Mobilizing the Micro-Ops: Exploiting Context Sensitive Decoding for Security and Energy Efficiency

Mohammadkazem Taram   Ashish Venkat   Dean M. Tullsen
University of California, San Diego
{*mtaram* | *asvenkat* | *tullsen*}@*cs.ucsd.edu*

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

This architecture, first by allowing execution to transition between different translation modes rapidly, defends against a variety of attacks, completely obfuscating code-dependent cache access, only sacrificing 5% in steady-state performance – orders of magnitude less than prior art. By selectively disabling the vector units without disabling vector arithmetic, context-sensitive decoding reduces energy by 12.9% with minimal loss in performance. Both optimizations work with no significant changes to the pipeline or the external ISA.

***Keywords*-Microcode; Side Channel; Security; Power Gating;**

I. Introduction

The post-Dennard scaling era has witnessed an upsurge in the adoption of specialized processing elements to improve the execution efficiency of domain-specific workloads. While general-purpose processors continue to gradually add domain-specific instructions every CPU generation, the technical challenges and market risks associated with legacy software have significantly limited innovation in the ISA design space. This work 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.

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 [1], [2]. These architectures enjoy the dual benefits of a versatile backward-compatible CISC front-end and a simple cost-effective RISC back-end. Moreover, the additional level of indirection enables seamless optimization of the internal micro-op ISA, under the covers, without any change to the programmer interface. 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.

In this paper, we unlock 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. This presents operating systems, runtime systems, and antivirus programs with the unique opportunity of triggering different custom translation modes, at microsecond or finer granularity, by simply configuring a set of model-specific registers (MSRs). 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.

By leveraging existing native-to-microcode translation functionality in the decoder and exploiting an already well-established microcode update procedure outlined by Intel [1], we further empower runtime systems and virtual machines (that operate at a certain privilege level) to push custom translation updates written in native x86 code into the processor. At the decode stage of the pipeline, the CSD framework intercepts such custom microcode updates, *auto-translates* and optimizes them into a compact set of micro-ops, and pushes them into the microcode engine. These custom updates could potentially enable instrumentation for profiling and performance monitoring, profile-guided optimizations, and API-hooks for security updates, among other applications.

The CSD framework we describe allows custom translation modes to be triggered by hotspot detection [3], unit-criticality predictors [4], thread-criticality predictors [5], protection-domain crossings [6], interception of a tainted input [7], [8], [9], [10], a watchdog timer event, changes in power or energy availability, or thermal events – all with no significant changes to the pipeline or the ISA. In fact, a major contribution of this work is a set of microarchitectural techniques that enable the seamless integration of the context-sensitive decoding framework into Intel's legacy decode pipeline and micro-op cache design.

Due to its low performance overhead and non-intrusive nature, context-sensitive decoding has potential applications in areas such as malware detection and prevention [11], [12], [13], dynamic information flow tracking (DIFT) [7], [14], [15], [10], runtime profiling and performance programming [16], [17], on-demand type-safety [18], [19], program verification and debugging [20], [21], [22], and runtime phase tracking and code specialization [23], [24]. This paper showcases two diverse applications of context-sensitive decoding – an obfuscation-based security defense against cache-based side channel attacks, and criticality-aware power gating to improve energy efficiency.

Side-channel attacks have been used to leak secret information by exploiting the micro-architectural and physical characteristics of a cryptosystem. Many types of side-channel attacks have been described in the literature to subvert prominent cryptographic algorithms such as RSA, DES, and AES. These attacks hinge on a spy program running side-by-side with a victim that leaks timing and other execution characteristics via shared micro-architectural structures.

By leveraging custom translation modes offered by context-sensitive decoding, we provide a low-cost, high-performance, and reconfigurable alternative to existing side-channel mitigations [25], [26], [27]. Owing to its unfettered access to on-chip microarchitectural structures and an array of hardware control signals, context-sensitive decoding allows us to inject *decoy micro-ops* into execution that give the attacker an illusion of a modified architectural state, by obfuscating micro-architectural characteristics alone. These decoy micro-ops are unreadable from both user and kernel modes as they exist within the processor outside any addressable memory. As a result, they remain invulnerable to spyware, rootkits, and other rogue programs, even if they are able to execute with the highest privileges. This paper shows that by causing micro-architectural perturbations at the decoder level, we can be more performance-efficient than a software-based obfuscation technique and less intrusive than a system that causes anomalies at the gate level.

Aside from security, we also showcase the potential of CSD in efficiently emulating infrequently used feature sets on alternative functional units. This enables aggressive power gating, even for units that are infrequently but regularly used. In this case, we scalarize vector instructions via micro-op translation onto the scalar units, enabling the system to make more global decisions about when to turn on the vector units, rather than always responding to instruction demand. In summary, the framework we describe in this paper offers the following unique capabilities:

- Fine-grained dynamic instruction stream customization of legacy binaries without recompilation, and without the full overhead of binary translation.
- Seamless integration into a state-of-the-art Intel processor with no significant changes to the pipeline.
- A flexible *auto-translated* microcode update procedure that allows runtime systems to inject custom translation modes into the microcode engine.
- A firmware-based security defense that completely thwarts instruction and data cache-based side channel attacks on the RSA and AES cryptographic algorithms, while significantly outperforming state-of-the-art software-only solutions.
- An energy-optimization mechanism that scalarizes vector instructions in order to power gate vector units during phases of minimal vector activity to save an average of 12.9% in overall energy.

II. Background and Related Work

Translated Instruction Sets. Modern ISAs such as x86 and ARM typically translate complex native instructions into simpler internal micro-ops [1], [2]. While this was originally intended to simplify CPU design and allow complex long latency instructions to be pipelineable, it has been instrumental in enabling several ISA and micro-architectural optimizations [28], [29], [30], [31], [32] that improve the front-end throughput and overall instruction-level parallelism [33]. Speculative decode [28] exploits the macro-op to micro-op translation in order to enable dynamic optimizations such as memory reference combining and silent store squashing. While similar in nature with respect to decode-time instrumentation, CSD offers broader functionalities such as the ability to be programmed via microcode updates, on-demand customization at an extremely fine granularity, as well as seamless integration into the modern x86 front end.

Multi-ISA Architectures. Many modern decoders are equipped with multiple decode units to translate instructions from different feature sets, ISA extensions, and sometimes completely different ISAs. For example, ARM supports three major instruction sets (A32, T32, and A64) and several other feature sets such as Jazelle and NEON. It also allows developers and compilers to take advantage of the ability to switch between the different instruction sets at exception boundaries (A64 to A32) or by simply executing a branch and exchange instruction (A32 to T32). Furthermore, the multi-ISA heterogeneous chip multiprocessor architectures [34], [35], [36], [37] allow applications to migrate back and forth between different ISAs at basic block boundaries. While this work offers similar capabilities in terms of seamlessly switching execution between different custom translations, it does so at a much finer granularity, requires no re-compilation, and no significant changes to the architecture.

Binary Translators and Code-Morphing Machines. In the software world, binary translators have long been used to port/emulate legacy binaries on new architectures [38]. Furthermore, managed runtimes and browsers employ dynamic binary translation to perform profile-guided optimization [39] of hot code regions, program sheperding, and JIT hardening [40]. On the hardware front, several binary-translation-driven processor designs have been proposed. These are typically equipped with a code-morphing software binary translation layer that feeds translated instructions into the processor's decoder. IBM's DAISY [41], Transmeta's Crusoe and Efficieon [42], and Nvidia's Denver processors [43] have sparked further innovation in this space. Clark, et al. [44] describe a hybrid approach to instruction set customization that involves statically identifying code regions to offload and dynamically replacing jumps to such regions by complex custom instructions that trigger an accelerator.

The most related work to this research is DISE (Dynamic Instruction Stream Editing) [45], [46], [47], a macro-engine that exposes the API, allowing programmers to dynamically reconfigure a stream of instructions in order to perform bounds-checking, debugging, and prefetching. However, this work differs in many important ways. First, they require complex patten-matching and user-defined production rules to be integrated into their decoder framework, whereas context-sensitive decoding can be triggered by mere reconfiguration of a set of model-specific registers or even a pipeline event, or a thermal or energy event. Second, while this work is easily integrated, exploiting existing features of modern processors, DISE adds significant new complexity to the pipeline. Third, this work fully explores performance, power, and area implications of incorporating the decoding framework into existing modern designs, including those that sport a wide variety of micro-op optimizations [1]. Finally, our research builds on works like DISE by introducing new applications – better protection against several new attack models that have gotten more sophisticated over the years, and energy-efficient management of vector computation.

Side-channel attacks. Side-channel attacks typically steal secret information from cryptosystems and other sensitive data from a co-located user on the cloud [48], [49]. Numerous spy programs have demonstrated the full/partial reconstruction of a victim's execution behavior by observing its instruction/data cache access patterns [50], [51], [52], [53], [54], branch access patterns [55], differential power consumption characteristics [56], electromagnetic radiation [57], acoustics [58], and fault behavior [59].

Several cache-based side channel attacks have been proposed in the literature [60], [53], [61]. Prominent ones include PRIME+PROBE and FLUSH+RELOAD attacks that can be performed on both a shared private data/instruction cache or on last-level caches. Notable mitigations to these attacks include secure cache partitioning [25], compiler-based obfuscation [26], [27], and run-time software diversity [62]. In this paper, we leverage context-sensitive decoding to provide stealth-mode custom translations, as a novel security defense that mitigates such side channel attacks. Context-sensitive decoding-based security enhancement has no software performance cost when not in use, minimal software overhead when used, no additional vulnerability, and very minimal hardware/power cost.

Unit-level Power Management. Many power management techniques have been proposed in prior work ranging from unit-level [63], [64], [65] to coarse-grained core-level power management [66], [67], in both cases powering off idle blocks to reduce overall static leakage. *Vector processing units* (VPUs) are promising candidates for power gating since they're typically not in use during most scalar phases, and yet account for a significant portion of the core's peak power. However, phases of intermittent vector activity create small idle intervals that are below the break-even time needed to compensate the power gating overhead.

Dynamic devectorization [68], [69] achieve significant energy savings by using a translation optimization layer to profile and devectorize non-critical vector instructions while the VPUs are power-gated. Similarly, PowerChop [4] proposes a binary translation-driven approach that uses a unit-criticality predictor to assist power-gating of multiple units in the processor (including VPUs). While binary translation can be an effective tool, it is not ideal for adaptive energy optimizations – in many scenarios we can hide the considerable startup cost (in performance and energy) of binary translation; however, when we trigger a new optimization due to an energy event or emergency, we bear the entire brunt of the startup cost at the worst possible time.

III. Context-Sensitive Decoding

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

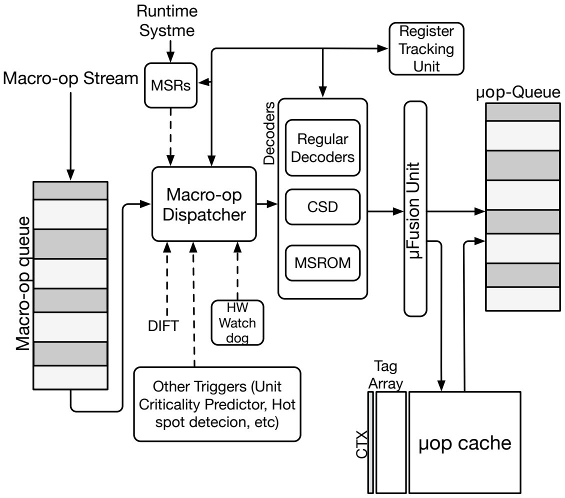


Figure 1: Intel Front End with CSD Support.

A. Overview of the x86 Front End

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

The legacy decode pipeline includes an instruction-length decoder that feeds from a 16-byte fetch buffer and decodes the variable-length x86 instruction byte-by-byte. The decoded instructions are inserted into an 18-entry macro-op queue, 6 macro-ops at a time. These macro-ops then feed 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 a microcode ROM.

The micro-ops translated by the legacy decode pipeline are cached in an 8-way set associative micro-op cache that can hold up to 1536 micro-ops. When the micro-op cache is active, the legacy decode pipeline is disabled to conserve power. 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.

B. Integration with the x86 Front End

This section describes techniques to integrate our architecture into the x86 front end with the legacy decode pipeline and micro-op cache designs, and further study its synergy/interference with existing front-end optimizations such as micro-op fusion. Figure 1 highlights necessary hardware components required to enable context-sensitive decoding in the x86 front-end.

Integration with the Legacy Decode Pipeline. To enable context-sensitive decoding in the x86 front-end, we provision 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 microcode ROM. CSD does not require that micro-ops of an instruction be committed atomically. This is consistent with the current implementation of Intel processors which only commit 6 fused micro-ops per cycle while they allow as many as 260 micro-ops per one instruction (e.g., FBSTP) [70].

Furthermore, when context-sensitive decoding is turned on, we update the macro-op dispatch logic to redirect macro-ops that require custom translation to the custom decoder. In our initial implementation, this logic can be triggered in three different scenarios. First, software programs such as the operating system can trigger this logic by configuring a set of model-specific registers (MSRs) [1]. We leverage the already existing register-tracking optimization in the decoder to track updates to the MSRs and consequently trigger context-sensitive decoding. Second, we allow a translation context switch to be triggered by hardware events such as the interception of a tainted input by information-flow tracking or a power-gating decision by the unit criticality predictor. Finally, we allow a hardware watchdog timer to periodically trigger a translation mode switch.

Interactions with the Micro-Op Cache. The micro-op cache is an important performance and energy optimization that allows certain hot code regions to be completely serviced from the micro-op cache. The Intel Optimization manual recommends software to be carefully optimized since frequent switching between the micro-op cache and the legacy decode pipeline could cause more performance degradation than running without the micro-op cache [71]. This particularly conflicts with one of the major goals of context-sensitive decoding – the ability to frequently switch translation context at a low performance overhead.

Flushing the micro-op cache every translation mode switch could have a major performance impact. We instead choose to extend the tag bits of the micro-op cache with an additional set of context bits (one bit per custom translation mode) that associate a particular micro-op way with the decoder that translated it. While this could potentially create artificial conflict misses, it allows us to improve the micro-op cache utilization by co-locating micro-op translations from different custom decoders.

Finally, customization could involve injecting multiple micro-ops at a time. This not only clutters the execution stream, but could pollute the micro-op cache. The x86 micro-op cache design has a check that does not allow 32-byte code regions to occupy more than 3 ways (amounting to 18 micro-ops) in the micro-op cache. This is because, unlike a regular cache, the micro-op cache simply allows the front-end engine to stream instructions from it, to avoid expensive indexing and tag comparison. Furthermore, it does not allow instructions longer than six fused micro-ops to be cached. Although we can imagine several options that would allow the architecture to remove that constraint, to be conservative we assume that it still holds in this paper, which does impact many of our translated micro-op sequences.

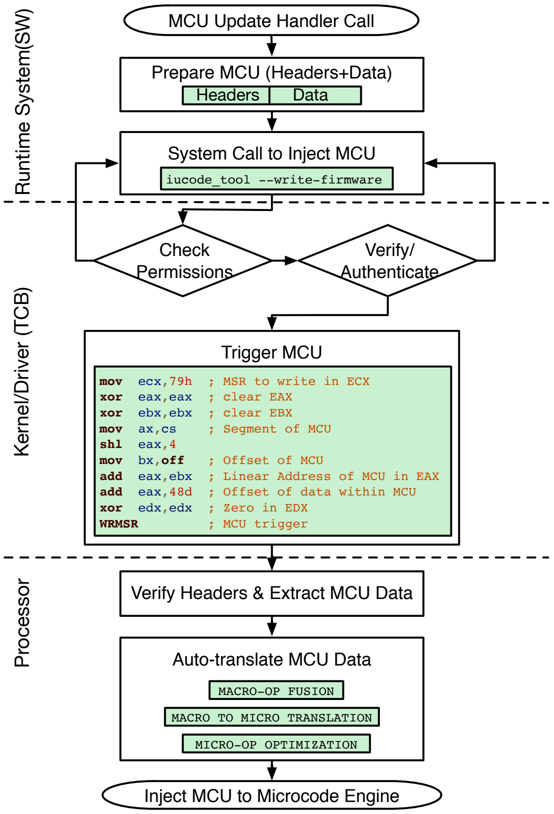
C. MicroCode Update and Auto-Translation

CSD exploits the already existing *Microcode update* (MCU) procedure of Intel processors [1] to empower the runtime system with the ability to inject custom translations into the processor's microcode engine, with the API provided to the runtime being the entire x86 instruction set. The CSD framework further *auto-translates* such microcode updates by exploiting Intel's existing front-end translation and optimization infrastructure. While this offers significant flexibility to software agents such as the OS and the runtime system, the chip designer exerts more control over the microcode engine, potentially allowing custom translations that include non user-visible features such as a micro-op that can change the state of the branch predictor or the hardware return address stack. We also note that custom translations injected via microcode updates should not alter architectural register and memory state, unless explicitly specified in the MCU header.

Figure 2 shows the MCU procedure in more detail. Since microcode update is performed via a privileged instruction or system call, only trusted entities [72], [73], [74] within the OS/runtime system should have the ability to successfully inject microcode updates into the processor. The microcode update system call invokes Intel's microcode driver [75] that performs sanity and integrity checks, and further invokes the processor's microcode update feature via an MSR update[1]. The MCU itself is provisioned with a descriptive header prepended by data containing custom translations injected by the runtime. When the header contains a reserved field that indicates context-sensitive decoding, the microcode update is assumed to contain only native x86 instructions, and is further marked for *auto-translation*. On the processor end, the MCU header is again verified for sanity and integrity, before extracting the data part. In the event that the MCU is marked for *auto-translation*, the native instructions in the data part of the MCU are further translated into internal micro-ops by leveraging the existing translation capabilities in the decoder. The translated micro-ops are further optimized into more compact micro-ops using existing front-end optimizations such as macro/micro-op fusion, adhering to certain performance guidelines described below. We further note that virtually all of the building blocks we use to provide this feature are already well-established mechanisms that appear in mainstream Debian Linux kernel releases[75].

D. Performance Guidelines and Optimizations

The micro-op expansion due to customization could potentially have a negative impact on overall performance if the custom translations are not optimized before they are injected into the dynamic execution stream. In this paper, we take advantage of several existing optimizations in the micro-op engine in order to eliminate bottlenecks at the front-end and potentially, throughout the pipeline.



```
mov  ecx,79h   ; MSR to write in ECX
xor  eax,eax   ; clear EAX
xor  ebx,ebx   ; clear EBX
mov  ax,cs     ; Segment of MCU
shl  eax,4
mov  bx,off    ; Offset of MCU
add  eax,ebx   ; Linear Address of MCU in EAX
add  eax,48d   ; Offset of data within MCU
xor  edx,edx   ; Zero in EDX
WRMSR          ; MCU trigger
```

Figure 2: Auto-translation Procedure

Micro-op fusion. To take advantage of the micro-op fusion optimization, we use *load+op* and *load+br* combinations as much as we can in our custom micro-op sequences. By doing so, we gain 1.6% in performance and eliminate bottlenecks in the front end stages, and the micro-op cache.

Micro-loop Specialization. We attempt to create short and tight loops that benefit from the improved micro-op cache utilization, and take advantage of the loop cache when present. The Intel optimization manual [71] recommends loop fission [76] in case of longer loops. If the custom decoder decides to employ loop fission on micro-loops, we recommend that these loops don't occur too close together in the pipeline as they can potentially knock native instructions in a 32-byte region out of the micro-op cache, causing severe degradation in performance.

Register Tracking. Finally, customization may potentially involve reusing micro-registers (e.g., as a loop induction variable). By extending the stack pointer tracker to perform full register tracking, more compact instances of custom micro-op translations could be created.

E. Potential Applications

Later sections of this paper focus on particular applications of context-sensitive decoding 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 (subject of future work) abound.

Programming Languages – To reduce costly time spent on finding and fixing bugs, developers are increasingly encouraged to employ software practices that ensure software fault isolation, type safety, and formal verification [18], [19], [77]. While most type checkers and proof assistants rely on static verification, the dynamic nature of JavaScript and other JITed code has made it increasingly hard to statically infer and reason about types. Typed assembly languages [78] and Google's Portable Native Client [18] ensure deep sandboxing and type safety of inherently native and/or JITed code, but at a prohibitively high performance cost for many workloads. With CSD, we can read metadata at the time of a load to identify type, and pass flags between instructions to track computation on registers, thereby increasing the coverage for sensitive code regions where static verification is insufficient.

Debugging – Aside from type checking, breakpoints and watchpoints are indispensible tools that software developers use to find and fix evasive memory errors. Modern ISAs with hardware debugging support reserve a small number of monitor/debug registers to encode breakpoint/watchpoint rules [20], [22]. However, most debuggers and runtime analyses typically run out of debug registers and resort to software breakpoints and watchpoints which are extremely inefficient [21]. In translated ISAs, a context-sensitive decoder can microsequence a performance-efficient watchpoint implementation since it has direct access to microarchitectural structures such as the address translation unit.

Performance Counters – Modern processors implement a variety of performance counters, but with several limitations. Only a few can be used at once, and the actual counters typically change from generation to generation, often dependent on where the designer had room for a counter and where they did not. However, we can add many counters in the decoder, with no limit to the number of counters active at once, and providing compatibility across generations despite different layouts and space availability.

Profiling – Modern systems typically rely on instrumentation to profile code. However, intrumentation alters the code, potentially resulting in *heisenbugs*. That is, instruction cache, data cache, and even memory interference behavior is altered by the instrumentation. With CSD, we can add profiling with no change whatsoever to code layout or data layout.

IV. Case Study I:Side-Channel Defense

In this section, we demonstrate the security potential of context-sensitive decoding. We first lay out our assumptions and threat model, then describe the stealth-mode translation feature of context-sensitive decoding. Finally, we leverage this feature to secure commercial implementations of RSA and AES against the exploitation of the two major data and instruction cache side channel attacks.

A. Assumptions and Threat Model

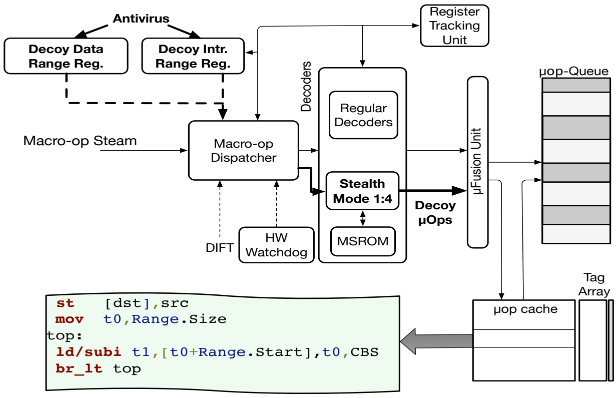
Trusted Computing Base. We assume that the micro-op engine – which includes both the legacy decode pipeline and the micro-op cache – is tamper-proof and is a part of the Trusted Computing Base (TCB) [79], [73]. We also further extend the TCB to include all hardware or software mechanisms that can potentially trigger context-sensitive decoding. These include register tracking, dynamic information flow tracking [7], hardware watchdog timers, and anti-virus-driven stealth mode configuration (e.g., an Intel/McAfee security solution). Moreover, we assume that such hardware security mechanisms, including any microcode that enables security, are formally verified [80]. Note that we only assume that the trigger/injection mechanisms (e.g., watchdog timer, microcode update kernel module, context-sensitive decoder) are a part of the TCB, but the instruction stream itself need not be. Finally, since the API exposed to software only consists of macro-ops, we continue to assume that the translated micro-ops in the micro-op cache (both native and custom-translated) can neither be read by software nor be probed via hardware side-channels.

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 hardware performance counters. This allows them to make inferences about the software algorithm being run by the victim, by observing the micro-architectural (cache) characteristics alone.

B. 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 that could potentially 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. The goal of the stealth-mode translation is to provide an illusion of a modified architectural state by obfuscating the micro-architectural characteristics alone. In this specific implementation, we obfuscate a victim's control path and/or access to sensitive data structures in an attacker-oblivious way. 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. While we intend stealth-mode to be a security feature to be deployed by the chip manufacturer, trusted software entities [73] with the right privileges can achieve similar effects by leveraging the *auto-translated* microcode update feature.

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) [1] in x86 that allow system software to control cache policies for specific address ranges (e.g., write-back vs write-through). The decoy address range registers, on the other hand, allow anti-virus 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 the 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.



```
 st   [dst],src
 mov  t0,Range.Size
top:
 ld/subi t1,[t0+Range.Start],t0,CBS
 br_lt top
```

Figure 3: Context-Sensitive Decoding with stealth mode

```
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

CSD injects *decoy micro-ops* into all instructions that use memory operands and/or attempt control transfer (i.e., all instructions that get translated to load/store/branch micro-ops) 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 effectively obfuscates the architectural state by loading all sensitive cache blocks whose addresses have been specified by a software agent (e.g., antivirus) in the MSRRs.

We implement 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 information-flow tracking in hardware where the taint-checking is performed dynamically. 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 once all the address ranges in the context-sensitive decoder's internal copy of the *decoy address-range* MSRs have been emptied out (all blocks specified by the range registers are loaded) by the *decoy micro-ops*. However, before turning itself off, the context-sensitive decoder 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.

In the next section, we show how stealth-mode translation can be used to secure the instruction and data caches by defeating both the PRIME+PROBE and FLUSH+RELOAD variants of cache-based side-channel attacks. We demonstrate these on two specific instances of known vulnerabilities, but our scheme extends to any application where the secure-data-dependent code and data access ranges can be identified.

C. Securing the Instruction Cache

Instruction cache-based attacks have been able to successfully break prominent cryptographic algorithms, such as RSA, by being able to capture their key-dependent instruction access patterns. In this section, we show how RSA is inherently vulnerable to the I-cache attack and describe how stealth-mode translation can defeat the attack.

The RSA Cryptographic Algorithm. RSA is a popular public-key cryptographic algorithm that uses modular exponentiation for encrypting messages. Most commercial implementations of the RSA algorithm, including PGP and the open-source version GnuPG, use the *square-and-multiply* algorithm that considerably speeds up modular exponentiation. The algorithm iterates over the binary representation of the exponent in order to selectively perform exponentiation using three major sub-operations: *square*, *multiply*, and *reduce*. While *square* and *reduce* are performed in each loop iteration, *multiply* is invoked only when the exponent bit is 1, which invariably entails a key-dependent branch.

I-Cache Side-Channel Attack on RSA. The sub-operations of the *square-and-multiply* algorithm are implemented as fairly large functions that span multiple cache blocks. This implies that the I-cache access patterns captured by a co-located spy process could potentially reveal several bits of the exponent. A PRIME+PROBE attack that exploits this side channel [81] 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 on the other hand, leverages shared libraries and/or page de-duplication in order to flush the cache line that corresponds to the *multiply* routine, and further reloads it after a carefully chosen probe interval – a quick access indicates the victim has accessed the code and brought it into a shared cache.

Effect of Stealth-Mode Translation. As a defense mechanism against I-cache based side-channel attacks, we use stealth-mode translation to periodically (potentially, at every probe interval) inject decoy instruction-load micro-ops into the pipeline. Stealth-mode can seamlessly and instantaneously 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.

D. Securing the Data Cache

As with instruction cache-based attacks, attackers have also exploited data-cache side channels to infer secret information by observing a victim's key-dependent data access patterns. In this section, we describe a well-known data cache-based side channel in OpenSSL's implementation of the AES algorithm and discuss the potential of stealth-mode translation to thwart these attacks.

The AES Cryptographic Algorithm. AES Algorithm is 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 in order to speed up the substitution-permutation rounds, which then consist of several simple table lookup and *xor* operations. The index computation for the T-table lookup involves an *xor* operation between the key bits and the plaintext bits, thereby entailing a key-dependent load.

D-Cache Side-Channel Attack on AES. The OpenSSL implementation of AES employs four 256-entry T-tables, which amounts to sixty-four 64-byte cache blocks in the data cache. A spy process that monitors the access patterns of these blocks during encryption can significantly reduce the possible key space, and potentially reconstruct the entire key, by using a large number of carefully chosen plaintext [51]. 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 de-duplication 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. Similar to the use of stealth-mode translation to defend against I-cache attacks, 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. Furthermore, by carefully choosing a watchdog timeout period to enable periodic *decoy load* injection, we can also defend against brute-force key extraction attacks [82].

E. Securing other Side-Channels.

Although the primary focus of this work is to defend against cache-based side-channel attacks, we note that the stealth-mode translation feature of CSD could be exploited

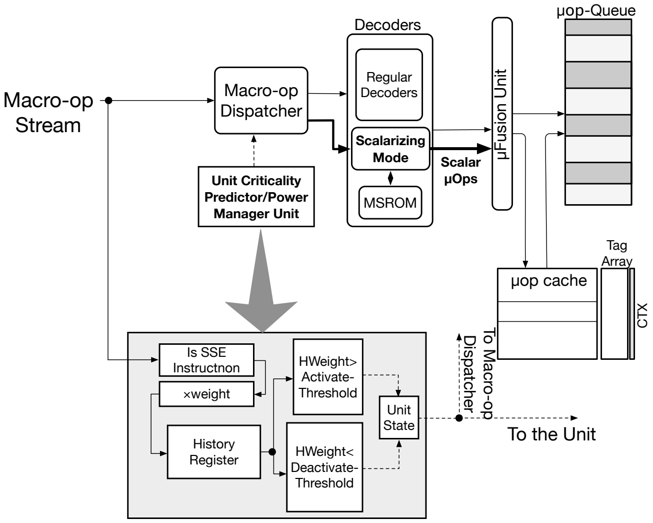


Figure 5: CSD with Selective Devectorization

in 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 non-deterministically microsequencing instructions that cause switching activity across different microarchitectural structures.

V. Case Study II: Unit-Level Power Gating

In this section we present another use case of context-sensitive decoding, selective devectorization for unit-level power-gating. While similar in functionality to software-based devectorization approaches proposed in prior work [4], [68], [69], 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 leakage power by disconnecting the supply voltage of an idle unit. 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 cost but also slows execution, typically stalling until the unit is activated. We leverage context-sensitive decoding to enable efficient and more fine-grained power-gating of vector units. CSD alleviates these shortcomings in two ways: 1) for infrequent vector instructions, it avoids powering on the unit by translating vector instruction to equivalent scalar micro-ops, and, 2) it hides the power-on delay by continuing the execution of instructions using scalar mode until the unit is ready.

Figure 5 shows our hardware support (beyond the decoder) for dynamic devectorization. We employ nothing more than a simple counter that tracks a window of instructions, counting up one for simple vector instructions and more than one for more complex vector instructions (higher micro-op count).

```
PADDB(xmm_t src,  xmm_t dst){
    xmm_t masked_src, masked_dst, temp_res;
    xmm_t Mask = 0x00000000000000ff;
    for(int8_t i = 0; i < 16; i++){
        masked_src = src & Mask;
        masked_dst = dst & Mask;
        temp_res   = masked_src + masked_dst;
        temp_res   = temp_res & Mask;
        dst         = dst | temp_res;
        Mask        = Mask << 8;
    }
}
```

(a) Pseudocode

```
PADDB(xmm_t src,  xmm_t dst){
    .DEF M1 0x00FF00FF00FF00FF
    .DEF M2 0xFF00FF00FF00FF00
    ;LOW 64-bits , MASK M1
    andi t1, src_l, $M1 ;Mask 1st Op
    andi t2, dst_l, $M1 ;Mask 2nd Op
    add  t3, t1, t2  ;Add Masked Op
    andi t3, t3, $M1 ;Mask output
    ;HIGH 64-bits, MASK M1
    andi t1, src_h, $M1
    andi t2, dst_h, $M1
    add  t4, t1, t2
    andi t4, t4, $M1
    ;LOW 64-bits , MASK M2
    andi t1, src_l, $M2
    andi t2, dst_l, $M2
    add  dst_l, t1, t2
    andi dst_l, dst_l, $M2
    or   dst_l, dst_l, t3
    ;HIGH 64-bits, MASK M2
    andi t1, src_h, $M2
    andi t2, dst_h, $M2
    add  dst_h, t1, t2
    andi dst_h, dst_h, $M2
    or   dst_h, dst_h, t4
}
```

(b) Optimized micro-ops

Figure 6: Devectorization of add byte instruction

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. It also includes a cycle counter to continue devectorization until the vector unit is fully powered.

When devectorization is enabled, the microcode engine translates the vector instructions to an equivalent set of scalar micro-ops. As an example, Figure 6a shows the pseudocode that devectorizes the SSE *PADDB* instruction which performs integer addition on packed bytes. This code is further compiled and optimized into a set of native x86 instructions by a runtime system which performs the actual microcode update. Figure 6b shows the equivalent auto-translated micro-op version of such an update. While it is possible to use a simpler translation with a loop, we find that it is more efficient to unroll the loop in this case. This is because, by employing suitable masks, the computation itself can be optimized in a way that allows us to just perform four adds and accumulate the results. While this optimization holds true for this particular example, the decision to use micro-loops and other optimizations purely depends upon the nature and purpose of the custom translation.

Power Modeling and Power Gating Overheads For powering a unit on and off, power gating uses a header transistor that connects or disconnects the power source of

Baseline Processor			
Frequency	3.3 GHz	I cache	32 KB, 8 way
Fetch width	4 fused uops	D cache	32 KB, 8 way
Issue width	6 unfused uops	ROB size	168 entries
INT/FP Regfile	160/144 regs	IQ	54 entries
LQ/SQ size	64/36 entries	Functional	Int ALU(6), Mult(1),
Branch Predictor	LTAGE	Units	FP ALU/Mult(2), SIMD(2)

Table I: Architecture detail for the baseline x86 core

the unit. A sleep signal is applied to the gate of the header device to control its operation. Switching the unit off and on comes at the cost of asserting and de-asserting the sleep signal plus switching on and off the header device. These costs are responsible for the non negligible timing and energy overhead of power gating. We use Equation 1 from a model proposed by Hu et. al. [83] to account for the energy overhead of power gating.

$$E_{Overhead} \approx 2W_H \frac{E^s_{cyc}}{\alpha} \tag{1}$$

Where W_H is the ratio of the area of the sleep transistor to the area of the unit and E^s_{cyc}/α is the switching energy of the unit for one cycle when switching factor $\alpha = 1$. We use a conservative value of 0.20 for W_H, as the literature uses an estimated range of 0.05 to 0.20 [4], [83], [84], [85], and for E^s_{cyc}/α we use McPAT estimates [86]. Power gating cycles should be made long enough to compensate for the $E_{Overhead}$. The break-even time is defined as the number of cycles a unit should stay in power-gated state so that the aggregate energy savings of power gating (E_{saved}) matches the energy of switching the unit on then off ($E_{Overhead}$). We model the leakage current of the header transistor itself, using McPAT. We use Laurenzano et. al.'s [4] estimate of 30 cycles for powering on the VPU.

VI. Methodology

This section details the evaluation methods we use for our two case study applications of context-sensitive decoding. Since most of the complexity of both is hidden by the decoder, most of the infrastructure, including simulation, is shared between the two. However, because the techniques have very different goals, we do evaluate them in different ways.

A. *Performance Evaluation*

We model the x86 pipeline using the gem5 [87] 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 [71]. Our baseline processor is based on the Intel Sandybridge microarchitecture [88], adapted to the instruction queue model of gem5. Table I lists the capabilities of our baseline processor in more detail.

To evaluate the performance of the stealth-mode translation technique we describe in this paper, we need an application that exhibits a cache-based side-channel vulnerability – due to the context-sensitive nature of this technique, we seldom deviate from native performance when not in use and incur no overhead on unaffected code. Therefore, we evaluate the performance of our technique 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 [89]: Blowfish and Rijndael cryptographic algorithms that are vulnerable to data cache based side channel attacks. We do not use PGP's IDEA algorithm and the SHA algorithm available in MiBench because we could not find any key-dependent loads or branches that would require our stealth-mode translation mechanism. Each of our included security applications can be run in two modes (encrypt and decrypt) that each perform different computations and therefore exhibit different performance characteristics, giving us 8 performance datapoints.

We use DIFT as a trigger mechanism that detects key-dependent loads and further enables stealth-mode translation. While DIFT can be implemented via CSD, we want to separate the micro-op expansion and related effects of stealth-mode (the primary topic of this paper) by leveraging a lightweight hardware implementation [7] which incurs an extra 4-cycle L2-tag access latency. Finally, to enable our antivirus-driven trigger mechanism, we set aside five scratchpad registers in the context-sensitive decoder that each contain the addresses of potentially tainted instructions.

To evaluate the selective devectorization mode, we use a wide range of high and low ILP applications from the SPEC CPU2006 suite. We model power using McPAT with a 32nm technology node [86]. Finally, we use SimPoint [90] and Pinplay [91] to select simulation regions.

B. *Security Evaluation*

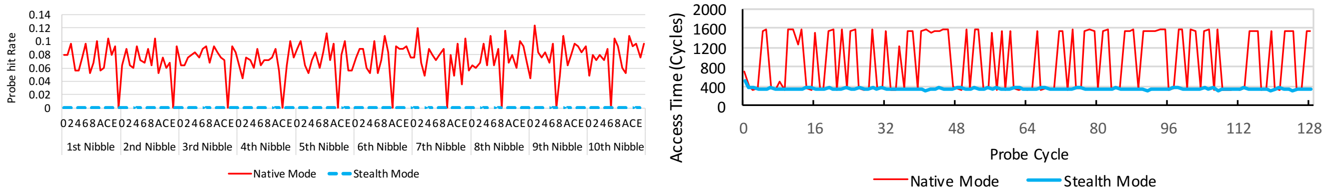
To evaluate the effectiveness of our security defense, we subject AES and RSA running on our architecture to the FLUSH+RELOAD variant of cache-based side-channel attacks, modeled after the AES T-Table attacks by Gruss, et. al. [60]. As further demonstration, we try the PRIME+PROBE attack on AES and RSA. Our attack models exploit the I-cache side-channel for RSA and the D-cache side-channel for AES, demonstrating defense against both side channels. Since we model our stealth-mode translation on a cycle-accurate simulator, we allow our attack models to benefit from precise counters and therefore do not require a calibration phase to set thresholds that distinguish between a hit and a miss, and subsequently determine probe intervals.

VII. Results

In this section, we evaluate each of our instantiations of CSD, starting first with our side-channel defense mechanism and following that with selective devectorization.

A. *Stealth Mode*

Security Evaluation. Figure 7a shows the results of a well-known PRIME+PROBE attack on AES [92]. 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 in a matter of 64000 attempts as shown in the figure. These bits are



(a) PRIME+PROBE on AES

(b) FLUSH+RELOAD on RSA

Figure 7: Effect of the cache attacks on AES and RSA with stealth-mode translation enabled.

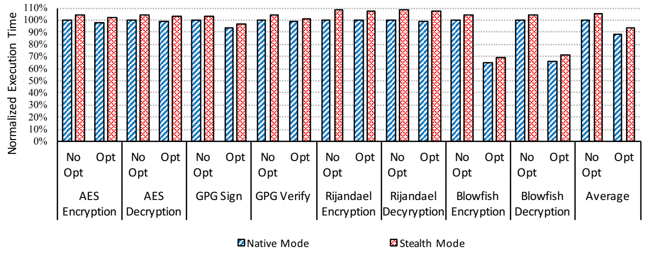


Figure 8: The execution time impact of context-sensitive decoding when implementing secure cache obfuscation normalized to insecure execution mode. The watchdog timer is set so that secure mode is re-entered every 500 microseconds.

sufficient enough to reverse-engineer the rest of key by well-established cryptanalysis techniques. However, when stealth-mode translation is enabled, the attacker-perceived data access patterns are completely obfuscated and always result in a hit for every probe, almost mimicking the behavior of a constant-time defense [51].

Figure 7b shows the results of a FLUSH+RELOAD attack on the RSA algorithm [53]. 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 (shown as dips and spikes in the figure) 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 and related side effects (micro-op queue pressure, micro-op cache pressure) and possible cache effects due to increased cache pressure from decoy loads. Careful construction of the secure-mode micro-op translation allow us to minimize many of the side effects of micro-op expansion.

Figure 8 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. This is to be compared with the current state of the art obfuscation techniques, which rely on the compiler (and consequently don't enjoy a hardware DIFT), that see performance expansion on the order of 20X [26] for applications with large memory footprints.

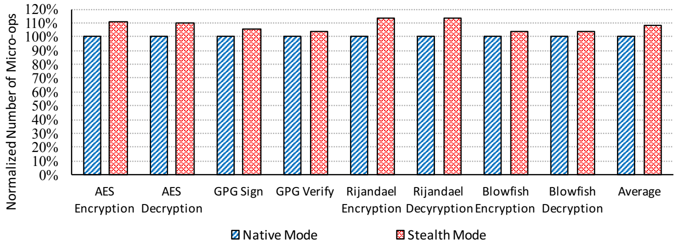


Figure 9: Micro-op expansion due to CSD.

To break down the performance overhead of the context-sensitive decoding, we first study micro-op expansion relative to unaltered execution. As shown in figure 9, context-sensitive decoding causes a micro-op expansion of 8.0% on average. Comparing these results together with Figure 8 seems to indicate that the primary cost of context-sensitive decoding is in fact the micro-op expansion. This is somewhat surprising, because we expect additional overheads from the higher incidence of loads and greater memory activity.

We investigate this further with several experiments. First, we measure performance in cycles/micro-op, and find that in fact this figure does not increase (and in some cases decreases), despite the fact that the percentage of load micro-ops has increased. Second, we see in Figure 10 that the number of cache misses per kilo instruction (MPKI) stays about the same on average. This indicates the vast majority of additional injected loads are hits. In yet another experiment where we discounted the cost of micro-op expansion, we saw an overall performance increase on average – this was due to a prefetching effect from the added micro-ops. Thus, the cache prefetching effect of the decoy loads is actually muting some of the performance cost of micro-op expansion.

Another negative side effect of context-sensitive decoding is on the micro-op cache. Because we introduce translations not allowed in the micro-op cache, or in other cases expand loops so they no longer fit, we do lose some of the effectiveness of that cache. However, that effect is small, especially when modeled with micro-op fusion enabled. Without fusion, the micro-op cache hit rate, on average, drops from 44% to 39% when we introduce CSD, but in the presence of micro-op fusion (which shortens some of the code sequences we expanded), it is much more stable dropping only from 43% to 42% with CSD-based stealth mode enabled.

In all above experiments, the watchdog timer is set to 1000 cycles (500 microseconds), so the decoy loads are deployed at the first decoded tainted load or branch encountered,

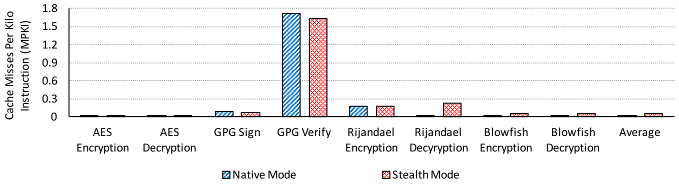


Figure 10: Number of cache misses without and with CSD.

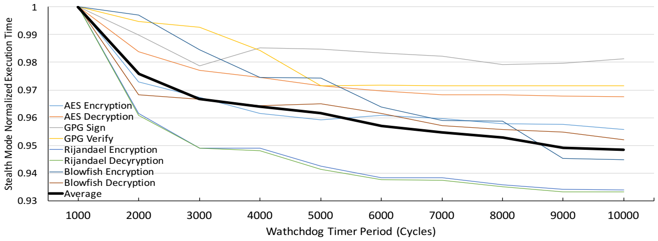


Figure 11: Effect of CSD timer on execution time.

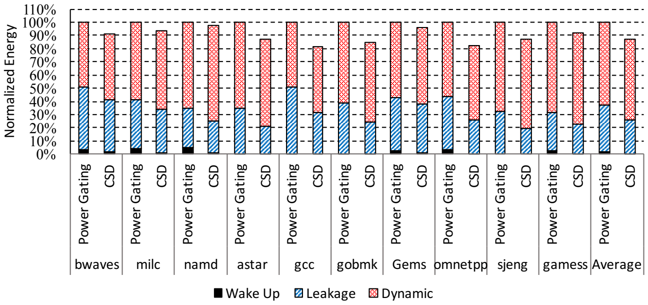


Figure 12: Total energy consumption of CSD's devectorizing mode normalized to that of power-gating

then decoding returns to normal mode until the timer fires again. While re-injecting decoy micro-ops every 1000 cycles provides almost perfect obfuscation against these cache-based side-channel attacks, based on system characteristic (e.g., cache miss/hit delays) and targeted attacks one can tune this parameter to increase the performance of the defense. Figure 11 shows the normalized execution time of our defense, sweeping the watchdog timer from 1000 to 10000 cycles. The decrease in execution time is caused by fewer extra micro ops and fewer micro-op cache conflicts.

Overall, we find that obfuscation of secure-data dependent microarchitecture state can be enabled with context-sensitive decoding with almost no performance cost, particularly in comparison with prior techniques.

B. *Selective Devectorization*

Figure 12 shows the breakdown of energy for regular decoding with (1) conventional power gating and (2) our devectorizing mode using context sensitive decoding. Energy numbers are normalized to the total energy of conventional power gating. On average, dynamic devectorization results in a 12.9% improvement of total energy consumption. This is despite the fact that several of our SPEC benchmarks make little use of vectorization.

Figure 13 compares the execution time of three different VPU power-gating polices: (1) Always execute on the vector units (*Always On*), (2) Context Sensitive Decoding which scalarizes the instructions based on the unit criticality predictor's decisions (*CSD*), (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 context-sensitive decoding reduces this to only 1.6% overhead. In general, 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 we are much faster than full devectorization and conventional power gating, but still incur a high cost.

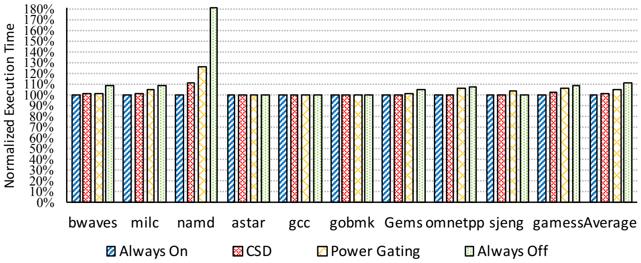


Figure 13: Execution time for different power gating policies, normalized to always on policy

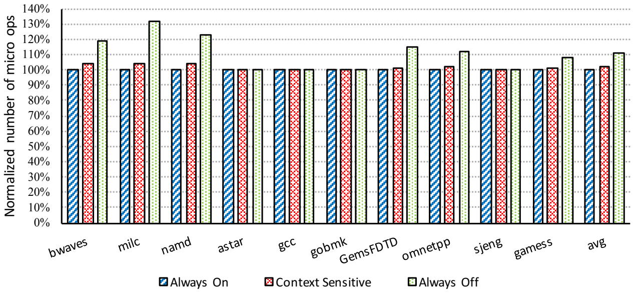


Figure 14: Micro-op expansion due to context sensitive decoding normalized to native mode

Figure 14 shows the number of dynamic micro-ops for the three VPU power-gating polices. Here we see that performance scales with micro-op expansion, the primary performance cost of CSD dynamic devectorization.

Figure 15 depicts the percentage of time that the VPU is kept power-gated for each benchmark. On average, context sensitive decoding can keep the VPU power-gated more than 70% of the execution time. For benchmarks *astar*, *gcc*, *gobmk*, and *sjeng* with low (but not nonexistent) vector activity, we are able to keep the vector unit turned off just about all the time, not having to turn it on for occasional outliers.

Figure 16 shows the breakdown of the SSE instructions in each benchmark. We categorize instructions into three categories: 1) instructions that are executed on the VPU (*Powered On*), 2) instructions that are devectorized and executed on scalar units because the VPU was in the process of powering on (*Powering On*), and 3) instructions that are executed on scalar units because the VPU was in power-gated state (*Power-Gated*). We find that *bwaves* and *milc* are frequently forced to execute scalar instructions while waiting for the vector unit to power on. They are still able to slightly come out ahead in energy, though, due to the

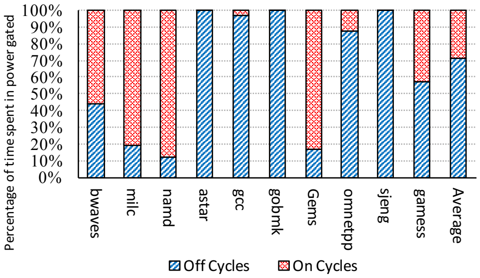


Figure 15: Percentage of time that CSD power gates VPUs.

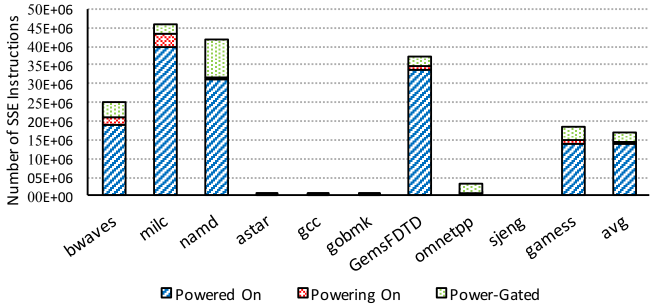


Figure 16: Breakdown of vector activity

performance advantage of not having to stall to wait for the VPU to turn on. *Namd* executes the largest number of vector instructions in gated mode, and is in fact gated 20% of the time despite having a large amount of vector activity. This implies that the threshold that performed overall was too aggressive for *namd*, and a more dynamic threshold or usage predictor would work better. Omnetpp, on the other hand, has a reasonable number of scalar operations but executes nearly all of them with the vector unit disabled, resulting in a significant gain in energy. And *gamess* is able to judiciously enable power gating, as it is gated nearly half the time, yet only about 20% of vector instructions are affected.

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.

VIII. Conclusion

The paper presents context-sensitive decoding, which enables the decoder to dynamically alter the decoding of programmer-visible ISA instructions. This allows the system to change the functionality of the software without programmer or compiler intervention. We use the technique for stealth mode translation, where the decoder injects instructions which completely obfuscate the effect of secure-data dependent branches and data accesses, defending against multiple variants of cache-based side channel attacks at just 5% performance degradation. This removes the microarchitectural footprint of the secure code from an attacker. Additionally we enable selective devectorization via CSD, saving 12.9% in energy while simultaneously achieving a speedup of 3.4% over conventional power gating.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful insights. This research was supported in part by NSF Grant CNS-1652925 and DARPA under the agreement number HR0011-18-C-0020.

References

[1] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, Intel Corporation, August 2011.

[2] *ARM Architecture Reference Manual*, ARM Limited.

[3] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars, "Protean code: Achieving near-free online code transformations for warehouse scale computers," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*. IEEE, 2014, pp. 558–570.

[4] M. A. Laurenzano, Y. Zhang, J. Chen, L. Tang, and J. Mars, "Powerchop: Identifying and managing non-critical units in hybrid processor architectures," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[5] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[6] E. Witchel, J. Cates, and K. Asanović, *Mondrian memory protection*. ACM, 2002.

[7] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[8] J. Newsome and D. X. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, 2005.

[9] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, 2008.

[10] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "Theoretical fundamentals of gate level information flow tracking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[11] S. Bhatkar and R. Sekar, "Data space randomization," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[12] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, and M. Castro, "Data randomization," Technical Report MSR-TR-2008-120, Microsoft Research, Tech. Rep., 2008.

[13] C. Rohlf and Y. Ivnitskiy, "Attacking clientside JIT compilers," *Black Hat, USA*, 2011.

[14] J. Oberg, S. Meiklejohn, T. Sherwood, and R. Kastner, "Leveraging gate-level properties to identify hardware timing channels," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[15] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "On the complexity of generating gate level information flow tracking logic," *IEEE Transactions on Information Forensics and Security*, 2012.

[16] T. H. Heil and J. E. Smith, "Concurrent garbage collection using hardware-assisted profiling," in *Proceedings of the 2Nd International Symposium on Memory Management*, 2000.

[17] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using hardware features for increased debugging transparency," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[18] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009.

[19] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "Rocksalt: better, faster, stronger sfi for the x86," in *ACM SIGPLAN Notices*, 2012.

[20] R. Wahbe, "Efficient data breakpoints," in *ACM SIGPLAN Notices*, 1992.

[21] J. L. Greathouse, Z. Ma, M. I. Frank, R. Peri, and T. Austin, "Demand-driven software race detection using hardware performance counters," in *ACM SIGARCH Computer Architecture News*, 2011.

[22] J. L. Greathouse, H. Xin, Y. Luo, and T. Austin, "A case for unlimited watchpoints," in *ACM SIGARCH Computer Architecture News*, 2012.

[23] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *ACM SIGARCH Computer Architecture News*, 2003.

[24] M. Gupta, V. Sridharan, D. Roberts, A. Prodromou, A. Venkat, D. Tullsen, and R. Gupta, "Reliability-aware data placement for heterogeneous memory architecture," in *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, 2018.

[25] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: Secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016.

[26] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution." in *USENIX Security Symposium*, 2015.

[27] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," *ACM SIGARCH Computer Architecture News*, 2015.

[28] I. Kim and M. H. Lipasti, "Implementing optimizations at decode time," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[29] ——, "Macro-op scheduling: Relaxing scheduling loop constraints," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[30] S. Hu, I. Kim, M. H. Lipasti, and J. E. Smith, "An approach for implementing efficient superscalar cisc processors," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, 2006.

[31] D. L. Howard and M. H. Lipasti, "The effect of program optimization on trace cache efficiency," in *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[32] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processors," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[33] R. Rajwar, M. Dixon, and R. Singhal, "Specialized evolution of the general purpose cpu." in *CIDR*, 2015.

[34] M. DeVuyst, A. Venkat, and D. M. Tullsen, "Execution migration in a heterogeneous-isa chip multiprocessor," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[35] A. Venkat and D. M. Tullsen, "Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor," in *Proceedings of the International Symposium on Computer Architecture*, 2014.

[36] A. Venkat, S. Shamasunder, H. Shacham, and D. M. Tullsen, "Hipstr: Heterogeneous-isa program state relocation," in *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, 2016.

[37] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms," in *Proceedings of the 10th European Conference on Computer Systems*, Apr. 2015.

[38] A. Branković, K. Stavrou, E. Gibert, and A. González, "Performance analysis and predictability of the software layer in dynamic binary translators/optimizers," in *Proceedings of the ACM International Conference on Computing Frontiers*, 2013.

[39] S. Hu and J. E. Smith, "Using dynamic binary translation to fuse dependent instructions," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.

[40] A. Venkat, A. Krishnaswamy, K. Yamada, and R. Palanivel, "Binary Translation driven Program State Relocation," in *United States Patent Grant US009135435B2*, 2015.

[41] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye, "Dynamic binary translation and optimization," *IEEE Transactions on Computers*, 2001.

[42] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2003.

[43] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman, "Denver: Nvidia's first 64-bit ARM processor," *IEEE Micro*, 2015.

[44] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customization," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[45] M. L. Corliss, E. C. Lewis, and A. Roth, "Dise: A programmable macro engine for customizing applications," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, 2003.

[46] ——, "Low-overhead interactive debugging via dynamic instrumentation with dise," in *11th International Symposium on High-Performance Computer Architecture*, 2005.

[47] ——, "Using dise to protect return addresses from attack," *SIGARCH Comput. Archit. News*, Mar. 2005.

[48] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[49] H. M. G. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood, "Surfnoc: A low latency and provably non-interfering approach to secure networks-on-chip," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[50] O. Acıiçmez, W. Schindler, and Ç. K. Koç, "Cache based remote timing attack on the aes," in *Cryptographers Track at the RSA Conference*, 2007.

[51] D. J. Bernstein, "Cache-timing attacks on aes," Tech. Rep., 2005.

[52] K. Mowery, S. Keelveedhi, and H. Shacham, "Are aes x86 cache timing attacks still feasible?" in *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, 2012.

[53] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack." in *USENIX Security*, 2014.

[54] Y. Yarom, D. Genkin, and N. Heninger, "Cachebleed: A timing attack on openssl constant time rsa," in *International Conference on Cryptographic Hardware and Embedded Systems*, 2016.

[55] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers Track at the RSA Conference*, 2007.

[56] E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems," in *Annual International Cryptology Conference*, 1997.

[57] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic

analysis: Concrete results," in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2001.

[58] D. Genkin, A. Shamir, and E. Tromer, "Acoustic cryptanalysis," *Journal of Cryptology*, 2016.

[59] J. J. Hoch and A. Shamir, "Fault analysis of stream ciphers," in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2004.

[60] M. C. W. K. Gruss, Daniel and S. Mangard, *Flush+Flush: A Fast and Stealthy Cache Attack*, 2016.

[61] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[62] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity." in *NDSS*, 2015.

[63] N. Madan, A. Buyuktosunoglu, P. Bose, and M. Annavaram, "A case for guarded power gating for multi-core processors," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011.

[64] S. Dropsho, V. Kursun, D. H. Albonesi, S. Dwarkadas, and E. G. Friedman, "Managing static leakage energy in microprocessor functional units," in *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, 2002.

[65] H. Farrokhbakht, M. Taram, B. Khaleghi, and S. Hessabi, "Toot: an efficient and scalable power-gating method for noc routers," in *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2016.

[66] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis, "Power management of datacenter workloads using per-core power gating," *IEEE Computer Architecture Letters*, 2009.

[67] T. Chen, A. Rucker, and G. E. Suh, "Execution time prediction for energy-efficient hardware accelerators," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[68] 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.*, 2014.

[69] ——, "Efficient power gating of simd accelerators through dynamic selective devectorization in an hw/sw codesigned environment," *ACM Trans. Archit. Code Optim.*, 2014.

[70] A. Fog, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus," http://www.agner.org/optimize/instruction_tables.pdf.

[71] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, March 2009.

[72] G. E. Suh, C. W. O'Donnell, and S. Devadas, "Aegis: A single-chip secure processor," *Information Security Technical Report*, 2005.

[73] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Aegis: Architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003.

[74] ARM, "Arm security technology - building a secure system using trustzone technology," 2009.

[75] "Debian microcode update," https://wiki.debian.org/Microcode, 2017.

[76] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM Trans. Program. Lang. Syst.*, Jul. 1996.

[77] A. Leung, M. Gupta, Y. Agarwal, R. Gupta, R. Jhala, and S. Lerner, "Verifying gpu kernels by test amplification," in *ACM SIGPLAN Notices*, vol. 47, no. 6. ACM, 2012, pp. 383–394.

[78] K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic, "Talx86: A realistic typed assembly language," in *1999 ACM SIGPLAN Workshop on Compiler Support for System Software Atlanta, GA, USA*, 1999.

[79] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and implementation of the aegis single-chip secure processor using physical random functions," in *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*, 2005.

[80] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a practical hardware security architecture through static information flow analysis," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[81] O. Aciiçmez, "Yet another microarchitectural attack:: exploiting i-cache," in *Proceedings of the 2007 ACM workshop on Computer security architecture*, 2007.

[82] D. A. Osvik, A. Shamir, and E. Tromer, *Cache Attacks and Countermeasures: The Case of AES*, 2006.

[83] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose, "Microarchitectural techniques for power gating of execution units," in *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*, 2004.

[84] C. Long and L. He, "Distributed sleep transistors network for power reduction," in *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*, 2003.

[85] H. Jiang, M. Marek-Sadowska, and S. R. Nassif, "Benefits and costs of power-gating technique," in *2005 International Conference on Computer Design*, 2005, pp. 559–566.

[86] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[87] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *Micro, IEEE*, 2006.

[88] "2nd Generation Intel Core vPro Processor Family," Intel, Tech. Rep., 2008, available at http://www.intel.com/content/dam/doc/white-paper/performance-2nd-generation-core-vpro-family-paper.pdf. [Online]. Available: http://www.intel.com/content/dam/doc/white-paper/performance-2nd-generation-core-vpro-family-paper.pdf

[89] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the 2001 IEEE International Workshop on Workload Characterization*, 2001.

[90] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[91] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2010.

[92] O. D. A. Tromer, Eran and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, 2010.