reInstruct: Toward OS-aware CPU microcode reprogramming

Yubo Wang
The Pennsylvania State University
University Park, PA, USA
yubow@psu.edu

Ruslan Nikolaev
The Pennsylvania State University
University Park, PA, USA
rnikola@psu.edu

Binoy Ravindran
Virginia Tech
Blacksburg, VA, USA
binoy@vt.edu

Abstract

Historically, the microcode layer has been a proprietary technology which is tightly controlled by the CPU vendors. The microcode layer enables a great flexibility for translating ISA-visible instructions into internal hardware micro-operations. In x86-64, many system-level instructions are microcoded, which enables a great untapped opportunity for OS developers, who want to experiment with future ISA extensions.

Recent research work has identified hidden CPU instructions, which are enabled via a firmware exploit, and also partially reverse-engineered and decrypted Intel Goldmont microcode. We go a step further and design an experimental framework for Linux, which allows to transparently modify existing microcoded instructions directly from an OS at runtime. We show how microcode alterations can be used to defeat normal root-privilege isolation in Linux almost without any trace. We also show our new approach which relies on ISA modification via microcode patching to improve performance of commonly-used lightweight Linux system calls. Our approach, effectively, adjusts the CPU ISA to better serve a *specific* OS kernel and applications, an idea which has been out of reach for commodity hardware previously.

CCS Concepts: • **Software and its engineering → Operating systems**; • **Security and privacy → Operating systems security**.

Keywords: Red Unlock, microcode, Goldmont, Linux

ACM Reference Format:
Yubo Wang, Ruslan Nikolaev, and Binoy Ravindran. 2025. reInstruct: Toward OS-aware CPU microcode reprogramming. In *3rd Workshop on Kernel Isolation, Safety and Verification (KISV '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3765889.3767043

1 Introduction

Modern CPUs implement complex instructions, e.g., `CPUID`, `SYSCALL`, and `RDRAND`, through an internal control layer called *microcode*. Rather than executing such instructions as a single atomic hardware operation, the CPU dispatches them into lower-level *micro-operations* (μops) executed by a per-core microcode engine. Intel x86 has more than 2,700 distinct μops [12]; many simple instructions map to one μop, while complex ones decompose into 50+ μops [6, 11].

Microcode is crucial for architectural features, hardware bug fixes [2, 9], and platform-specific extensions. Yet it remains proprietary: engines are undocumented, updates are cryptographically signed [17], and delivery is restricted to official BIOS/OS channels [10]. Consequently, (1) system software cannot fully observe or instrument instruction behavior, and (2) researchers cannot explore alternative designs, optimizations, or vulnerabilities at the instruction level.

This work challenges this traditional assumption by showing that custom, self-authored microcode patches can run on a standard Linux system without vendor blessing. We present a Linux-based framework that dynamically patches CPU microcode. Using `RDRAND` as a case study, we present:

1. Returning constant values instead of random numbers,
2. Reading arbitrary memory while bypassing privilege checks,
3. Accessing kernel data structures, e.g., `task_struct` list,
4. Stealing a sudo password by reading its input buffer from another process in user space.

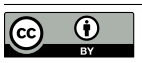
Beyond these demonstrations, we also show practical benefits of OS-driven customization. Our framework replaces lightweight system calls such as `getcpu(2)`, providing a fast path at the microcode level.

Overall, this prototype opens new opportunities for OS-based exploration of microcode behavior, performance optimizations, and instruction-level control without vendor blessing.

2 Background

2.1 Intel Microcode

Intel Goldmont (GLM) CPUs implement complex instructions using internal microcode sequences stored in hardware arrays [3, 5]. By default, microcode resides in a dedicated read-only memory (MSROM), with each instruction mapped to a fixed entry point. As we show in Figure 1, in GLM, instructions are organized into triads of three μops plus a



KISV '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2202-8/25/10
https://doi.org/10.1145/3765889.3767043

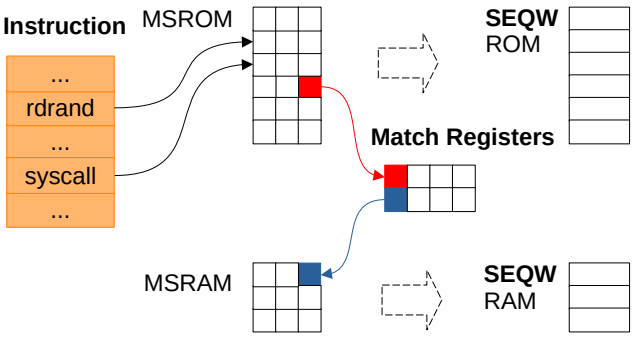


Figure 1. Microcode patching on Intel Goldmont CPUs.

sequence word (SEQW), which controls execution and can act as a synchronization primitive (e.g., LFENCE). Goldmont CPUs support up to 7,936 triads in MSROM [3].

A special writable area, MSRAM, enables *microcode patching*. GLM provides 128 triads in MSRAM, which can override MSROM microcode. Execution is redirected through *match registers*: if a microcode address in MSROM matches a register, control jumps to the corresponding MSRAM entry. Up to 32 hooks are available [3].

This design gives vendors fine-grained control over instruction semantics and post-silicon fixes, but not to system designers or researchers. Recent work has attempted to reverse-engineer and repurpose microcode [3, 5, 13, 14], challenging its presumed immutability for ordinary users.

2.2 INTEL-SA-00086

INTEL-SA-00086 is a buffer-overflow vulnerability in Intel ME, TXE, and SPS, affecting a wide range of processors (Atom, Celeron, Pentium, Core, Xeon, etc.) [8]. It enables privilege escalation and was exploited to unlock CPUs. Because ME/TXE/SPS run independently of the main CPU and OS, such escalation is nearly impossible to trace. Furthermore, the vulnerability can be exploited long after Intel has fixed it by simply downgrading the corresponding ME, TXE, or SPS version in the BIOS image, as long as the older version remains compatible with the processor.

Our unlocking method directly builds on INTEL-SA-00086 to enable access to MSRAM and match/patch structures.

2.3 Unlocking the CPU

The **CHIP-RED-PILL** project [5] leveraged INTEL-SA-00086 to expose Intel microcode internals. By reverse-engineering the PCH firmware, the authors discovered a hidden *red unlock* mechanism to enable undocumented debug interfaces on Atom and Celeron CPUs. They focused on Goldmont and Goldmont Plus, though the method can apply more broadly. Unlocking grants access to MSROM, MSRAM, and CRBUS.

Their tools support (1) dumping microcode arrays, (2) observing instruction entry points and μop sequences, (3) writing to MSRAM and match/patch tables, and (4) analyzing microcode payloads. This was the first demonstration that MSROM could be patched and arbitrary μops injected into MSRAM without vendor approval. However, the approach lacked reliability in standard environments, provided no OS integration, required boot-time or JTAG/SPI setups, and did not support real software workloads.

2.4 CustomProcessingUnit

Borrello et al. introduced *CustomProcessingUnit* [3], a UEFI-based framework for GLM microcode patching and tracing. The authors extracted update routines, built a microcode assembler/decompiler, and designed a domain-specific language (DSL) for writing custom payloads. Demonstrations included replacing INT1, INT3, and RDRAND to return fixed values or access sensitive memory regions (e.g., SMRAM).

However, CustomProcessingUnit is **not designed for commodity OSs** and has severe limitations:

- Runs only in the UEFI shell, so patches vanish after OS boot,
- Lacks multi-core support in SMP systems,
- Cannot produce Linux-visible behaviors or evaluate realistic use cases. If ported to Linux, patches would reset within ~10ms due to CPU sleep states.

These limitations motivate our work: bringing microcode patching into standard Linux.

2.5 Unlocking Issues

We found that the unlocking methods from Sections 2.3 and 2.4 fail on newer Goldmont boards. For example, we had to patch the BIOS SMIP area to avoid verification errors when incorporating the red-unlock exploit.

2.6 RDRAND Random Generator

The RDRAND instruction is widely used to generate random numbers using hardware CPU capabilities. In many of our microcode experiments, we piggyback on this instruction to avoid side effects due to CPU microcode changes. This instruction is not critical to the Linux functionality and the kernel can avoid using it altogether by specifying the nordrand parameter. Furthermore, the community does not fully trust RDRAND for security reasons, so it is often avoided by mainstream software, e.g., OpenSSL for Debian disables it [4].

3 Implementation

3.1 BIOS Modification

We modify the motherboard's underlying BIOS image and integrate an exploit similar to the previous discussion in [5]. Unlike the previous researchers, we use a different board with a newer version of Intel TXE, which we first downgrade (from v3.1 to v3.0) to enable the INTEL-SA-00086 vulnerability. We then integrate an exploit via the debug trace file into

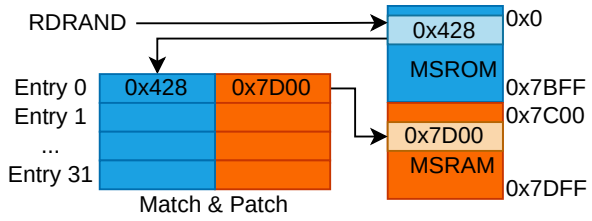


Figure 2. Microcode patch redirection for the RDRAND instruction. The original entry point is 0x428 in MSROM. We place the patch at MSRAM address 0x7d00 and register a Match-and-Patch entry mapping 0x428 to 0x7d00. Goldmont provides 32 such entries; in our experiments we precleared MSRAM and the match table, so any slot could be used. For simplicity, we chose the first entry. When RDRAND is executed, the CPU checks the table, detects the override, and executes the patched μops from MSRAM instead of the original microcode.

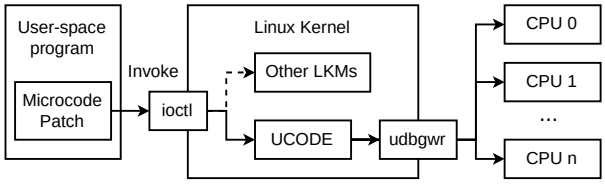


Figure 3. Overall kernel module workflow for dynamic microcode patching. A user-space program invokes ioctl() to submit a binary patch blob to our kernel module (UCODE). The module uses the undocumented Intel instruction UDBGWR (microcode debug write) to write the patch into MSRAM and update match-and-patch entries. The patch is then applied across all logical CPU cores.

the BIOS image, and additionally patch the SMIP area of the BIOS (with the original data) to bypass motherboard checks.

3.2 Patching Mechanism

In our design, we completely avoid the UEFI prompt and let the Linux kernel boot normally. We currently disable CPU power-savings features, as they may interfere and reset microcode-related modifications. (This can be revisited in the future by adding a recovery procedure when switching from a lower-power mode.)

To enable microcode patching in a standard Linux environment, we implemented a special kernel module. The kernel module writes the microcode patch into MSRAM (the base address is 0x7c00). Then it modifies the corresponding match entry, which consists of:

- The original microcode entry point to match (e.g., 0x428 for RDRAND),
- An offset into MSRAM,
- An enable bit and control flags.

When a microcode instruction is triggered, as shown in Figure 2, the sequencer first checks the match table. If a match exists, execution is redirected to MSRAM instead of MSROM, effectively replacing the instruction's behavior.

This patching architecture is undocumented but has been successfully reverse-engineered in prior work. Our implementation reuses this mechanism to override specific instructions such as RDRAND with custom μops.

3.3 Kernel Module Design

For convenience, we developed a mechanism which allows dynamic patch injection directly from *user space*, as shown in Figure 3. There are several challenges that we address in our kernel module:

- Applying microcode patches across all logical CPU cores (SMP support),
- Enabling runtime patching without reboot,
- Providing a clean user interface via `ioctl`.

Device Interface and Patch Application. The kernel module registers a character device node (e.g., /dev/ucode) and exposes a custom `ioctl` command that accepts a binary patch blob containing:

- Microcode μops implementing the desired behavior (e.g., `RAX = ADD(RBX, 0x1)`),
- Patch metadata such as the match address, MSRAM offset, and patch length.

When invoked, the module iterates over all online CPU cores, writes the patch into MSRAM via UDBGWR, and installs a match-and-patch entry to override the target instruction. (UDBGWR is an undocumented Intel instruction capable of modifying MSRAM and match-and-patch structures.) Once installed, the new microcode behavior is immediately active on all cores.

3.4 User-Space Interface

From the user's perspective, interaction with our system reduces to three steps:

1. Construct the patch blob,
2. Open /dev/ucode,
3. Invoke ioctl() to submit this patch blob:

   ```
   ioctl(fd, UCODE_PATCH_LOAD, &buffer)
   ```

   All device nodes share a common `ioctl` handler, which dispatches the request to our module when the command is UCODE_PATCH_LOAD.

Once loaded, the patched instruction (e.g., RDRAND) can be called directly from user programs via inline assembly.

3.5 Security Considerations

Our current prototype does not implement any security controls; the /dev/ucode interface is left open to demonstrate feasibility. In a realistic deployment, however, strong safeguards would be essential. We envision restricting access

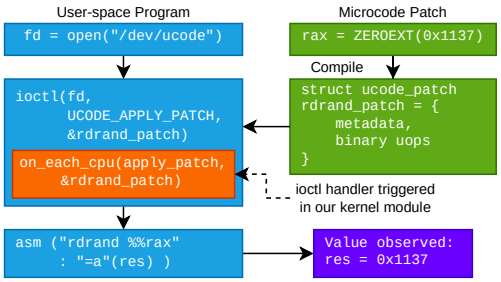


Figure 4. End-to-end patching flow for replacing RDRAND with a constant-return instruction. The patch is defined in a microcode DSL, compiled into a binary structure, and loaded via a user-space `ioctl` interface. The kernel module installs the patch on all cores and configures redirection logic. After patching, the user program executes RDRAND and receives the constant value `0x1137`.

so that only user-specified, trusted applications can install and use customized microcode. To further isolate workloads, microcode state could be treated as part of the thread context, with patches switched on context switches. This would ensure that custom instructions remain visible only to the application that installed them, preventing accidental or malicious invocation by other processes.

Modifying CPU microcode via a `/dev/ucode` interface operates under the same assumptions as accessing `/dev/kmem`. Just like `/dev/kmem`, which can modify kernel memory, microcode updates assume the same level of trust in the caller and bypass standard user-space protections. Therefore, from a security and privilege standpoint, we do not fundamentally change anything.

4 Evaluation

We evaluated our framework on a Goldmont CPU board (Intel Celeron N3350) running Debian 12 with a vanilla Linux kernel. Each demo highlights a distinct capability of our patched RDRAND instruction, including functional overrides, security bypasses, and performance optimizations. Although we tested other instructions, we use RDRAND throughout for consistency.

Patches were loaded via our kernel module (Section 3.3) using a user-space `ioctl` interface, and test programs invoked the modified instruction through inline assembly.

4.1 Replacing RDRAND with a Constant Value

To verify correctness, we first patched RDRAND to return a fixed constant value. This validates that microcode modifications are correctly loaded, triggered, and executed on hardware.

Patch Logic. The patch has one micro-op (μop) that zero-extends a 64-bit immediate value `0x1137` into a destination register (rax). In the GLM domain-specific language, this is:

```
rax := ZEROEXT(0x1137)
```

Execution Flow. Figure 4 shows the full patching pipeline. The patch is assembled into a binary format with metadata (hook address, patch address, etc.) and encoded μops, wrapped in a C structure, and passed to user space. The user program opens the character device exposed by the kernel module and installs the patch via an `ioctl` call. The kernel module writes it to MSRAM and updates the Match-and-Patch table, redirecting RDRAND from address 0x428 to the patch at 0x7d00. After patching, executing RDRAND in user space returns `0x1137`, confirming successful patching.

4.2 Arbitrary Memory Read

In this experiment, we show that a modified RDRAND instruction can act as a generic memory read primitive, capable of accessing **any** virtual address without triggering protection mechanisms, i.e., accessing kernel-space addresses from user space. The patch logic is minimal: we replace the original microcode with a single load operation that reads memory from the address in `rax` and returns the value in `rbx`:

```
rbx := LDZX(rax)
```

This instruction behaves as an unchecked dereference: it does not perform any permission checks and succeeds even when accessing kernel memory. (Due to the in-kernel 1:1 physical map, we can even access other processes.) The patching and installation process is identical to Figure 4.

Once the patch is installed, a user-space program passes a desired address via `rax` and invokes RDRAND to retrieve the 64-bit value stored at that address.

4.2.1 Reading Kernel `task_struct` List.

As a first demonstration, we use the patched RDRAND to traverse the kernel's `task_struct` list. Starting from the global `init_task` symbol, we follow the doubly-linked `tasks` list to enumerate all running processes.

For each task, we use RDRAND to retrieve in-kernel memory fields such as the process ID (PID) and the `comm` string. Since the patched RDRAND bypasses all privilege checks, this works from user space without requiring any system calls, kernel modules, or debugging interfaces.

The output shown in Figure 5 confirms that our program can accurately identify and print kernel process metadata entirely via microcode-based memory reads. This experiment demonstrates that a single μop (`rbx := LDZX(rax)`) is sufficient to leak arbitrary kernel structures from user space.

4.2.2 Stealing sudo Password Input.

We then show a more impactful demonstration: stealing a password typed into a sudo prompt from another user. In our test scenario, an

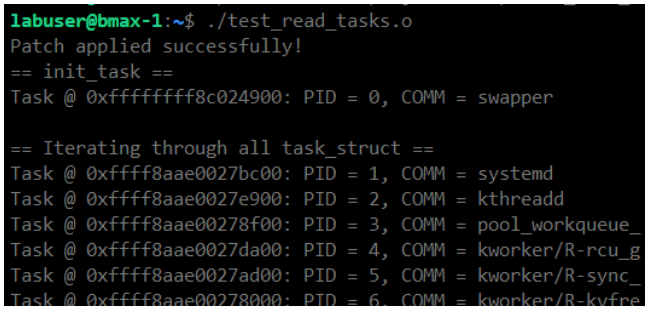


Figure 5. Reading the kernel's `task_struct` list using patched RDRAND. The user-space program starts from `init_task`, follows the `tasks` list, and prints each task's PID and `comm` field, similar to the output of `ps`.

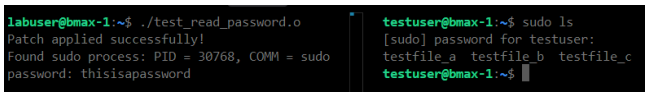


Figure 6. Real-time password extraction via patched RDRAND. Left: the attack program, running as `labuser`, identifies the `sudo` process and reads the cleartext password `thisisapassword` from its input buffer. Right: the victim user `testuser` enters the password into a `sudo` prompt, which does not echo input.

unprivileged user-space program (run as `labuser`)[1] continuously scans the `task_struct` list for any process that uses `sudo`. Once found, it uses patched RDRAND to probe memory regions of the target process and locate its input buffer.

As shown in Figure 6, this technique allows us to extract the cleartext password (`thisisapassword`) entered by another user (`testuser`) in real time, despite having no privileges or direct interaction.

This highlights a realistic attack scenario: in shared environments such as university labs or research clusters, users often have `sudo` rights on their own machines, but not on others. If an attacker with local microcode patching capability can extract the password of an IT administrator during a maintenance session, they may gain access to additional systems where that password is valid.

The most concerning aspect of this attack is its invisibility: since the memory read is performed entirely at the microcode level, the operating system has no way to detect or log it. From the OS's perspective, the attack program is simply executing the RDRAND instruction multiple times—an entirely legal and unprivileged action. No system calls, page faults, or kernel transitions occur during the attack, leaving no trace in audit logs, syscall monitors, or traditional detection tools.

[1]We do not even require root privileges for this program. Although to insert a kernel module, we typically need corresponding permissions on this *specific* machine, we can retrieve password for *any* admin user who potentially has access to a variety of *other* machines with the same password.

This illustrates a new class of *stealthy microarchitectural attacks*, where malicious behavior is implemented not in software, but in hardware-level control logic that the OS cannot observe or defend against.

4.3 Replacing `getcpu()` with Microcode for Performance Optimization

We also explore the potential for OS-aware optimizations at the microcode level. One specific area of optimization is replacing expensive OS system calls with customized microcoded calls, which directly access kernel memory. A fair amount of Linux system calls can be repeatedly used by various programs and can potentially be replaced: `getcpu(2)`, `getuid(2)`, `getpid(2)`, `getppid(2)`, `uname(2)`, `getcwd(2)`, `getrlimit(2)`, `sysconf(2)`, etc. Many of these calls are uncachable in user space since their output depends on the current kernel state or other system calls, e.g., `setuid(2)`.

In our experiment, we reimplement the functionality of the `getcpu(2)` system call entirely at the microcode level.

Design. The `getcpu(2)` syscall returns the *current* logical CPU core ID [15]. To emulate this functionality, we install different microcode patches on each core. Each patch replaces RDRAND with a constant-return μop, as in Section 4.1, but the returned constant is set to that core's ID:

- On CPU 0: `rax := ZEROEXT(0x0)`
- On CPU 1: `rax := ZEROEXT(0x1)`
- On CPU 2: `rax := ZEROEXT(0x2)`, etc.

Usage. After patching, a user-space program can retrieve the current core ID by simply executing the RDRAND instruction. No system call, no kernel transition, and no vDSO lookup is required. This acts as a direct, one-instruction replacement for `getcpu()`.

Performance. To measure the latency of each approach, we run the corresponding code in a tight loop with many iterations and compute the average cycle count using the RDTSC instruction. Our results are summarized below:

- **Traditional `getcpu()` syscall:** 511 cycles
- **Microcode-based `getcpu()`:** 43 cycles

This demonstrates that microcode can be used not only to override instruction semantics, but also to significantly accelerate performance-critical kernel functionality. In our experiments, the microcode-based version is **12× faster** than the traditional syscall path. In applications with frequent per-core optimizations (e.g., schedulers, packet routing, NUMA-aware tasks), such microcode-based fast paths may offer significant benefits.

Discussion. While this use case is relatively simple, it points to a broader possibility: lightweight syscall offloading into the microcode layer. Unlike traditional syscalls, microcode execution avoids mode switches and stack changes, offering a path toward OS-aware optimization.

Although the measured latency of 43 cycles may appear high compared to the native `RDRAND` instruction (about 9 cycles), it still represents a 12× speedup over the traditional `getcpu()` syscall path. We suspect that this overhead stems from inherent microcode redirection costs and MSRAM access latency. Further investigation and optimization are part of our future work, with the goal of approaching the cost of native microcode execution.

5 Future Work

Looking ahead, we see several directions to extend this work.

5.1 Fine-Grained Microcode Customization

We plan to support more flexible deployment strategies:

- *Per-core patching*: By applying different microcode versions to different cores, we can optimize low-level primitives such as `getcpu(2)` for local contexts, improving multi-core efficiency,
- *Per-thread patching*: This enables different applications (or even different threads within the same application) to use microcode tailored to their specific workloads. For instance, one thread may benefit from a patched floating-point operator while another prefers a custom memory access pattern,
- *Per-phase patch switching*: Applications with distinct execution phases (e.g., training vs. inference, initialization vs. steady-state, mapping vs. reducing) could dynamically switch between optimized microcode variants suitable for each phase.

5.2 Improved Tooling and Framework Support

Our prototype is functional but low-level. We aim to develop a robust toolchain with better debugging, compatibility checks, and a DSL for patch specification. MSROM already reuses functions across instructions (e.g., privilege checks in virtualization). Our DSL should capture such higher-level structures, supporting function reuse, modular composition, and richer control flow. Achieving this requires stronger patch representation, code generation, and linking. Fine-grained customization (Section 5.1) also demands infrastructure integration, such as embedding patch-switching into Linux context switches.

5.3 More Powerful Patch Capabilities

With deeper understanding and more flexible patching mechanisms, we aim to: (1) simplify redundant safety checks in trusted contexts, (2) repurpose microarchitectural registers to reduce memory traffic, and (3) apply fine-grained μop optimizations (e.g., eliminating mutually canceling actions). Our current grasp of the `SEQW` mechanism and CRBUS addresses is incomplete, limiting patch scope. We plan to further reverse-engineer undocumented features and integrate them into a more capable toolchain.

5.4 Other CPU Architectures

INTEL-SA-00086 has also been applied to Skylake [1]. Moreover, recent AMD processors, including Zen 5, have also been found to be vulnerable to microcode changes without vendor blessing [16]. These architectures still require microcode decryption and reverse engineering to better understand their microcode formats. While our current focus is Goldmont, we plan to port the framework as more details emerge.

6 Conclusion

This work presents *reInstruct*, a prototype framework that enables dynamic microcode patching from user space on x86 CPUs. We show that microcode, which is traditionally considered opaque and tightly controlled by vendors, can be exposed as a programmable system interface, opening new possibilities for application-specific CPU behavior.

Our framework applies and switches microcode patches at runtime via a Linux kernel module. Case studies demonstrate both performance gains, such as a microcode-based syscall replacement, and security risks, such as unauthorized memory reads.

To our knowledge, this is the first system to enable such flexible user-space microcode changes on commodity hardware, entirely within the standard Linux stack. By allowing software to adapt hardware behavior dynamically, we open paths for research in custom acceleration, μop-level tuning, patch scheduling, and OS-level micro-architectural control.

Unlike prior approaches such as Alpha's PAL code or RISC-V custom extensions, which require vendor support and pre-silicon changes, our framework emphasizes *dynamic programmability*. Commodity CPUs can be specialized in software, with optimizations tailored to workloads and even switched at context-switch time, enabling per-application or per-phase specialization beyond static ISA customization. In trusted settings, users may even relax or remove certain safety checks after careful consideration of their needs, an option infeasible for vendor-supplied microcode, but one that highlights the greater flexibility of our approach.

We envision evolving this prototype into a safe and flexible dynamic microcode interface, akin to the eBPF framework [7] for microarchitectural behavior: effectively an "eBPF for microcode" that lets developers extend and specialize CPU functionality without leaving the standard OS environment.

Availability

We plan to release all aspects of our implementation as open-source software at a future publication date.

Acknowledgments

We thank the anonymous reviewers for their suggestions and feedback that significantly helped improve the paper.

This work is supported in part by the US Office of Naval Research (ONR) under grant N000142412642.

References

[1] Youness Alaoui. 2019. Exploiting Intel's Management Engine - Part 2: Enabling Red JTAG Unlock on Intel ME 11.x (INTEL-SA-00086). https://kakaroto.homelinux.net/2019/11/exploiting-intels-management-engine-part-2-enabling-red-jtag-unlock-on-intel-me-11-x-intel-sa-00086.

[2] AMD. 2023. White paper: Software Techniques for Managing Speculation on AMD Processors – Revision 5.09.23. https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf.

[3] Pietro Borrello, Catherine Easdon, Martin Schwarzl, Roland Czerny, and Michael Schwarz. 2023. CustomProcessingUnit: Reverse Engineering and Customization of Intel Microcode. In *IEEE Workshop on Offensive Technologies (WOOT 23)*. doi:10.1109/SPW59333.2023.00031

[4] Debian. 2022. Accepted openssl 3.0.7-1 (source) into unstable – Disable rdrand engine (the opcode on x86). https://tracker.debian.org/news/1380694/accepted-openssl-307-1-source-into-unstable.

[5] Mark Ermolov, Dmitry Sklyarov, and Maxim Goryachy. 2021. CHIP RED PILL: How We Achieved to Execute [Micro]code Inside Intel Atom CPUs. https://github.com/chip-red-pill. Presentation, Positive Technologies.

[6] Agner Fog. 2025. The microarchitecture of Intel, AMD and VIA CPUs. https://www.agner.org/optimize/microarchitecture.pdf. Section 16.2.

[7] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (Heraklion, Greece) *(CoNEXT '18)*. Association for Computing Machinery, New York, NY, USA, 54–66. doi:10.1145/3281411.3281443

[8] Intel. 2018. INTEL-SA-00086. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00086.html.

[9] Intel. 2018. White paper: Intel Analysis of Speculative Execution Side Channels. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/analysis-speculative-execution-side-channels.html.

[10] Intel. 2020. Microcode Update Guidance. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/microcode-update-guidance.html.

[11] Intel. 2025. Intel 64 and IA-32 Architectures Software Developer Manuals, Volume 3: System Programming Guide. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[12] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. 2009. Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation. In *Computer Aided Verification*, Ahmed Bouajjani and Oded Maler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 414–429. doi:10.1007/978-3-642-02658-4_32

[13] Benjamin Kollenda, Philipp Koppe, Marc Fyrbiak, Christian Kison, Christof Paar, and Thorsten Holz. 2020. An Exploratory Analysis of Microcode as a Building Block for System Defenses. arXiv:2007.03549 [cs.CR] https://arxiv.org/abs/2007.03549

[14] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. 2017. Reverse Engineering x86 Processor Microcode. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1163–1180. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/koppe

[15] Linux manual page. 2025. System Calls Manual: getcpu - determine CPU and NUMA node on which the calling thread is running. https://man7.org/linux/man-pages/man2/getcpu.2.html.

[16] Google Security Team. 2025. AMD: Microcode Signature Verification Vulnerability. https://github.com/google/security-research/security/advisories/GHSA-4xq7-4mgh-gp6w.

[17] Zhiguo Yang, Qingbao Li, Ping Zhang, and Zhifeng Chen. 2020. Reverse Engineering of Intel Microcode Update Structure. *IEEE Access* 8 (2020), 169676–169687. doi:10.1109/ACCESS.2020.3024243