.SILENT`, `L2_RQSTS.ALL_CODE_RD`, `L2_RQSTS.CODE_RD_HIT`, `BR_MISP_RETIRED.{ALL_BRANCHES,COND}`, and `BACLEARS.ANY` counters showed large fluctuations across all preloading runs, suggesting that the internal CPU state varies substantially across runs. This might explain the non-deterministic nature of the larger programs under BROL.

Adversarial Scenario 2 To test BROL's reliability in adversarial scenario 2, we need to evict the protected code from the data and unified caches, i.e., L1d, L2, and L3 on x86. The protected code is never present in L1d, leaving only L2 and L3. To guarantee protection with BROL, we need to evict every protected code line with certainty from the unified caches, which is challenging due to the following reasons.

- Earlier work on microarchitectural attacks constructs eviction sets consisting of N congruent addresses, i.e., addresses with the same set-selection bits [45, 67]. The challenges lie in knowing which addresses are congruent and how big N should be. Finding congruent addresses is only challenging when the set-selection strategy is unknown, for example, in LLC slicing on Intel CPUs. Most works use an N around the associativity of the targeted cache level [45, 65].

- Intel does not disclose the details of its replacement policy. Furthermore, they only disclosed that the caches in our system are non-inclusive of their lower-levels, but did not provide additional details. We therefore do not know into which cache level(s) the data is initially fetched, and whether the data moves to higher or lower-level caches upon eviction from a certain cache level. Yan et al. reverse engineered the cache hierarchy in the Skylake-X architecture [70]. They showed the existence of a Traditional Directory and an Extended Directory, which orchestrates the movement of cache lines through cache levels. The L2 cache in their cache hierarchy, however, is inclusive of the L1 cache, while our system's L2 is non-inclusive of L1, indicating a different directory structure.

We leave it as an open problem to determine whether there is a precise and universal strategy to evict unified caches on x86 while keeping the instruction cache intact. Such a strategy has to respond effectively to all subtleties of the targeted CPU's replacement and inclusion policy. On architectures where it is impossible to evict the protected code from the unified caches, BROL is unable to protect the code in adversarial scenario 2.

We instead evaluated BROL's effectiveness in adversarial scenario 2 on an ARMv7 system with only one active cache level. The Cortex-A7 CPU on our Banana Pi has two cache levels but allows the kernel to disable L2 together with L1d.

This effectively leaves us with a system with only an instruction cache and no data caches. We ported BROL to ARMv7 only to discover that the Linux kernel re-enables the L1d and L2 caches at regular intervals. We were, therefore, unable to evaluate BROL in its highest protective mode, i.e., when the protected code is not present in any data or unified cache. It is noteworthy that many Cortex-M CPUs have only one cache level, with separate L1i and L1d caches; for example, the Cortex-M7 [16]. On these systems, BROL can run naturally in its highest protective mode without evicting the protected code from unified caches.

Conclusion We conclude that BROL can reliably execute real-world programs of medium complexity with at most 1 KiB of executed code. BROL supports larger programs if the code complexity is low, i.e., code containing tight loops and a large portion is data-dependencent straight-line code, and in environments where only one successful execution is required among multiple invocations. The precise limit on code sizes depends heavily on the internal structure of the program and will require custom empirical evaluation. BROL can protect code in adversarial scnerario 1 and 2 on systems with only one cache level. On systems with multiple cache levels, BROL can, at present, only guaranty protection for adversarial scenario 1 due to our inability to precisely evict the protected code from the unified caches, while keeping it cached in the instruction cache.

7 Discussion

BROL's biggest threat to validity is the unwanted evictions of the preloaded protected code. As we demonstrated, the risk is especially high when only using one cache level, as required for adversarial scenario 2. In adversarial scenario 1, on the other hand, BROL can rely on multiple cache levels to store the protected code, and, therefore, tolerates an unwanted eviction from one level if the code is still present in another level.

7.1 Preloader Vulnerabilities

As discussed in Section 4, BROL provides strong protection for the program after the cache preloader completes its work. However, our approach leaves the preloading procedure itself vulnerable to attacks. The attacker has sufficient capabilities in both adversarial scenarios to pause the preloader process and inspect its memory. Below, we describe how the attacker can tamper with the preloading procedure shown in Figure 1 to circumvent BROL's protections.

- While decrypting the binary image in step $\textcircled{1}$, the preloader should query the PUF for the decryption key. In both adversarial scenarios, the attacker can intervene in this process to intercept the decryption key. With this key, the attacker can decrypt the static image and access the plain protected code.

- BROL requires memory with physical memory aliasing. It enforces this property by incorporating the effects of memory aliasing in the PUF queries and responses, and, thus, in the derived decryption key. However, in adversarial scenario 2, an attacker could mimic physical memory aliasing by introducing software-controlled aliasing in the memory management unit. The attacker can then enable aliasing while deriving the decryption key, and disable it while erasing the protected code from memory in step $\textcircled{3}$.

- The decrypted code is present in memory between steps $\textcircled{1}$ and $\textcircled{3}$. In this time window, the attacker can store the memory to disk to extract the decrypted protected code.

- Likewise, the protected code is present in the data and unified caches between step $\textcircled{2}$ and $\textcircled{3.5}$, during which the attacker in adversarial scenario 2 can read the protected code from those caches.

- The attacker could manipulate the preloader process to skip the erase step $\textcircled{3}$. This effectively disables BROL and allows the attacker to perform any reverse engineering technique at any time on the memory representation of the protected code.

- Likewise, the attacker could skip step $\textcircled{3.5}$ to keep the protected code in the data and unified caches, allowing them to read it at any time.

These attacks are only conceivable during preloading. However, preloading only runs once and takes very little time. Therefore, BROL reduces the attack surface to a limited time window rather than the full run time of the protected software.

We can protect the preloader using existing anti-tamper mechanisms that provide strong protection guarantees. Because the preloader only runs once for a limited time, we argue that it is acceptable that these measures have a high overhead. Dynamic Root of Trust for Measurement (DRTM), for example, is a mechanism that provides run-time launch integrity independent of the level of trust in the currently running software stack. It uses a secure launch program that takes control over all CPUs while it validates the environment [48]. This program starts the chain of trust and is itself integrity-protected by a measurement stored in a TPM. This technique is implemented in Windows as System Guard Secure Launch [25, 48]. TrenchBoot is an open-source implementation for Linux[3]. On its own, DRTM is insufficient to prevent piracy and reverse engineering, as attackers can still use memory analysis tools to read the protected code from

[3] https://trenchboot.org/

memory. It could, however, complement BROL and protect the preloader. As discussed in Section 3, the systems BROL targets do not contain specialized hardware such as TPMs, which are essential to existing DRTM implementations. BROL does, however, rely on a memory PUF, which could function as a hardware root of trust for a secure launch program similar to that of DRTM. In this case, we can protect code against piracy and reverse engineering with BROL while integrity-protecting BROL's preloader with a PUF variant of DRTM.

7.2 Microarchitectural Attacks

Cache-timing side channels like Flush+Reload and Prime+Probe leak the access pattern made by the protected process into a targeted memory region [45, 71]. They construct an initial cache setup where no data from the targeted region is cached. After the protected process completes its operations, the attacker performs multiple data accesses to either the shared data or to data residing in the same cache set. An attacker can determine which data the protected process used and refetched from memory based on the access times. In SMaCk, the authors increase the timing resolution using self-modifying code conflicts in L1i [59]. These side-channel attacks use frequent refetches of the targeted data from memory and, therefore, require that this data resides in memory. The access pattern into BROL-protected code cannot be analyzed using these cache-timing side channels because the targeted data, i.e., the protected code, is not resident in memory.

Speculative execution attacks like Spectre and Meltdown leverage these cache-timing side channels to leak secret data [39, 44]. They do not, however, use these side channels directly on the secret data. Instead, they leverage code gadgets in which a probe array gets indexed by the secret data. To launch the attack, they trigger the CPU to speculatively perform this indexing operation, after which they can derive the accessed indices, i.e., the secrets, by leaking the access pattern into the probe array. These attacks cannot leak BROL-protected code, as code is generally not used to index into data structures. One exception to this rule is when the protected code modifies itself. BROL, however, cannot support self-modifying code by design, as discussed in Section 4.

Microarchitectural Data Sampling (MDS) attacks, such as RIDL, Fallout, Zombieload, and CacheOut, can leak secret data while it is in transit through microarchitectural buffers within the CPU [19, 57, 64, 65]. RIDL, Zombieload, and CacheOut target the line-fill buffer between L1d and L2, while Fallout targets the store buffer. Code protected with BROL, however, does not move through this particular line-fill buffer because it does not flow to the L1d cache. Similarly, BROL-protected code cannot modify itself, nor get modified by an attacker, so it will never transit through the store buffer. It is, however, unknown whether CPUs implement similar vulnerable line-fill buffers along the instruction fetch path.

7.3 Practicality

The size of the protected software could exceed the cache size. If that happens, BROL can only protect a vendor-selected portion of the protected software that fits in the cache levels used by BROL. This is particularly limiting in adversarial scenario 2, where BROL only uses the L1i cache. Instruction cache sizes are typically in the order of kilobytes on modern high- and middle-end CPUs. Intel and ARM CPUs have at least 32 KiB instruction cache since the launch of the Core line in 2006 and the Cortex-A line in 2005, respectively [12, 32].

If the software underutilizes the available CPU cores, BROL could be modified to leverage those cores and their caches to protect a larger portion of the protected software. Alternatively, if the portion exceeds the size of the available caches, we could modify BROL so it dynamically reload parts of the protected software on demand. BROL could accomplish this by running the preloading procedure, as described in Section 4, multiple times on different small chunks that fit in the available caches. However, this increases the attack surface as the preloader now runs for a longer time. To reduce the number of required reloads, a custom compiler could lay out the code to maximize code locality, optionally aided by profiling information.

When using multiple preload iterations, BROL could protect itself, as it does for the software. By preloading the preloader code in a cache, it is protected against all attacks mentioned in Section 4, therefore, keeping the attack surface limited to the initial start-up. Due to cache size constraints, this method likely requires the caches of at least a second CPU core.

7.4 Rowhammer

The main idea in this work is to bring the system into an incoherent state where only the instruction cache holds the protected code. BROL uses physical memory aliasing to erase the protected code from memory while preventing the CPU's coherence mechanism from updating the caches. We could also reach the desired incoherence by modifying the protected code directly inside the memory module itself, for example, by using Rowhammer [38]. Rowhammer is a disturbance error in DRAM modules that allows us to flip storage bits in the memory module by rapidly accessing specific patterns of DRAM rows at high frequency. This increases charge leakage in neighboring cells until the stored logical bit flips. Rowhammer affects most commodity DDR3 and DDR4 DRAM chips [33, 37, 52], including high-end ECC RAM [20], and some DDR5 RAM [34]. The induced bit flips are, furthermore, oblivious to the OS and CPU. We could use this primitive to flip a number of bits in the protected code to erase it from memory without physical memory aliasing. However, this approach requires a large number of bit flips to make the transformation irreversible. As earlier work pointed out, Rowhammer-induced bit flips are only partially reliable [38, 47, 56]. However, for this use case only the number of bit flips in each execution matters and not the reliability of individual flips. The set of flippable bits, furthermore, depends on variances in the manufacturing process of the DRAM chip. It can, therefore, also double as the PUF used to encrypt the binary image of the protected code.

8 Related Work

Cache-as-RAM (CAR) is a technique used by bootloaders to use the CPU's caches as RAM [46, 50]. This allows bootloader developers to write in a higher-level language and, temporarily, use a stack in the cache until the DRAM controller is initialized. To enable CAR on x86, the bootloader enables the caches and marks a memory region as write-back, allowing the CPU to perform all memory operations in that region on the caches. Because the DRAM controller is inactive, this region is not backed by DRAM, so write-backs or evictions should be avoided. The bootloader can achieve this level of stability because there is no interference from other running code. This approach reaches the level of stability we want in BROL but without flexibility, i.e., only running one victim program, no operating system, no DRAM, and all code and data has to fit in the cache. With BROL, however, we aim to build a more flexible system that protects the victim code and, therefore, uses the caches only to store it.

In GlueZilla, Mechelinck et al. use the Rowhammer effect as a PUF to bind a software instance to an associated hardware instance [47]. They modify the software to exhibit a predefined unintended behavior and record the changes, i.e., the junction bits, in the binary code. At run time, they restore the software's originally intended behavior by flipping the junction bits with Rowhammer-induced bit flips. They show that their approach is practical with acceptable overhead. Their approach protects against static and dynamic analysis on cloned hardware, but remains vulnerable to memory attacks on the associated machine. In contrast, BROL protects the victim software, even when the attacker has complete control over the memory on the associated machine.

Dorfmeister et al. use PUFs to bind a software instance to an associated hardware instance to protect industrial applications from piracy [23]. They represent a program as an abstract state machine and define state changes based on the PUF responses; therefore, they couple the software's behavior tightly to the PUF. One of their concerns is to guarantee safety for the machine and the environment when the PUF returns faulty responses or when an attacker runs a stolen image on a machine different from the associated one. They use symbolic execution on the state machine to verify that the protected version of the program cannot enter unsafe states in the presence of faulty PUF responses. Dorfmeister et al. bind intellectual property in the form of neural networks to the underlying hardware [24]. They link the model's weights to the unique and unclonable hardware properties of a PUF, effectively making the model's behavior dependent on the hardware. When an attacker steals the neural network and runs it on a machine clone, the model uses different weights, yielding unexpected results. The authors show that by only binding 10-20% of the weights to the PUF, the accuracy of a text classifier already reaches that of a random classifier. In both aforementioned works, an attacker could intercept the PUF responses on the associated hardware to reconstruct the intended state changes and model weights.

Instead of protecting the IP in software, Fischer et al. protect the data used by the software [26]. They represent this data in binary form and decompose it into a list of boolean expressions that, if combined correctly, resolve to the correct value. They then combine the boolean expressions based on the responses of the PUF, effectively binding the reconstruction of the data to the machine it runs on. They validate their approach on a microcontroller board with an SRAM PUF. In contrast to BROL, this technique protects the data IP and leaves the software IP vulnerable.

In TLB;DR, Tatar et al. desynchronize the TLB to reverse engineer its implementation details including its hierarchy and replacement policy [61]. They note that TLBs do not enforce coherence with the in-memory page table. Using this insight, they deliberatly introduce incoherences between TLB entries and page table entries in memory and record whether the CPU performs an address translation using the old entry still in the TLB or using the overwritten entry in memory. This approach is similar to the deliberate cache incoherences used by BROL. Future work could reverse engineer the implementation details of the cache replacement policy in a similar fashion using deliberate incoherences between cache and memory.

9 Conclusion

We presented BROL, a system that protects software against piracy and reverse engineering while running on industrial machines with limited protective resources. BROL provides strong protection against attacks on site, and on attacker-owned systems, using only commodity hardware components. It uses physical memory aliasing to set up an environment in which the protected code is only available in the CPU caches. It achieves its strongest protection when using targeted cache eviction to make the code unavailable in any cache level except for the instruction cache. We implemented BROL for x86 and ARMv7 platforms and evaluated its practicality on code of varying complexity. We show that BROL can reliably protect real-world applications of limited size without unwanted evictions. On non-inclusive cache hierarchies, we found that evicting the unified cache levels while keeping the protected code in the instruction cache is challenging, therefore locking the full potential of BROL in adversarial scenario 2.

Acknowledgments

We thank the reviewers and our shepherd for their feedback on improving the paper. The research reported in this paper has been funded by the Federal Ministry for Innovation, Mobility and Infrastructure (BMIMI), the Federal Ministry for Economy, Energy and Tourism (BMWET), and the State of Upper Austria in the frame of the COMET Module Dependable Production Environments with Software Security (DEPS) (FFG grant no. 888338) and the SCCH competence center INTEGRATE (FFG grant no. 892418) within the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

References

[1] *perf-intel-pt(1) — Linux manual page*. URL https://man7.org/linux/man-pages/man1/perf-intel-pt.1.html. Accessed: 2025-12-23.

[2] *ARM Cortex™-R4 and Cortex-R4F Technical Reference Manual*, April 2011, Revision: r1p4. ARM DDI 0363G (ID041111).

[3] *ARM Cortex™-A7 MPCore™ Technical Reference Manual*, April 2013, Revision: r0p5. ARM DDI 0464F (ID051113).

[4] *AMD64 Architecture Programmer's Manual Volumes 1–5*, April 2024, Revision 4.08. Publication No. 40332.

[5] *Arm® Architecture Reference Manual for A-profile architecture*, April 2025. Document number: ARM DDI 0487.

[6] *ARM Cortex-A5™ Technical Reference Manual*, January 2016, Revision: r0p1. ARM DDI 0433C (ID021016).

[7] *Allwinner A40i User Manual*, June 2018, Revision 1.1.

[8] *Arm® Cortex®-A53 MPCore Processor Technical Reference Manual*, June 2018, Revision: r0p4. DDI 0500J (ID012219).

[9] *Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, June 2025. Order Number: 325462-088US.

[10] *ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition*, March 2018. ARM DDI 0406C.d (ID040418).

[11] *Intel® Itanium ® Architecture Software Developer's Manual, Volume 1: Application Architecture*, May 2010, Revision 2.3. Document Number: 245317.

[12] *ARM Cortex™-A8 Technical Reference Manual*, May 2010, Revision: r3p2. ARM DDI 0344K (ID060510).

[13] *11th Generation Intel® Core™ Processor Datasheet, Volume 1 of 2, Tiger Lake*, May 2023, Revision 012. Document Number: 631121-012.

[14] *12th Generation Intel® Core™ Processors Datasheet, Volume 1 of 2, Tiger Lake*, May 2025, Rev. 011. Doc. No.: 655258.

[15] *ARM1176JZF-S™ Technical Reference Manual*, November 2009, Revision: r0p7. ARM DDI 0301H (ID012310).

[16] *Arm® Cortex®-M7 Processor Technical Reference Manual*, November 2018 Revision r1p2. ARM DDI 0489F (ID121118).

[17] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 870–887, 2019. doi: 10.1109/SP.2019.00066.

[18] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, pages 1–18. Springer, 2001.

[19] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 769–784, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3363219. URL https://doi.org/10.1145/3319535.3363219.

[20] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting correcting codes: On the effectiveness of ECC memory against Rowhammer attacks. In *S&P*, 2019.

[21] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.

[22] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. BadRAM: Practical memory aliasing attacks on trusted execution environments. In *46th IEEE Symposium on Security and Privacy (S&P)*, May 2025.

[23] Daniel Dorfmeister, Flavio Ferrarotti, Bernhard Fischer, Evelyn Haslinger, Rudolf Ramler, and Markus

Zimmermann. An approach for safe and secure software protection supported by symbolic execution. In Gabriele Kotsis, A. Min Tjoa, Ismail Khalil, Bernhard Moser, Atif Mashkoor, Johannes Sametinger, and Maqbool Khan, editors, *Database and Expert Systems Applications - DEXA 2023 Workshops*, pages 67–78, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-39689-2.

[24] Daniel Dorfmeister, Flavio Ferrarotti, Bernhard Fischer, Martin Schwandtner, and Hannes Sochor. A puf-based approach for copy protection of intellectual property in neural network models. In Peter Bludau, Rudolf Ramler, Dietmar Winkler, and Johannes Bergsmann, editors, *Software Quality as a Foundation for Security*, pages 153–169, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-56281-5.

[25] Microsoft Enterprise and OS Security. Force firmware code to be measured and attested by secure launch on windows 10, 2020. URL `https://www.microsoft.com/en-us/security/blog/2020/09/01/force-firmware-code-to-be-measured-and-attested-by-secure-launch-on-windows-10`. Accessed: 2025-01-02.

[26] Bernhard Fischer, Daniel Dorfmeister, Flavio Ferrarotti, Manuel Penz, Michael Kargl, Martina Zeinzinger, and Florian Eibensteiner. *Software-Hardware Binding for Protection of Sensitive Data in Embedded Software*, pages 570–577. Association for Computing Machinery, New York, NY, USA, 2025. ISBN 9798400706295. URL `https://doi.org/10.1145/3672608.3707855`.

[27] Jorge Guajardo, Sandeep S. Kumar, Geert-Jan Schrijen, and Pim Tuyls. Fpga intrinsic pufs and their use for ip protection. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, pages 63–80, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74735-2.

[28] Maryam S. Hashemian, Bhanu Singh, Francis Wolff, Daniel Weyer, Steve Clay, and Christos Papachristou. A robust authentication methodology using physically unclonable functions in dram arrays. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 647–652, 2015. doi: 10.7873/DATE.2015.0308.

[29] Charles Herder, Meng-Day Yu, Farinaz Koushanfar, and Srinivas Devadas. Physical unclonable functions and applications: A tutorial. *Proceedings of the IEEE*, 102 (8), 2014.

[30] Daniel E Holcomb, Wayne P Burleson, Kevin Fu, et al. Initial sram state as a fingerprint and source of true random numbers for rfid tags. In *Proceedings of the Conference on RFID Security*, volume 7, page 01, 2007.

[31] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. Power-up sram state as an identifying fingerprint and source of true random numbers. *IEEE Transactions on Computers*, 58(9):1198–1210, 2009. doi: 10.1109/TC.2008.212.

[32] Intel. Next leap in microprocessor architecture: Intel® core™ duo processor white paper, 2006. URL `https://www.intel.com/pressroom/kits/centrino/CoreDuoWhitePaper.pdf`.

[33] Patrick Jattke, Victor van der Veen, Pietro Frigo, Stijn Gunter, and Kaveh Razavi. Blacksmith: Scalable rowhammering in the frequency domain. In *S&P*, 2022.

[34] Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Matej Bölcskei, and Kaveh Razavi. ZenHammer: Rowhammer attacks on AMD Zen-based platforms. In *USENIX Security*, 2024.

[35] Christoph Keller, Frank Gürkaynak, Hubert Kaeslin, and Norbert Felber. Dynamic memory-based physically unclonable function for the generation of unique identifiers and true random numbers. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2740–2743, 2014. doi: 10.1109/ISCAS.2014.6865740.

[36] Jeremie S. Kim, Minesh Patel, Hasan Hassan, and Onur Mutlu. The dram latency puf: Quickly evaluating physical unclonable functions by exploiting the latency-reliability tradeoff in modern commodity dram devices. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 194–207, 2018. doi: 10.1109/HPCA.2018.00026.

[37] Jeremie S Kim, Minesh Patel, A Giray Yağlıkçı, Hasan Hassan, et al. Revisiting RowHammer: An experimental analysis of modern DRAM devices and mitigation techniques. In *ISCA*, 2020.

[38] Yoongu Kim et al. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3), 2014.

[39] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[40] Florian Kohnhäuser, André Schaller, and Stefan Katzenbeisser. PUF-based software protection for low-end embedded devices. In *TRUST*, 2015.

[41] Vincent Lefebvre, Gianni Santinelli, Tilo Müller, and Johannes Götzfried. Universal trusted execution environments for securing sdn/nfv operations. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ARES '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450364485. doi: 10.1145/3230833.3233256. URL https://doi.org/10.1145/3230833.3233256.

[42] Nora Lieberknecht. Application of trusted computing in automation to prevent product piracy. In Alessandro Acquisti, Sean W. Smith, and Ahmad-Reza Sadeghi, editors, *Trust and Trustworthy Computing*, pages 95–108, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-13869-0.

[43] Daihyun Lim, Jae W Lee, Blaise Gassend, G Edward Suh, Marten Van Dijk, and Srinivas Devadas. Extracting secret keys from integrated circuits. *VLSI*, 13(10), 2005.

[44] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[45] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015. doi: 10.1109/SP.2015.43.

[46] Yinghai Lu, Li-Ta Lo, Greg Watson, and Ronald Minnich. Car: Using cache as ram in linuxbios.

[47] Ruben Mechelinck, Daniel Dorfmeister, Bernhard Fischer, Stijn Volckaert, and Stefan Brunthaler. Gluezilla: Efficient and scalable software to hardware binding using rowhammer. In Federico Maggi, Manuel Egele, Mathias Payer, and Michele Carminati, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 416–438, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-64171-8.

[48] Microsoft. System guard: How a hardware-based root of trust helps protect windows, 2025. URL https://learn.microsoft.com/en-us/windows/security/hardware-security/how-hardware-based-root-of-trust-helps-protect-windows. Accessed: 2025-01-02.

[49] Jasvir Nagra and Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[50] Eswaramoorthi Nallusamy. A framework for using processor cache as ram (car). *University of New Mexico*, 2005.

[51] OECD/EUIPO. Trends in trade in counterfeit and pirated goods, illicit trade. Technical report, 2019.

[52] Lois Orosa et al. A deeper look into RowHammer's sensitivities: Experimental analysis of real DRAM chips and implications on future attacks and defenses. In *MICRO*, 2021.

[53] Manuel Penz, Martina Zeinzinger, Michael Kargl, Florian Eibensteiner, Phillip Petz, and Josef Langer. Sram pufs for device authentication on resource-constrained systems. In *2025 9th International Conference on Cryptography, Security and Privacy (CSP)*, pages 169–176, 2025. doi: 10.1109/CSP66295.2025.00035.

[54] Ugo Piazzalunga, Paolo Salvaneschi, Francesco Balducci, Pablo Jacomuzzi, and Cristiano Moroncelli. Security strength measurement for dongle-protected software. *Security & Privacy*, 5(6):32–40, 2007.

[55] Thomas Rokicki, Clémentine Maurice, and Michael Schwarz. Cpu port contention without smt. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian D. Jensen, and Weizhi Meng, editors, *Computer Security – ESORICS 2022*, pages 209–228, Cham, 2022. Springer Nature Switzerland. ISBN 978-3-031-17143-7.

[56] André Schaller, Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, et al. Intrinsic rowhammer PUFs: Leveraging the Rowhammer effect for improved security. In *HOST*, 2017.

[57] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 753–768, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3354252. URL https://doi.org/10.1145/3319535.3354252.

[58] Boris Škorić, Pim Tuyls, and Wil Ophey. Robust key extraction from physical uncloneable functions. In *ACNS*, 2005.

[59] Seonghun Son, Daniel Moghimi, and Berk Gulmezoglu. Smack: Efficient instruction cache attacks via self-modifying code conflicts. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, page 1107–1123, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400710797. URL https://doi.org/10.1145/3676641.3716274.

[60] G Edward Suh and Srinivas Devadas. Physical unclonable functions for device authentication and secret key generation. In *DAC*, 2007.

[61] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 989–1007, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/tatar.

[62] Fatemeh Tehranipoor, Nima Karimian, Kan Xiao, and John Chandy. Dram based intrinsic physical unclonable functions for system level security. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '15, page 15–20, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334747. doi: 10.1145/2742060.2742069. URL https://doi.org/10.1145/2742060.2742069.

[63] Flavio Toffalini, Martín Ochoa, Jun Sun, and Jianying Zhou. Careful-packing: A practical and scalable anti-tampering software protection enforced by trusted computing. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, page 231–242, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450360999. doi: 10.1145/3292006.3300029. URL https://doi.org/10.1145/3292006.3300029.

[64] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105, 2019. doi: 10.1109/SP.2019.00087.

[65] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 339–354, 2021. doi: 10.1109/SP40001.2021.00064.

[66] VDMA. VDMA study product piracy 2022. Technical report, 2022. URL https://www.vdma.org/documents/34570/51629660/VDMA+Study+Product+Piracy+2022_final.pdf.

[67] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 39–54, 2019. doi: 10.1109/SP.2019.00042.

[68] Henry Wong. Intel Ivy Bridge cache replacement policy, jan 2013. URL http://blog.stuffedcow.net/2013/01/ivb-cache-replacement/. Accessed: 2025-10-26.

[69] Wenjie Xiong, André Schaller, Stefan Katzenbeisser, and Jakub Szefer. Software protection using dynamic PUFs. *IEEE Transactions on Information Forensics and Security*, 15, 2019.

[70] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 888–904, 2019. doi: 10.1109/SP.2019.00004.

[71] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[72] Martina Zeinzinger, Josef Langer, Florian Eibensteiner, Phillip Petz, Lucas Drack, Daniel Dorfmeister, and Rudolf Ramler. Comparative analysis of sram puf temperature susceptibility on embedded systems. In *2023 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, pages 1–8, 2023. doi: 10.1109/ICECET58911.2023.10389242.

[73] Dimitrios Ziakas, Allen Baum, Robert A. Maddox, and Robert J. Safranek. Intel® quickpath interconnect architectural features supporting scalable system architectures. In *2010 18th IEEE Symposium on High Performance Interconnects*, pages 1–6, 2010. doi: 10.1109/HOTI.2010.24.