Theme Article

Context-Sensitive Decoding: On-Demand Microcode Customization for Security and Energy Management

Mohammadkazem Taram
University of California

Ashish Venkat
University of Virginia

Dean M. Tullsen
University of California

***Abstract*—Modern instruction set decoders feature translation of native instructions into internal micro-ops to simplify the CPU design and improve instruction-level parallelism. However, this translation is static in most known instances. This paper proposes *context-sensitive decoding*, a technique that enables customization of the micro-op translation based on the current execution context and/or preset hardware events. Further, it can transition between different translation modes rapidly. While there are many potential applications, this paper demonstrates its effectiveness with two use cases: 1) as a novel security defense to thwart instruction/data cache-based side-channel attacks; and 2) as a power management technique that performs selective devectorization to enable efficient unit-level power gating.**

The post-Dennard scaling era has witnessed an upsurge in the adoption of specialized processing elements to improve the execution efficiency of diverse workloads. This paper exploits an underutilized feature of modern instruction set decoders to show that even general-purpose processors can be customized, and in fact that customization can be seamlessly configured dynamically at an extremely fine granularity.

Digital Object Identifier 10.1109/MM.2019.2910507
Date of publication 11 April 2019; date of current version 8 May 2019.

Published by the IEEE Computer Society

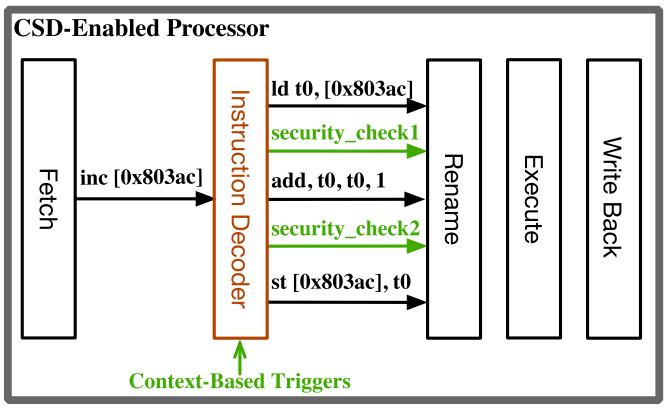

Figure 1. Under normal operation, an x86 decoder would translate the *increment* instruction into three micro-ops (in black). With CSD, that instruction would either be translated the same way, or depending on various possible triggers, could be translated into more micro-ops, for example with extra security checks (in green).

The key to this change is the fact that most modern processors employ translated ISAs, as the Intel and AMD x86 processors and many ARM processors typically feature translation from the native instruction set into internal micro-ops that enter the pipeline for execution. These architectures enjoy the dual benefits of a versatile backward-compatible CISC front-end and a simple cost-effective RISC back-end. However, for those architectures the translation is static, changing once per generation. Instead, we propose that translation be dynamic, potentially changing frequently within the execution of a single program.

This research unlocks the full potential of translated ISAs via context-sensitive decoding (CSD), a technique that allows native instructions to be decoded/translated into a different set of custom micro-ops based on their current execution context (see Figure 1). This presents operating systems, runtime systems, and antivirus programs with the unique opportunity of triggering different custom translation modes, at microsecond or finer granularity. In this way, for example, an insecure executable can instantly become a secure executable, or performance-optimized code can become energy optimized, without recompilation or binary translation.

CSD is a mechanism that is an enabler for a multitude of architectural solutions. In particular, it is a new approach that enables subtle changes to the execution model and instruction stream that can be applied at an extremely fine granularity. Further, it can be applied to just about any application, even postcompilation and postdeployment.

There are clear applications of CSD to improve performance, energy efficiency, usability, and debugging, etc. However, in the area of security, this research is particularly timely. The importance of the architecture being part of the security solution has become painfully apparent. Also apparent is the critical need for rapid deployment of security solutions, with a new variant of the Spectre attacks[1] seemingly being revealed every couple of weeks.

Traditionally, software solutions to new attacks require significant developer effort, extensive software patching and recompilation, or worse reverting to translation or interpretation. With CSD, we can quickly deploy a new defense which 1) is immediately universal (no recompilation of existing software); 2) is a hardware/architectural solution and thus inaccessible and invisible to attackers; and 3) owing to its fine-grained reconfigurability, can be deployed surgically only where needed,[2,3] preserving performance.

The CSD framework we describe allows custom translation modes to be triggered by events such as hotspot detection, unit-criticality predictors, thread-criticality predictors, protection-domain crossings, interception of a tainted input, a watchdog timer event, changes in power or energy availability, or thermal events—all with no significant changes to the pipeline or the ISA—a major shift from prior work on firmware-driven binary translation and dynamic instruction stream editing[4] that require complex pattern-matching and user-defined production rules to be integrated into the decoder framework.

The hardware and architectural barriers to CSD are quite low. This requires minimal changes to the architecture—only the decoder and some related structures. Even the decoder change is primarily adding a new table to an existing table lookup. In fact, virtually all of the building blocks of the CSD framework leverage existing functionality implemented in most modern RISC and CISC processors.

CSD has potential applications in areas such as malware detection and prevention, dynamic information flow tracking (DIFT), runtime profiling and performance programming, on-demand type-safety, program verification and debugging, and runtime phase tracking and code specialization. This work showcases two diverse applications of CSD—an obfuscation-based security defense against cache-based side channel attacks, and criticality-aware power gating to improve the energy efficiency.

CONTEXT-SENSITIVE DECODING

In this section, we provide a brief overview of the x86 front-end, describe techniques to enable CSD in an x86 architecture, and discuss potential applications.

The x86 front end in Figure 2 has two major components: 1) the legacy decode pipeline that translates native instructions into micro-ops; and 2) a micro-op cache that delivers already translated micro-ops into the instruction queue.

The legacy decode pipeline first decodes the variable-length x86 instructions and inserts them into a macro-op queue, which then feeds instructions into one of the four decoders that translate them into micro-ops. These decoders use a static table-driven approach for the micro-op translation. In fact, only one of the decoders can translate an instruction to more than one micro-op, with the other three performing a simple one-to-one mapping operation. Complex instructions that decompose into more than four micro-ops are microsequenced by an MSROM.

The translated micro-ops are cached in a micro-op cache. The front-end then streams micro-ops from the micro-op cache into the instruction (micro-op) queue until a miss occurs, at which point it switches back to the legacy decode pipeline. The front-end also sports a number of optimization features such as stack-pointer tracking, micro-op fusion, macro-op fusion, and loop stream detection.

CSD-Enabled x86 Front End

Figure 2 highlights necessary hardware components required to enable CSD in the x86 front-end.

Integration with the Legacy Decode Pipeline. To enable CSD in the x86 front-end, we provision

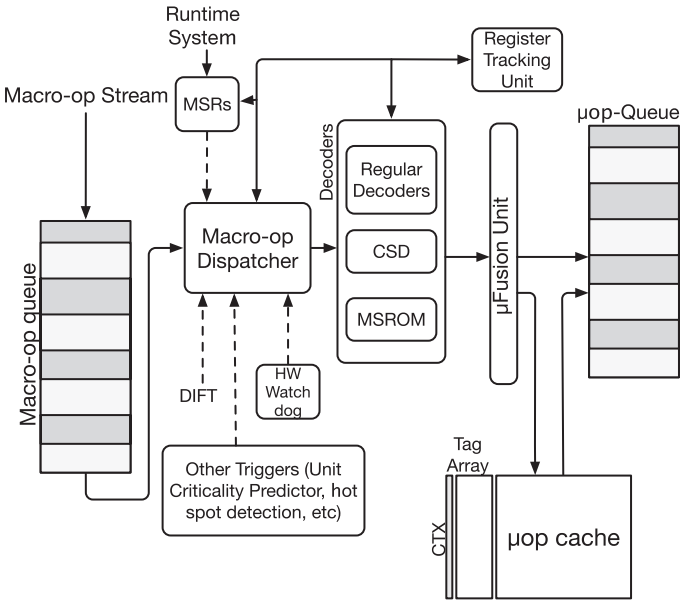

Figure 2. Intel front end with CSD support.

the legacy decode pipeline with one or more custom decoders that perform custom translations. These decoders continue to employ a simple static table-driven translation model, like the four native x86 decoders. However, they can generate more sophisticated micro-op flows by relegating to the MSROM.

When CSD is turned ON, we redirect macro-ops that require custom translation to the custom decoder. In our initial implementation, this logic can be triggered in three different scenarios: via OS-filled model-specific registers (MSRs), by hardware events such as the interception of a tainted input by DIFT or a power-gating event from the unit criticality predictor, or from a hardware watchdog timer which periodically triggers a translation mode switch.

Interactions with the Micro-Op Cache. The micro-op cache is an important performance and energy optimization. Thus, CSD is designed to work well with the micro-op cache so that most micro-ops are served by the micro-op cache, even when frequent custom translations are employed. We extend the tag bits of the micro-op cache with an additional set of context bits that associate a particular micro-op way with the decoder that translated it. This allows us to improve the micro-op cache utilization by co-locating micro-op translations from different custom decoders.

MicroCode Update (MCU) and Autotranslation

CSD exploits the already existing MCU procedure of Intel processors[5] to empower the runtime system with the ability to inject custom translations into the microcode engine, with the API provided to the runtime being the entire x86 instruction set. The CSD framework further *auto-translates* such MCUs by exploiting Intel's existing front-end translation and optimization infrastructure.

Since MCU is performed via a privileged instruction or system call, only trusted entities within the OS/runtime system should have the ability to successfully inject MCUs into the processor.

Performance Optimizations

The micro-op expansion due to customization could potentially have a negative impact on performance if the custom translations are not optimized, or not integrated well with the rest of the front end, before they are injected into the dynamic execution stream. Therefore, we carefully craft the new micro-op flows to maximize the use of micro-op fusion, and we create short and tight loops that maximize micro-op cache utilization.

Potential Applications

Later sections of this paper focus on particular applications of CSD including 1) adding security on-demand; and 2) selectively moving vector computation on and off the vector unit to minimize energy. However, other potential applications abound.

These include runtime dynamic type checking, debugging (nearly cost-free breakpoints and watchpoints, without altering the binary at all), endlessly configurable performance counters without current hardware limits to the quantity of counters, and profiling and instrumentation support, again with no changes to the binary (which among other benefits gives true instruction cache behavior).

CASE STUDY I:SIDE-CHANNEL DEFENSE

In this section, we demonstrate the security potential of CSD, demonstrating one possible use of CSD to eliminate any visibility in the cache of data-dependent accesses to sensitive data structures, or sensitive-data dependent control flow. We first lay out our assumptions and threat model, then describe the stealth-mode translation feature of CSD. Finally, we leverage this feature to secure commercial implementations of RSA and AES against the two major data and instruction cache side channel attacks.

Assumptions and Threat Model

Trusted Computing Base. We assume that the micro-op engine (including the micro-op cache) is tamper-proof and is a part of the trusted computing base (TCB). We also further extend the TCB to include all hardware or software mechanisms that trigger CSD. However, the instruction stream itself need not be trusted.

Attacker Environment. We assume an active attacker who can effortlessly probe, flush, or evict a co-located victim's cache lines, but does not have direct access to the contents in the cache. We also assume that the attacker has the ability to make precise timing measurements and has unlimited access to performance counters. This allows them to make inferences about the victim by observing the cache characteristics.

Stealth-Mode Translation

Cache-based side channel attacks typically involve probing one or more cache lines of a co-located victim in order to capture its memory access patterns, and reveal secret information. For example, an attacker who intends to break a cryptographic algorithm could compute one or more bits of a secret key by capturing access patterns of key-dependent loads and branches. In this specific implementation, we obfuscate a victim's control path and/or access to sensitive data structures. In particular, we use *decoy micro-ops* that load data into the caches that would be touched on all data-dependent paths. These include all cache blocks that contain T-tables of AES and the *multiply* functions of RSA.

Figure 3 shows context-sensitive decoding with stealth-mode translation in action. Stealth-mode translation is primarily triggered by updates to register-tracked *decoy address-range registers*, similar to the already existing memory type range registers (MTRR)[5] in x86. The decoy address range registers allow antivirus and other hardware/software trusted entities to mark specific data and instruction address ranges

in a program's address space as sensitive. As soon as stealth-mode translation is triggered, these decoy address ranges are copied to the context-sensitive decoder's internal registers; after that, the macro-op dispatcher starts redirecting all loads and branches within the PC range for custom translation.

CSD thereby injects *decoy micro-ops* into all load, store, or branch-containing instructions during stealth-mode translation. Figure 4, as an example, shows stealth-mode translation of the MOV instruction for cache-based side-channel prevention. In this example, CSD injects a micro-loop into the micro-op stream. The micro-loop obfuscates the architectural state by loading all sensitive cache blocks whose addresses have been specified in the MSRRs.

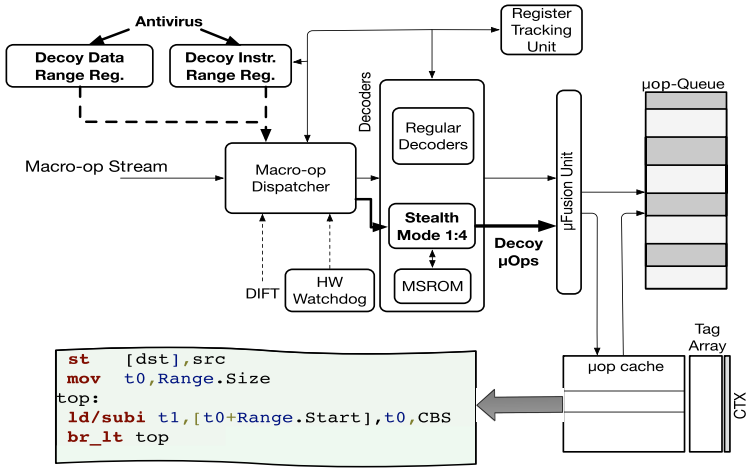

Figure 3. CSD with stealth mode.

We examine two schemes of translation—one for a software anti-virus-driven stealth-mode configuration where the tainted program counter (PC) values are known *a priori* with the help of binary analysis and configured in specific MSRs, and one for architectures that implement full hardware DIFT. In both instances, the *decoy micro-ops* execute only for tainted instructions—for the DIFT-enhanced architectures, this decision is made dynamically at run time.

We do not need to load the decoy structures constantly, since they will stay in the cache for a time, or until the attacker removes them. Thus, stealth-mode translation automatically turns itself off after one affected load (ensuring that all addresses in the range are loaded into the cache). However, before turning itself off, CSD starts the hardware watchdog timer in order to periodically trigger stealth-mode translation. It is important to carefully configure the watchdog's timeout period to a value that is smaller than the attacker's best possible probe interval period, but large enough to minimize the performance degradation caused by the *decoy micro-ops* now flowing through the pipeline.

Securing the Instruction Cache Running RSA

Instruction cache-based attacks have been able to subvert prominent cryptographic algorithms, such as RSA, by being able to capture their key-dependent instruction access patterns. Commercial implementations of RSA, such as PGP and GnuPG, use the *square-and-multiply* algorithm to speed up modular exponentiation, which iterates over the binary representation of the exponent to selectively perform exponentiation using three major suboperations: *square*, *multiply*, and *reduce*. While *square* and *reduce* are performed each loop iteration, *multiply* is invoked only when the exponent bit is 1, invariably entailing a key-dependent branch.

These suboperations are implemented as large functions that span multiple cache blocks. This implies that I-cache access patterns

```
MOV_M_R(reg_t src, reg_t dst){
 asm(
  "mov [dst],src     ;real store "
 );
 for (int i=Range.start; i<Range.End; i+=cache_blk_size){
     asm (
        "mov ebx,[eax+Range.Start]     ;decoy load"
     );
 }
}
```

(a) Pseudocode

```
MOV_M_R(reg_t src, reg_t dst){
 mov [dst],src      ;real store
 mov eax,Range.Size     ;initialize eax
top:
 mov ebx,[eax+Range.Start]   ;decoy load
 sub eax, cache_blk_size   ;point to next cache block
 jl top          ;iterate over all cache blocks
}
```

(b) X86 Instructions

```
MOV_M_R(reg_t src, reg_t dst){
 st   [dst],src          ;real store
 mov  t0,Range.Size      ;initialize t0
top:
 ld/subi t1,[t0+Range.Start],t0,cache_blk_size;fused ld,sub
 br_lt top          ;iterate over all cache blocks
}
```

(c) Micro-ops

Figure 4. Translation of MOV instruction in stealth mode.

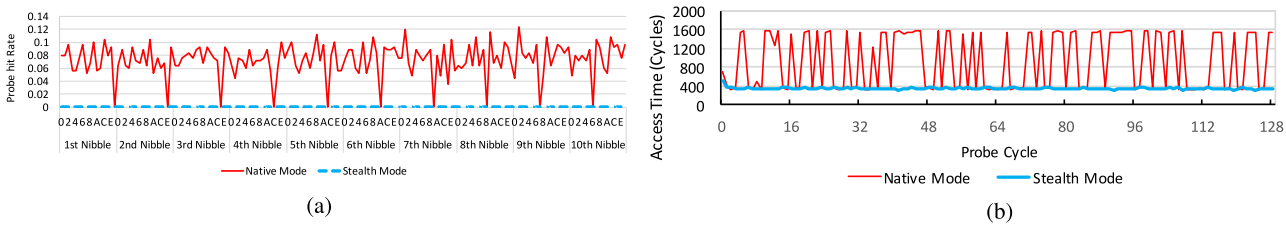

Figure 5. Effect of the cache attacks on AES and RSA with stealth-mode translation enabled. (a) PRIME+PROBE on AES. (b) FLUSH+RELOAD on RSA.

potentially reveal bits of the exponent. A PRIME+PROBE attack on RSA[6] fills up the I-cache set for the *multiply* operation in the prime phase, and probes for it after a carefully chosen interval to check if the victim evicted its block. A FLUSH+RELOAD attack[7] flushes the cache line that corresponds to the *multiply* routine, and reloads it after a carefully chosen probe interval—if that load is quick, it indicates the victim has accessed the code.

Effect of Stealth-Mode Translation. As a defense mechanism against I-cache attacks, we use stealth-mode translation to periodically inject decoy instruction-load micro-ops into the pipeline. Stealth-mode can be enabled for RSA by configuring the instruction *decoy address range* MSRs with the address-range of the *multiply* functions. By periodically loading the right set of cache blocks into the I-cache, stealth-mode successfully obfuscates the instruction access pattern that is perceived by the attacker.

Securing the Data Cache Running AES

Attackers have also exploited data-cache side channels to infer secret information by observing a victim's key-dependent data access patterns. Bernstein's original chosen plaintext attack[8] targets the AES algorithm, a substitution-permutation block cipher that performs several rounds of simple substitution and permutation during encryption. Several software implementations including OpenSSL employ lookup tables called T-tables to speed up these rounds, which then include several simple table lookups. The index computation for the T-table lookup turns out to be a key-dependent load.

As with I-cache attacks, a PRIME+PROBE attack fills up the D-cache sets for one or more of these T-table blocks in the prime phase, and probes for them after a certain interval to check if the victim made an access to any of the primed sets. A FLUSH+RELOAD D-cache attack exploits deduplication in order to flush one or more of the T-table blocks, and reload them after a carefully chosen probe interval.

Effect of Stealth-Mode Translation. By configuring the data *decoy address range* MSRs with the appropriate address range of the T-tables, we can successfully obfuscate the key-dependent data access patterns.

Securing Other Side-Channels.

Although the primary focus of this paper is to defend against cache-based side-channel attacks, we note that the stealth-mode translation feature of CSD could be exploited in the future to defend against other timing and physical attacks. For example, the decoy micro-ops could alter the branch predictor tables and BTB entries to confuse branch prediction analysis attacks, or potentially introduce a random stream of NOPs (and different types of NOPs) to skew timing analysis. Furthermore, owing to its ability to microsequence instructions using the MSROM, it can add additional noise into the sensed power by nondeterministically microsequencing instructions that cause switching activity across different microarchitectural structures.

CASE STUDY II: UNIT-LEVEL POWER GATING

In this section, we present another use case of CSD, selective devectorization for unit-level power-gating. While similar in functionality to software-based devectorization approaches proposed in prior work,[9,10] we eliminate binary translation costs and related cache effects, add the ability to switch modes at a finer granularity, allow cheaper and more effective monitoring, and provide more direct control over power gating and ungating.

Unit-level power gating is one of our primary tools to reduce the leakage power. However, upon encountering demand for that unit, it must connect back to the supply voltage. Powering a unit on and off uses power and energy but also slows execution, typically stalling until the unit is activated. We leverage CSD to enable efficient and more fine-grained power gating of vector units. CSD does this in two ways: 1) for infrequent vector instructions, it avoids powering on the unit by translating vector instructions to equivalent scalar micro-ops; and 2) when it does need to power ON, it hides the power-on delay by continuing the execution of instructions using scalar mode until the unit is ready.

For this optimization, we employ a simple counter that tracks vector instructions over a window of instructions. When it goes below a threshold, it turns on devectorization and powers off the entire vector unit, and when it goes above a (higher) threshold, it turns the vector unit back on. When devectorization is enabled, the microcode engine translates the vector instructions to an equivalent set of scalar micro-ops using the alternative CSD decoder.

METHODOLOGY

We model the x86 pipeline using the gem5 architectural simulator, which already features micro-op translation. We further extend the gem5 front-end to include a micro-op cache and support micro-op fusion as described in the Intel Architectures Optimization Reference Manual. Our baseline processor is based on the Intel Sandybridge microarchitecture, adapted to the instruction queue model of gem5. The modeled processor is a 3.3-GHz, 4-issue out-of-order superscalar processor with 32-KB L1 instruction and data caches.

We evaluate the security of CSD on two commercial implementations of cryptographic algorithms: OpenSSL's AES and GnuPG's RSA. We also include two additional security benchmarks from the MiBench suite that are vulnerable to data cache based side channel attacks for performance analysis. Each of our security applications can be run in two distinct modes (encrypt and decrypt), giving us eight performance datapoints.

To evaluate the selective devectorization mode, we use a wide range of high and low ILP applications from the SPEC CPU2006 suite.

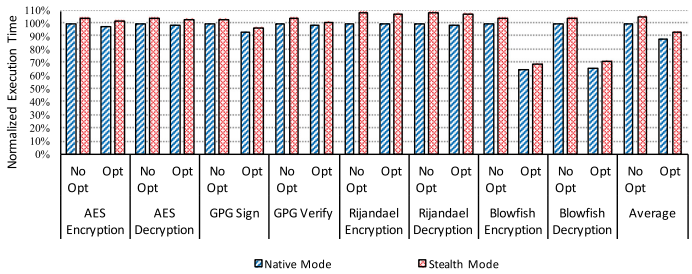

Figure 6. Execution time impact of CSD when implementing secure cache obfuscation normalized to insecure execution mode. The watchdog timer is set so that secure mode is reentered every 500 μs.

RESULTS

Stealth Mode Translation

Security Evaluation. Figure 5(a) shows the results of the PRIME+PROBE attack on AES. The attack repeatedly triggers encryptions with carefully chosen plaintexts while probing 16 different addresses of the AES T-tables. For each probe, only one plaintext has a 100% hit rate (steep dips in the curve), revealing 4 bits of the key. When the stealth-mode translation is not enabled, 64 bits out of the 128 bits of the key get compromised after 64,000 attempts. However, when stealth-mode translation is enabled, the attacker-perceived data access patterns are completely obfuscated and result in a hit for every probe.

Figure 5(b) shows the results of a FLUSH +RELOAD attack on RSA. In the absence of stealth-mode translation, the attacker can almost always detect when a *multiply* function has been invoked by measuring hits and misses (access time around 400 cycles versus access time around 1600 cycles) to the corresponding *reloaded* cache line. However, when stealth-mode translation is in effect, the attacker-perceived instruction access pattern is completely obfuscated resulting in a perceived I-cache hit at the end of every probe interval. The PRIME +PROBE attack on RSA (not shown) is also defeated, recording a miss on the attacker end after every probe interval.

Performance. The potential performance overheads of CSD include micro-op expansion (sending more micro-ops through the pipeline) and related side effects (micro-op queue pressure, micro-op cache pressure) and possible cache

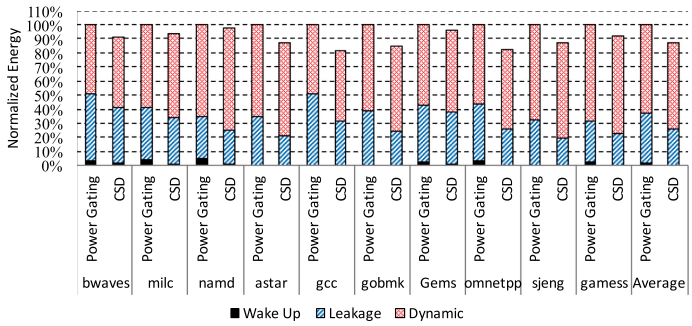

Figure 7. Total energy consumption of CSD's devectorizing mode normalized to that of power-gating.

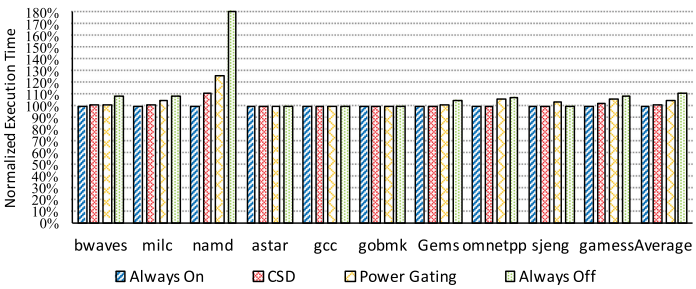

Figure 8. Execution time for different power gating policies, normalized to always on policy.

effects due to increased cache pressure from decoy loads.

Figure 6 compares the execution time of our pipeline without any optimization (*NoOpt*) and with both micro-op cache and micro-op fusion enabled (*Opt*). In these results, we see performance loss consistently below 10% and averaging 5.6% when secure mode is enabled.

To break down the performance overhead of CSD, we further examine both the impact of micro-op expansion and cache pressure. In these experiments, CSD causes a micro-op expansion of 8.0% on average. Further, we see a very slight loss in effectiveness of the micro-op cache (from 43% hit rate to 42%). On the other hand, we see almost no negative cache effects from the increased number of loads—misses per instruction remain nearly constant, meaning that overall hit rate has increased due to the prefetching effect of constantly bringing in the entire table. More detailed performance results can be found[2].

In these experiments, the watchdog timer is set to 1000 cycles, so the decoy loads are deployed at the first decoded tainted load or branch encountered, then decoding returns to normal mode until the timer fires again.

Selective Devectorization

Figure 7 shows the breakdown of energy for regular decoding with (1) conventional power gating and (2) our devectorizing mode using CSD. Energy numbers are normalized to the total energy of conventional power gating. On average, dynamic devectorization results in a 12.9% decrease in total energy consumption.

Figure 8 compares the execution time of three different VPU power-gating polices:1) Always execute on the vector units (*Always On*); 2) CSD which scalarizes the instructions in certain intervals; 3) Conventional power gating (*Power Gating*); and 4) Always translate instructions to scalar micro-ops to execute on scalar units (*Always Off*). The always-devectorize policy and power gating both incur considerable performance overheads (12% and 5%, on average respectively) while CSD reduces this to only 1.6% overhead. Thus, it keeps performance close to full vectorization even though the vector units are turned off much of the time. The only exception is *namd*, where CSD is much faster than full devectorization and conventional power gating, but still incurs a high cost.

More detailed results[2] show that for several applications with low (but not nonexistent) levels of vector activity, we are able to keep the vector units off virtually all the time, and over all benchmarks the vector units are powered down 70% of the time. Some applications that we observe to execute a significant number of vector instructions while in gated mode (e.g., *bwaves* and *milc*) still see little performance overhead due to the ability to execute the vector operations in scalar mode while the unit powers up. However, with *namd*, this happens too often, and the cost is higher.

Overall, for CSD-enabled selective devectorization, we find that we are able to power gate the vector unit for longer, unbroken periods, resulting in good energy savings with a small performance cost.

CONCLUSION

CSD enables the decoder to dynamically alter the instruction stream. This allows the system to change the functionality of the software without programmer or compiler intervention. This mechanism can enable any number of security,

performance, energy, or code analysis optimizations. In this paper, we demonstrate first a dynamic cache obfuscation technique, with minimal performance costs, that eliminates data-dependent access to code or data from being observed in the cache. Additionally, we demonstrate dynamic devectorization, allowing the vector units to power down in the face of low utilization.

ACKNOWLEDGMENTS

This work was supported in part by NSF under Grant CNS-1652925, in part by NSF/Intel Foundational Microarchitecture Research Grant CCF-1823444, and in part by DARPA under Agreement HR0011-18-C-0020.

REFERENCES

1. P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," *arXiv*, 1801.01203, 2018.
2. M. Taram, A. Venkat, and D. M. Tullsen, "Mobilizing the micro-ops: Exploiting context sensitive decoding for security and energy efficiency," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 624–637.
3. M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2019, pp. 395–410.
4. M. L. Corliss, E. C. Lewis, and A. Roth, "Dise: A programmable macro engine for customizing applications," in *Proc. 30th Annu. Int. Symp. Comput. Archit*, 2003, pp. 362–373.
5. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, Intel Corp., Aug. 2011.
6. O. Aciiçmez, "Yet another microarchitectural attack: Exploiting i-cache," in *Proc. 14th ACM Workshop Comput. Secur. Archit.*, 2007, pp. 11–18.
7. Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack." in *Proc. 23rd USENIX Security Symp.*, 2014, pp. 719–732.
8. D. J. Bernstein, "Cache-timing attacks on AES," Tech. Rep., 2005, available at http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
9. M. A. Laurenzano, Y. Zhang, J. Chen, L. Tang, and J. Mars, "Powerchop: Identifying and managing non-critical units in hybrid processor architectures," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 140–152.
10. R. Kumar, A. Martínez, and A. González, "Efficient power gating of simd accelerators through dynamic selective devectorization in an hw/sw codesigned environment," *ACM Trans. Archit. Code Optim.*, pp. 25:1–25:23, art no. 25, 2014.

Mohammadkazem Taram is currently a PhD student in the Department of Computer Science and Engineering , University of California San Diego. He is interested in computer architecture, compilers, and security. He has an MS in computer engineering from Sharif University of Technology. Contact him at mtaram@cs.ucsd.edu.

Ashish Venkat is an assistant professor in the Department of Computer Science, University of Virginia. His research interests are in computer architecture and compilers, especially in instruction set design, processor microarchitecture, binary translation, code generation, and their intersection with computer security and machine learning. He has a PhD from the University of California San Diego. His dissertation research has been successfully ported and transferred to the Cloud Platforms division of the IBM Haifa Research Lab. Contact him at venkat@virginia.edu.

Dean Tullsen is professor and chair of the Department of Computer Science and Engineering, University of California. He works in the areas of computer architecture, compilers, and security. He has been awarded the ISCA Influential Paper Award twice. He is a Fellow of the IEEE, a Fellow of the ACM, and a past chair of the IEEE Technical Committee on Computer Architecture. Contact him at tullsen@cs.ucsd.edu.