ConTExT: A Generic Approach for Mitigating Spectre

Michael Schwarz[1], Moritz Lipp[1], Claudio Canella[1], Robert Schilling[1,2], Florian Kargl[1], Daniel Gruss[1]
[1]Graz University of Technology [2]Know-Center GmbH

***Abstract*—Out-of-order execution and speculative execution are among the biggest contributors to performance and efficiency of modern processors. However, they are inconsiderate, leaking secret data during the transient execution of instructions. Many solutions and hardware fixes have been proposed for mitigating transient-execution attacks. However, they either do not eliminate the leakage entirely or introduce unacceptable performance penalties.**

In this paper, we propose ConTExT, a *Con*siderate *T*ransient *Ex*ecution *T*echnique. ConTExT is a minimal and fully backward compatible architecture change. The basic idea of ConTExT is that *secrets can enter registers but not transiently leave them*. ConTExT transforms Spectre from a problem that cannot be solved purely in software [65], to a problem that is not easy to solve, but solvable in software. For this, ConTExT requires minimal, fully backward-compatible modifications of applications, compilers, operating systems, and the hardware. ConTExT offers full protection for secrets in memory and secrets in registers. With ConTExT-light, we propose a software-only solution of ConTExT for existing commodity CPUs protecting secrets in memory. We evaluate the security and performance of ConTExT. Even when over-approximating with ConTExT-light, we observe no performance overhead for unprotected code and data, and an overhead between $0\,\%$ and $338\,\%$ for security-critical applications while protecting against all Spectre variants.

I. Introduction

As arbitrary shrinking of process technology and increasing processor clock frequencies is not possible due to physical limitations, performance improvements in modern processors are made by increasing the number of cores or by optimizing the instruction pipeline. Out-of-order execution and speculative execution are among the biggest contributors to the performance and efficiency of modern processors. Out-of-order execution allows processing instructions in an order deviating from the one specified in the instruction stream. To fully utilize out-of-order execution, processors use prediction mechanisms, e.g., for branch directions and targets. This predicted control flow is commonly called speculative execution. However, predictions might be wrong, and virtually any instruction can raise a fault, e.g., a page fault. Hence, in this case, already executed instructions have to be unrolled, and their results have to be discarded. Such instructions are called transient instructions [59], [50], [90], [96], [14], [78].

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.24271
www.ndss-symposium.org

Transient instructions are never committed, *i.e.*, they are never visible on the architectural level. Until the discovery of transient-execution attacks, e.g., Spectre [50], Meltdown [59], Foreshadow [90], [96], RIDL [91], and ZombieLoad [78], they were not considered a security problem. These attacks exploit transient execution, *i.e.*, execution of transient instructions, to leak secrets. This is accomplished by accessing secrets in the transient-execution domain and transmitting them via a microarchitectural covert channel to the architectural domain.

The original Spectre attack [50] used a cache covert channel to transmit data from the transient-execution domain to the architectural domain. However, other covert channels can be used, e.g., instruction timings [50], [80], contention [50], [8], branch-predictor state [24], or the TLB [49], [80]. For other covert channels [98], [97], [34], [22], [60], [44], [26], [32], [64], [72], [79], it is still unclear whether they can be used.

Several countermeasures have been proposed against transient-execution attacks, often relying on software workarounds. However, many countermeasures [99], [47], [48], [3], [36] only try to prevent the cache covert channel of the original Spectre paper [50]. This includes the officially suggested workaround from Intel and AMD [36], [3] to prevent Spectre variant 1 exploitation. However, Schwarz et al. [80] showed that this is insufficient.

State-of-the-art countermeasures can be categorized into 3 classes based on how they try to stop leakage [67], [14]:

1) Mitigating or reducing the accuracy of the covert channel communication, e.g., eliminating the covert channel or making gadgets unreachable [48], [47], [99].
2) Aborting or preventing transient execution when accessing secrets [36], [5], [3], [71], [37], [16], [69], [89].
3) Ensuring that secret data is unreachable [75], [30].

In this paper, we introduce a new type of countermeasure. Our approach, ConTExT, prevents secret data from being leaked in the transient-execution domain by aborting or preventing transient execution only in a very small number of cases, namely when the secret data would leak into the microarchitectural state. ConTExT is efficient and still runs non-dependent instructions out-of-order or speculatively. We show that our approach prevents all Spectre attacks by design.

Implementing ConTExT in CPUs only requires repurposing one page-table entry bit (e.g., one of the currently unused ones) as a *non-transient bit*. Instead of the actual secret value, the CPU uses a dummy value (e.g., '0') when accessing a non-transient memory location during transient execution in a way that could leak into the microarchitectural state. To protect register contents, we introduce a *non-transient bit* per

register. For the special purpose `rflags` register (crucial for control-flow changes), we introduce a `shadow_rflags` register to track the taint bit-wise, *i.e.*, every bit in the `rflags` register has a corresponding taint bit. Same as for the memory locations, the CPU will use a dummy value during transient execution instead of the actual register content.

Today, mitigating certain Spectre variants already requires annotation of all branches that could lead to a secret-dependent operation during misspeculation [46]. We simply move these annotation requirements to the root, such that the developer only has to annotate the actual variables that can hold secrets in the source code. We do not propagate this information on the source level, *i.e.*, we do not perform software-level taint-tracking. Instead, we propagate this information into the binary to create a separate binary section for secrets, using compiler and linker support. For this section, the operating system sets the memory mapping to *non-transient*. We split the stack into an unprotected stack and a protected stack. The protected stack is marked as *non-transient* to be used as temporary memory by the compiler, e.g., for register spills. Local variables are moved to the transient stack. Similarly, we also split the heap into an unprotected and a protected part and provide the developer with heap-allocator functions that use the protected part of the heap. Thus, there is no performance impact for regular variables. Preventing leakage only requires a developer to identify the assets, *i.e.*, secret values, inside an application. Obviously, this is much easier than identifying *all code locations* which potentially leak secret values. If new Spectre gadgets are discovered (e.g., prefetch gadgets [14]), ConTExT-protected applications do not require any changes. In contrast, if every code location which potentially leaks secrets has to be fixed, the application has to be changed for new types of Spectre gadgets.

To emulate the minimal hardware adaptions ConTExT requires, we over-approximate it via ConTExT-light[1], a software-only solution which achieves an over-approximation of the behavior using existing features of commodity CPUs. ConTExT-light relies on the property that values stored in *uncacheable memory* can generally not be used inside the transient-execution domain [21], [59], except for cases where the value is architecturally in registers, or microarchitecturally in the load buffer, store buffer, or line fill buffer. While ConTExT-light does not provide complete protection on most commodity systems due to leakage from these buffers, it can provide full protection on Meltdown- and MDS-resistant CPUs, e.g., on AMD CPUs, as long as secrets are not in registers. In this paper, we focus on mitigating Spectre-type attacks and consider Meltdown-type attacks out-of-scope. ConTExT-light also allows determining an upper bound for the worst-case performance overhead of the hardware solution. However, this upper bound is not tight, meaning that the actual upper bound can be expected to be substantially lower. Compared to practically deployed defenses against certain Spectre variants [46], ConTExT requires only a simpler direct annotation of secrets inside the program, which can be easily added to any existing C/C++ program to protect secrets from being leaked via transient-execution attacks.

We evaluate the security of ConTExT on all known Spectre attacks. Due to its principled design, ConTExT prevents the leakage of secret data in all cases, as long as the developer does not actively leak the secret. We evaluated the performance overheads of ConTExT-light for several real-world application where we identify and annoted the used secrets. Depending on the application, the overhead is between $0\,\%$ and $338\,\%$. In most cases it is lower than the overhead of the currently recommended and deployed countermeasures [36], [5], [3], [71], [75], [16], [53], [87]. To further support the performance analysis, we extended the Bochs emulator with the *non-transient bits* for registers and page tables and extended it with a cache simulator.

Concurrent to our work, NVIDIA patented a closely related approach to our design [10]. However, they do not provide protection for registers, but only for memory locations.

Contributions. The contributions of this work are:

1) We propose ConTExT, a hardware-software co-design for considerate transient execution, fully mitigating transient-execution attacks.
2) We show that on all levels, only minimal changes are necessary. The proposed hardware changes can be partially emulated on commodity hardware.
3) We demonstrate that ConTExT prevents all known Spectre variants, even if they do not rely on the cache for the covert channel.
4) We evaluate the performance of ConTExT and show that the overhead is lower than the overhead of state-of-the-art countermeasures.

Outline. The remainder of this paper is organized as follows. In Section II, we provide background information. Section III presents the design of ConTExT. Section IV details our approximative proof-of-concept implementation on commodity hardware. Section V provides security and performance evaluations. Section VI discusses the context of our work. We conclude our work in Section VII.

II. Background

In this section, we give an overview of transient execution. We then discuss known transient-execution attacks. We also discuss the proposed defenses and their shortcomings.

A. Transient Execution

To simplify processor design and to allow superscalar processor optimizations, modern processors first decode instructions into simpler micro-operations (μOPs) [25]. With these μOPs, one optimization is not to execute them in-order as given by the instruction stream but to execute them *out-of-order* as soon as the execution unit and required operands are available. Even in the case of out-of-order execution, instructions are retired in the order specified by the instruction stream. This necessitates a buffer, called reorder buffer, where intermediate results from μOPs can be stored until they can be retired as intended by the instruction stream.

In general, software is seldom purely linear but contains (conditional) branches. Without speculative execution, a processor would have to wait until the branch is resolved before execution can be continued, drastically reducing performance. To increase performance, speculative execution allows a processor to predict the most likely outcome of the branch using

[1] https://github.com/IAIK/contextlight

various predictors and continue executing along that direction until the branch is resolved.

At runtime, a program has different ways to branch, e.g., conditional branches or indirect calls. Intel provides several structures to predict branches [41], e.g., Branch History Buffer (BHB) [7], Branch Target Buffer (BTB) [57], [23], the Pattern History Table (PHT) [25], and Return Stack Buffer (RSB) [25], [63], [51]. On multi-core CPUs, Ge et al. [26] showed that the branch prediction logic is not shared among physical cores, preventing one physical core from influencing another core's prediction.

Speculation is not limited to branches. Processors can, e.g., speculate on the existence of data dependencies [35]. In the case where the prediction was correct, the instructions in the reorder buffer are retired in-order. If the prediction was wrong, the results are squashed, and a rollback is performed by flushing the pipeline and the reorder buffer. During that process, all architectural but no microarchitectural changes are reverted. Any instruction getting executed out-of-order or speculatively but not architecturally is called a *transient instruction*. *Transient execution* may have measurable microarchitectural side effects.

B. Transient-Execution Attacks & Defenses

While transient execution does not influence the architectural state, the microarchitectural state can change. Attacks that exploit these microarchitectural state changes to extract sensitive information are called transient-execution attacks. So-called Spectre-type attacks [50], [35], [51], [63] exploit prediction mechanisms, while Meltdown-type attacks [59], [90], [91], [78], [13], [96] exploit transient execution following an architectural or microarchitectural fault.

Kocher et al. [50] first introduced two variants of Spectre attacks. The first, Spectre-PHT (Variant 1), exploits the PHT and the BHB such that the processor mispredicts the code path following a conditional branch. If the transiently executed code loads and leaks the secret, it is called a Spectre gadget. Kiriansky and Waldspurger [49] extended this attack from loads to stores, enabling transient buffer overflows and, thus, extending the number of possible Spectre gadgets.

Spectre-BTB (Variant 2) [50] targets indirect branches and poisons the BTB with attacker-chosen destinations, leading to transient execution of the code at this attacker-chosen destination. An attacker mistrains the processor by performing indirect branches within the attacker's own address space to the address of the chosen address, regardless of what resides at this location. Chen et al. [17] showed that this can also be exploited in SGX.

For a memory load, the processor checks the store buffer for stored values to this memory location. Spectre-STL (Variant 4) [35], Speculative Store Bypass, exploits when the processor transiently uses a stale value because it could not find the updated value in the store buffer, e.g., due to aliasing.

Spectre-RSB [51] and ret2spec [63] are Spectre variants targeting the RSB, a small hardware stack of recent return addresses pushed during recent *call* instructions. When a *ret* is executed, the top of the RSB is used to predict the return address. An attacker can force misspeculation in various ways, e.g., by overfilling the RSB, or by overwriting the return address on the software stack.

All of the attacks discussed above have three things in common. First, they all use transient execution to access data that they would not access in normal, considerate execution. Second, they use this data to influence the microarchitectural state, which can be observed using microarchitectural attacks, e.g., Flush+Reload [100]. Third, all are executed locally on the victim machine, requiring the attacker to run code on the machine. Schwarz et al. [80] extended the original Spectre attack with a remote component and demonstrated that the microarchitectural state of the AVX2 unit can be used instead of the cache state to leak data.

Meltdown-type attacks exploit deferred handling of exceptions. They do not exploit misspeculation but use other techniques to execute instructions transiently. Between the occurrence of an exception and it being raised, instructions that access data retrieved by the faulting instructions can be executed transiently. The original Meltdown attack [59] exploited the deferred page fault following a user/supervisor bit violation, allowing to leak arbitrary memory. A variation of this attack allows an attacker to read system registers [5], [36]. Van Bulck et al. [90], [96] demonstrated that this problem also applies to other page-table bits, namely the present and the reserved bits. Canella et al. [14] analyzed different exception types, based on Intel's [42] classification of exceptions as *faults*, *traps*, and *aborts*. They found that all known Meltdown variants so far have exploited faults, but not traps or aborts. With so-called *microarchitectural data sampling* (MDS) attacks, Meltdown-type effects have been demonstrated on other internal buffers of the CPU. RIDL [91] and ZombieLoad [78] leak sensitive data from the fill buffers and load port. Fallout [13] exploits store-to-load forwarding to leak previous stores from the CPUs store buffer.

Defenses. Since the discovery of Spectre, many different defenses have been proposed. The easiest and most radical solution would be to entirely (or selectively) disable speculation at the cost of a huge decrease in performance [50]. Intel and AMD proposed a similar solution by using serializing instructions on both outcomes of a branch [3], [36]. Evtyushkin et al. [24] proposed to allow a developer to annotate branches that could leak sensitive data, which are then not predicted. Unfortunately, on Intel CPUs, serializing branches does not prevent microarchitectural effects such as powering up AVX units, or TLB fills [80].

For mitigating the RSB attack vector, Intel proposes RSB stuffing [37]. Upon each context switch, the RSB is filled with the address of a benign gadget.

Google Chrome limits the amount of data that can be extracted by introducing *site isolation* [75]. Site isolation relies on process isolation, *i.e.*, each site is executed in its own process. Thus, Spectre attacks cannot leak secrets of other sites. Speculative Load Hardening [16] and YSNB [69] are similar proposals, both limiting speculation by introducing data dependencies between the array access and the condition.

SafeSpec [47] and InvisiSpec [99] introduce additional shadow hardware for speculation. The results of transient instructions are only made visible to the actual hardware when

the processor determined that the prediction was correct. Both methods require major changes to the hardware.

DAWG [48] is another proposal requiring significant hardware changes. The idea is to partition the cache to create protection domains that are disjoint across ways and metadata partitions. Additionally to hardware changes, the approach requires changes to the replacement policy and cache coherence protocol to incorporate the protection domain.

All local Spectre variants so far use either Flush+Reload [100], [50], [35], [51], [63] or Prime+Probe [70], [88] to extract information from the covert channel, requiring access to a high-resolution timer. Thus, a defense mechanism is to reduce the accuracy of timers [66], [73], [86], [93] and eliminate methods to construct different timers [79].

To mitigate Spectre variant 2, both Intel and AMD extended the ISA with mechanisms to control indirect branches [4], [38], namely Indirect Branch Restricted Speculation (IBRS), Single Thread Indirect Branch Prediction (STIBP), and Indirect Branch Predictor Barrier (IBPB). With IBRS, the processor enters a special mode, and predictions cannot be influenced by operations outside of it. STIBP restricts the sharing of branch prediction mechanisms among hyperthreads. IBPB allows flushing the BTB. Future processors implement enhanced IBRS [37], a hardware mitigation for Spectre variant 2. With *retpoline* [89], Google proposes an alternative technique to protect against branch poisoning by ensuring that the return instruction predicts to a benign endless loop through the RSB. Similarly, Intel proposed *randpoline* [12], a heuristic but more efficient version of retpoline.

To mitigate Spectre variant 4, Intel provides a microcode update to disable speculation on the store buffer check [38]. The new feature, called Speculative Store Buffer Disable (SSBD), is also supported by AMD [2]. ARM introduced a new barrier (`SSBB`) which prevents loads after the barrier from bypassing a store using the same virtual address before the barrier [5]. Future ARM CPUs will feature a configuration control register that prevents the re-ordering of stores and loads. This feature is called Speculative Store Bypass Safe (SSBS) [5].

So far, all the proposed defense mechanisms against Spectre attacks either require substantial hardware changes or only consider cache-based covert channels. In the latter case, an attacker can circumvent the defense by using a different covert channel, e.g., AVX [80], TLB [77], or port contention [8]. This focus on cache covert channels only and the huge decrease in performance caused by state-of-the-art Spectre defenses shows the necessity for the development of efficient and effective defenses.

To mitigate microarchitectural attacks on the kernel, and specifically KASLR breaks, Gruss et al. [30] proposed KAISER, a kernel modification unmapping most of the kernel space while running in user mode [30]. As KAISER also mitigates Meltdown, the idea of KAISER has been integrated into all major operating systems, e.g., in Linux as KPTI [62], in Windows as KVA Shadow [43], and in Apple's xnu kernel as double map [58]. With the PCID and ASID support of modern processors, the performance overheads appear acceptable for real-world use cases [28], [29]. Additionally, to mitigate Foreshadow [90] on SGX enclaves, microcode updates are necessary. To mitigate Foreshadow-NG [96], several further steps need to be implemented for full mitigation. The kernel must use non-present page-table entries more carefully, e.g., not store the swap disk page frame number there for swapped-out pages. When using EPTs (extended page tables), the hypervisor must make sure that the L1 cache does not contain any secrets when switching into a virtual machine.

To mitigate MDS attacks [91], [78], [13], microcode updates are necessary that enable a legacy instruction to flush the affected microarchitectural buffers [40]. Furthermore, in environments utilizing simultaneous multithreading, the operating system must only schedule processes within the same security domain to sibling threads to mitigate user to user attacks [40]. To protect from attacks against the kernel, the operating system must guarantee a synchronized entry and exit on system calls and interrupts such that no untrusted user code is executed on a sibling thread [40]. To replace the expensive software workarounds, newer CPU microarchitectures provide fixes in hardware and, thus, are already resistant against Meltdown-type attacks [39]. In this paper, we focus on mitigating only Spectre-type attacks and consider Meltdown-type attacks out-of-scope.

C. Taint Analysis

Taint tracking is used to track data-flow dependencies on a hardware level [19], [83], binary-level [18], [76], or source level [81]. Taint analysis has a wide range of security applications: detecting vulnerabilities, e.g., by tracking untrusted user input; malware analysis, e.g., analyzing information flows in binaries; test case generation, e.g., automatically generating inputs. This can be either done statically [6], [94] or dynamically [68], [74].

Dynamic taint analysis allows tracking the information flow between sources and sinks [76]. Any value that depends on data derived from a tainted source, e.g., user input, is considered *tainted*. Values that are not derived from tainted sources are considered *untainted*. A policy defines how taint flows as the program executes and how new taints are introduced. Over-approximation can occur when tainting a value that is not derived from a taint source.

Taint tracking has also been proposed on a hardware level [92], [45], [56], [10], yet not in the context of speculative execution.

III. Design of ConTExT

In this section, we present the design of ConTExT, a considerate transient execution technique.

The idea of ConTExT is to introduce a new type of memory mappings, namely *non-transient* mappings. The *non-transient* option indicates that the mapping contains secrets that must not be accessed within the transient-execution domain. Consequently, *non-transient* values must not be used in transient operations, neither directly nor in a derived form, *iff* the effect of the transient operation could be microarchitecturally observable. Thus, there cannot be any perturbations of the microarchitectural CPU state, which might disclose *non-transient* values via side channels. To track whether a value is *non-transient* and must be protected, registers also track the

non-transient state. To ensure not only the original but also derived values are protected, this information is propagated to the results of operations using these values, until the secret is destroyed, e.g., by overwriting it.

Security Claim. A processor with ConTExT mitigates all speculative execution attacks as the processor cannot use *non-transient* registers in any way that would influence the microarchitectural state. Hence, if a software implementation is leakage-free on a strict in-order machine, it will also be leakage-free on an out-of-order or speculative machine with ConTExT, iff secrets are annotated.

ConTExT is a multi-level countermeasure which works on the application-, compiler-, operating-system-, and hardware-level. An application developer annotates secret values, and possible memory destinations for secret values in the source code, which the compiler groups inside the binary and marks as secret.

Besides annotation of secrets, it would also be possible to architecturally define groups of secrets, e.g., based on the data type as suggested by Carr and Payer [15], or by defining all userspace memory and user input as secret as proposed by Taram et al. [85]. However, this can be very expensive, and consequently, related work is also investigating annotation-based protection mechanisms [101].

When the operating system loads the binary, memory regions containing the annotated secrets are marked *non-transient*. The hardware does all subsequent tracking of secrets. The operating system only has to be aware of secret register states on interrupts, e.g., context switches. Other than these minimal changes, there are no additional adaptions required on any level of the software stack.

The full-protection ConTExT requires small hardware changes, which retrofits already existing mechanisms in today's CPUs, *i.e.*, there is no re-design required. Moreover, the change is fully backward compatible with existing hardware and software (*i.e.*, applications, libraries, and operating systems). As hardware changes cannot be conducted on commodity CPUs, we evaluate ConTExT based on ConTExT-light, an over-approximation which only requires software changes. As illustrated in Figure 1, ConTExT is a more considerate variant of transient execution. An unprotected application executes all instructions, including the instruction leaking the secret. In contrast, with the state-of-the-art solution of using lfences, the CPU stalls at the fence and aborts the transient execution, *i.e.*, it cannot continue to transiently execute any instruction at all. ConTExT has the advantage that the instructions leaking the secret are not executed, while independent instructions (marked with arrows) later on in the instruction stream can still be executed during the out-of-order execution. Although these instruction cannot retire, they already warm up caches and buffers, e.g., by triggering prefetchers. With ConTExT-light, the memory location containing the secret is marked as uncachable, which already leads to a CPU stall in current processors when accessing the memory location in transient execution. However, independent instruction can still be executed during the out-of-order execution. Current CPUs implement this by executing memory loads for memory marked as uncachable only at retirement, *i.e.*, the corresponding load instruction is only executed if it is at the head of the reorder buffer [27]. This is also the case for the `lock` prefix [20]. We envision to use the same mechanism for ConTExT.

ConTExT protects secrets which are stored in cache and DRAM, *i.e.*, attackers cannot access data from memory locations marked as *non-transient* during transient execution, and registers if they have been filled with data from protected cache or DRAM locations or other protected registers. ConTExT-light cannot protect secrets while they are architecturally stored in registers of running threads. Furthermore, ConTExT-light is not designed as a protection against Meltdown-type attacks. Mitigating Meltdown-type attacks, including MDS attacks, is orthogonal to our work, and we consider it out of scope. We only use it to obtain an upper bound for the performance overheads of ConTExT. Note that this upper bound is not tight, *i.e.*, the actual upper bound is expected to be substantially lower.

ConTExT is a multi-level countermeasure consisting of 3 major components which we describe in this section:

1) *non-transient* memory mappings (cf. Section III-A),
2) tracking of *non-transient* data (cf. Section III-B), and
3) software (*i.e.*, OS, compiler, and application) support for the hardware features (cf. Section III-C).

A. Non-Transient Memory Mappings

We present three possible implementations of *non-transient* memory mappings, *i.e.*, memory mappings, which indicate that the values cannot be used during transient execution.[2] All variants allow integrating ConTExT into the current architecture while maintaining backward compatibility, *i.e.*, if the operating system is not aware of ConTExT, the changes have no side effects. Hence, to implement ConTExT, *only one of the following* variants has to be implemented.

Currently Reserved Page-Table Entry Bit. There is already sufficient space to store the *non-transient* bit in the page tables of commodity CPUs. On Intel 64-bit (IA-32e) systems, each page-table entry has 64 bits, comprised of a 52-bit physical-address field and several flags. However, most processors do not support full 52 bits, but only up to 46 bits, which allows working with up to 64 TB of physical RAM if the hardware supports it.

Figure 2 shows a page-table entry for x86-64. Besides the already used bits, there are the 6 bits between bit 46 and 51, which are currently reserved for future use. This future use could be the extension of the physical page number if more physical memory is supported in future CPU generations. However, it could also be the repurposing of one of the bits (e.g., the last reserved bit) as a *non-transient* bit. This reduces the theoretical maximum amount of supported memory by factor 2. Thus, instead of $4\,\mathrm{PB}$, CPUs could only support $2\,\mathrm{PB}$ of physical memory. The repurposing of a reserved bit is automatically backwards-compatible, as the reserved bits currently have to be '0'. Hence, using such a bit does not have any undesirable side effects on legacy software.

[2]Concurrent to our work, NVIDIA patented a proposal closely related to our design [10]. However, they do not provide protection for registers, but only for memory locations. Similarly, also in concurrent work, Intel released a whitepaper introducing the idea of a new memory type against Spectre attacks [84].

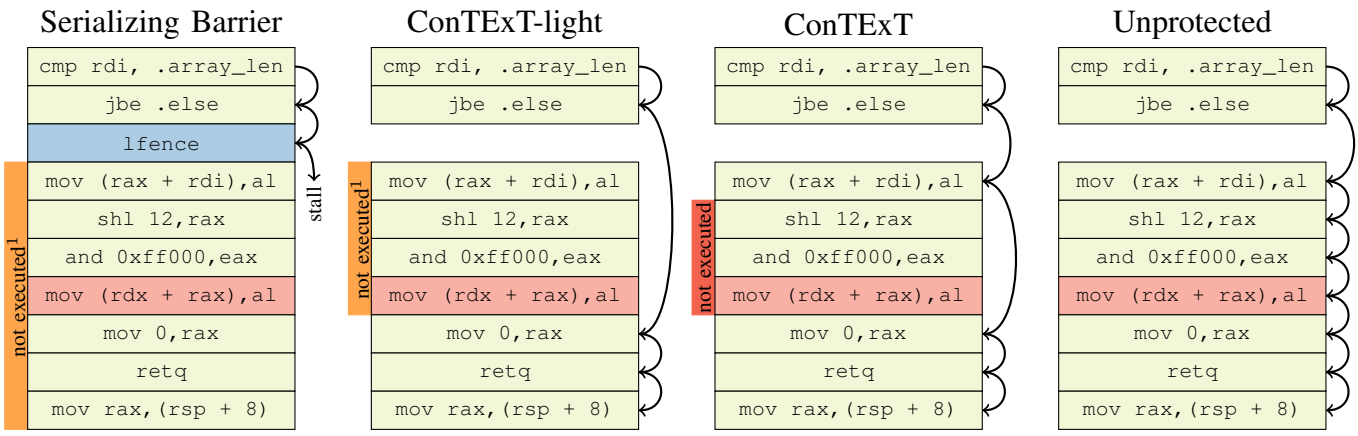

Fig. 1: Comparison of ConTExT with the current solution against the first Spectre attack example [50]. The leaking access, *i.e.*, the only line that must not be executed, is highlighted. The arrows show which instructions can be executed in the out-of-order execution. An unprotected application executes all instructions, including the one leaking the secret. Serializing barriers and ConTExT-light provide protection against Spectre-type attacks on commodity systems, as empirically shown in Figure 3.

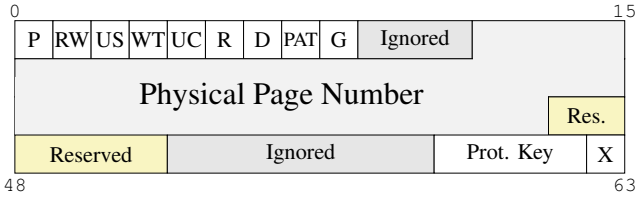

Fig. 2: A page-table entry on x86-64 consists of 64 bits which define properties of the virtual-to-physical memory mapping. Besides the already used bits, physical page number, and ignored bits (which can be freely used), there are 6 physical address bits that are currently reserved for future use since hardware is limited to 46-bit physical addresses. Future processors may support longer physical addresses.

Currently Ignored Page-Table Entry Bit and Control Register. An alternative to using one of the reserved bits is to use one of the ignored bits. These bits can be freely used by the operating system, thus, simply repurposing them is not possible. However, if the feature has to be actively enabled, the operating system is aware of the changed semantics of the specific ignored bit. Note that this approach was already taken for several other page-table bits, e.g., the protection key and the global bit are enabled via `CR4` and they are ignored otherwise. Hence, we also propose enabling the feature using a bit in one of the CPU control registers, e.g., `CR4`, `EFER`, or `XCR0`. These registers are already used for enabling and disabling security-related features, such as read-only pages, NX (no-execute) or SMAP (supervisor mode access prevention). Moreover, these registers still have up to 54 unused control bits which can be used to enable and disable the *non-transient* bit.

An advantage of repurposing an ignored bit is that CPU vendors do not lose potential address-space bits. That is, this approach is compatible with physical address spaces of up to 4 PB in future hardware. However, the approach comes with the limitation that operating systems cannot freely use the retrofitted ignored bit anymore, as it is now used as the *non-transient* bit.

TABLE I: The currently supported memory types which can be used in the PAT (Intel 64-bit systems), and the additional non-transient type (bold-italic) as new memory type.

Value	Type	Description
0	UC	Strong uncacheable, never cached
1	WC	Write Combining (subsequent writes are combined and written once)
2	***NS***	***Non-transient, cannot read in transient execution domain***
3	-	Reserved
4	WT	Write Through (reads cached, writes written to cache and memory)
5	WP	Write Protected (only reads are cached)
6	WB	Write Back (reads/writes are cached)
7	UC-	Uncacheable, overwritten by MTRR

Memory Type using Page-Attribute Table. A third alternative is to retrofit the Page-Attribute Table (PAT), a processor feature allowing the operating system to reconfigure various attributes for classes of pages. The PAT allows specifying the *memory type* of a memory mapping. On x86, there are currently 6 different memory types which define the cache policy of the memory mapping.

Table I shows the memory types which can be set using the PAT, including our newly proposed non-transient memory type. The PAT itself provides 8 entries for memory types. Such a PAT entry is applied to a memory mapping via the 3 page-table-entry bits '3' (write through), '4' (uncacheable), and '7' (PAT). These 3 bits combined to a 3-bit number select one of the 8 entries of the PAT.

Thus, to apply the non-transient memory type to a memory mapping, the OS sets one of the PAT entries to the non-transient memory type '2'. Then, this PAT entry can be applied through the existing page-table bits to any memory mapping. As the PAT supports 8 entries, and there are currently only 6 memory types (7 if the non-transient type is included), it is still possible to use all supported memory types concurrently

on different pages, *i.e.*, the approach is fully backwards-compatible.

An advantage of this approach is that no semantic changes have to be made to page-table entries, *i.e.*, all bits in a page-table entry keep their current meaning. However, this variant may require more changes in the operating system, as e.g., Linux already utilizes all of the PAT entries (some memory types are defined twice).

B. Secret Tracking

Non-transient mappings ensure that *non-transient* memory locations cannot be accessed during transient execution. However, we still need to protect secret data that is already loaded into a register. Registers in commodity CPUs do not have a memory type or protection. Thus, we require changes to the hardware to implement protection of registers. Based on patents from Intel [45], VMWare [56], and NVIDIA [10], we expect such tracking features to be implemented in future CPUs. Venkataramani et al. [92] proposed a technique in hardware that also taints registers, however, to identify software bugs rather than overly eager speculative execution.

Tainting Registers. For ConTExT, we introduce one additional *non-transient* bit per register, *i.e.*, a *taint* (cf. Section II-C). The *non-transient* bit indicates whether the value stored in the register is *non-transient* or not. If the bit is set, the entire register is marked as *non-transient*, otherwise, the register is unprotected. The taint generally propagates from memory to registers and from registers to registers. The rationale behind this is that results of operations on secret data have to be considered secret as well. Accessing only parts of a tainted register, e.g., `eax` instead of `rax`, still copies the taint from the source register to the target register and taints the entire target register, as we only have a single *non-transient* bit per register. This is also true for taint propagation in any other use of a tainted register.

One special case is the `rflags` register. The `rflags` register is a special purpose register, updated upon execution of various instructions. For the `rflags` register, we introduce a `shadow_rflags` register to track the taint bit-wise due to the special use of the single bits in this register for control flow. The taint propagation rules still apply, but the bits of `rflags` are tainted independently. Operations that update the `rflags` register can execute transiently. However, using a tainted bit from the `rflags` propagates the taint to the target operands in the case of register targets. For memory targets, regardless of the secret value, a default value is returned. Finally, branching on a tainted bit from the `rflags` stalls the pipeline to prevent any leakage. In general, we assume that the protected application is written in a side-channel-resistant manner. Hence, there should not be any secret-dependent branches. If there are such branches, ConTExT protects them but it might lead to unnecessary stalls.

We keep taint propagation very simple and consider only instructions with registers as destination operands. If any *non-transient* memory location is used as a source operand to an instruction, the instruction taints the destination registers, *i.e.*, the *non-transient* bit is set for every destination register. Similarly, if any *non-transient* register is used as a source operand to an instruction, the instruction also taints the destination registers. Thus, if a secret is loaded into a register, it is tracked through all register operations.

The taint is not propagated if the destination operand(s) are memory location(s), as all memory locations already have a *non-transient* bit managed by the operating system. However, if the instruction directly, or due to the fact that the destination operand(s) are memory location(s), influences the microarchitectural state, the instruction does not use the actual secret value but instead either stalls or works with a dummy value. This also includes branch instructions if the corresponding `shadow_rflags` bit is set. That is, branching on a secret stalls the pipeline.

Untainting Registers. There are not only operations which taint registers, but also operations which untaint registers. Replacing the entire content of a register without using *non-transient* memory or registers untaints the register. We do this to avoid over-tainting registers; a problem pointed out in earlier works [82]. In particular, all immediate or untainted values which replace the content of a register untaint it. Writing a tainted register to a normal memory location, *i.e.*, a memory location which is not marked as *non-transient*, also untaints the register. The rationale behind this is that if registers are spilled to normal (*i.e.*, insecure) memory locations, a potential secret can be leaked anyway. If such a memory operation happens unintentionally, it is a bug in the program and has to be fixed at the software level. As the developer has knowledge of secrets used in the application, it is assumed that the developer moves secrets only to memory locations marked as *non-transient* if the secrets should stay secret. In many cases, however, moving secrets to normal memory is intentional behavior, as the developer decided that the register does not contain a secret anymore. For instance, the output of a cryptographic cipher does not need protection from transient-execution attacks. Thus, the automated untainting keeps the number of tainted registers small.

Taint Propagation across Memory Operations. As the taint bit is an additional bit for each register, it can only be propagated to other registers, not to memory. If an operation writes a secret (*i.e.*, tainted) register to memory, the taint bit is irrecoverably lost. While this is intended if the developer explicitly writes values to memory, it might have undesirable consequences if this happens implicitly, e.g., due to the inner workings of the compiler. In Section III-C, we introduce the required changes to the compiler which ensure that the compiler never accidentally spills non-transient values to transient memory locations.

However, the compiler inevitably has to temporarily store (insecure) registers within memory regions marked as *non-transient*. With the solution as described so far, we would over-approximate and taint more and more registers over time by spilling them to *non-transient* memory locations and reading them back from there. Hence, spilling registers is not a security problem (*i.e.*, tainted registers are never untainted, only untainted registers are tainted), but a loss in performance due to unnecessarily tainted registers.

Optimizing Performance via Caching. To prevent this potential performance loss, we propose an additional change to the cache to reduce the impact of the taint over-approximation. We introduce one additional bit of meta data per 64 bits to

the cache, *i.e.*, 8 additional bits of meta data per 64 B cache line. This allows us to store the register-taint information transparently in the cache. Note that this change does not influence the architectural size of a cache line, as it only extends the meta data that is already stored for each cache line. Whenever a register is written to *non-transient* memory, the taint bit of the register is stored in the corresponding cache line. When reading from memory, the bit stored in the cache line has precedence over the information from the TLB, *i.e.*, the cache overwrites the taint bit defined by the memory mapping. The information in the cache allows the hardware to temporarily keep track of the taint information of a register if the register value is moved to the stack. This happens, e.g., if register values are spilled on the stack, exchanged via the stack, or upon function calls.

Evicting the cache line corresponding to a register is never a security issue. An evicted cache line only loses the information that a register was *not tainted*. Thus, if the cache line is evicted, the registers become automatically tainted.

1) Taint Control: Besides the automated tainting and untainting of registers, ConTExT provides a privileged interface to modify the taint of registers. This interface is necessary for the operating system to save and restore taint values upon context switches.

A straightforward solution would be to introduce new instructions in the ISA. However, we try to keep the hardware changes to a minimum, especially changes which are not hidden in the microarchitecture. Hence, we propose instead to use model-specific registers (MSR) to access the taint information of registers.

Read/Write Taint. To read and write the current taint information of all registers, we introduce an MSR `IA32_TAINT`. The taint bit of every architectural register directly maps to one bit of this MSR, which allows the operating system to read and write all taint bits in a single operation. As there are only 56 architectural registers (16 general purpose, 8 floating point, 32 vector) which have to be tracked, one 64-bit MSR is sufficient to read or write all taint bits at once. While the physical register file typically contains more registers, these are not visible to the developer. Hence, the MSR only has to provide access to the taint bits of the architectural registers.

Interrupt Handling. MSRs can only be accessed indirectly using an instruction (*i.e.*, `rdmsr` on x86), and require registers both to specify the MSR and as source and destination operands. On an interrupt, the first thing to save should be the `IA32_TAINT` MSR, because it contains the taints of the previous context. However, as registers must not be clobbered in the interrupt routine, all the registers used in the interrupt handler have to be saved first. We resolve this problem by automatically copying the `IA32_TAINT` to an additional MSR, `IA32_SHADOW_TAINT`, on every interrupt. This ensures that the taint of all registers is preserved before any taint is potentially modified by a register operation in the interrupt handler. The `IA32_SHADOW_TAINT` can then be treated like any other register, e.g., the operating system can save it into a kernel structure upon a context switch.

When returning from an interrupt, the CPU restores the values from `IA32_SHADOW_TAINT` to the register taint values. Hence, with this mechanism, we ensure that an interrupt does not influence the taint value of any register. This also works for the unlikely event of nested interrupts, *i.e.*, if an interrupt is interrupted by a different interrupt. The only critical region in such a case is if the first interrupt has not yet locally saved the `IA32_SHADOW_TAINT` MSR, and the second interrupt overwrites the MSR. However, as long as within this critical region (*i.e.*, the time window between first interrupt and second interrupt) no register is untainted, there can be no leakage. In Section III-C, we show that this situation can be avoided solely in software.

C. Software Support

We propose changes to applications, compilers, and operating systems to leverage the hardware extensions introduced in Section III-A and Section III-B. The idea is that instead of annotating all branches that potentially lead to a secret-dependent operation, application developers simply annotate the secret variables in their applications directly. These annotations are processed by the compiler and then forwarded to the operating system to establish the correct memory mappings (cf. Section III-A).

Compiler. The compiler parses the annotations of secrets. Annotations are already implemented in modern compilers, e.g., with `__attribute__((annotate("secret")))` in clang. The secrets identified this way are allocated inside a dedicated section of the binary. The compiler marks this section as *non-transient*. The operating system maps this section from the binary using a *non-transient* memory mapping.

Besides parsing the annotations, our modified compiler ensures that it never spills data from registers marked as secret into unprotected memory. Otherwise, an attacker could leak the spilled secrets from memory. Still, it is unavoidable that the compiler spills registers to the stack, e.g., to preserve register contents over function calls. Furthermore, due to the calling convention, some (possibly secret) values have to be passed over the stack. Hence, we have to assume that the stack contains secrets. As a consequence, the stack has to be mapped using a *non-transient* memory mapping as well.

To reduce the performance impact of a *non-transient* stack, we modify the compiler to only use the *non-transient* stack if really necessary. This *non-transient* stack only contains register spills, possibly function arguments, and return values. All other values are stored at a different memory location, the *unprotected* stack. This concept is similar to the SafeStack [52] and our implementation even reuses parts of the SafeStack infrastructure of modern compilers. The difference to SafeStack, where only "unsafe" memory allocations (e.g., buffers) are stored on the SafeStack, is that we move all variables normally allocated on the stack to the *unprotected* stack. Thus, for ConTExT, only the absolute minimum is stored on the *non-transient* stack, e.g., return addresses. By only moving local variables to the *unprotected* stack, and leaving return addresses and function arguments on the stack, we do not break ABI compatibility with existing binaries. Thus, a developer can still use external libraries without recompiling them, and libraries compiled for ConTExT can be used in ordinary unprotected applications.

Moving local variables from the stack to a different memory location does not impact the runtime of the application and

```
pushall
rep xor rcx, rcx ; clear rcx, rep prefix keeps taint
add rcx, IA32_TAINT
rdmsr ; taint in rax, rdx
[...]
popall
push rax, rcx, rdx
mov rcx, IA32_TAINT ; also updates IA32_SHADOW_TAINT
wrmsr ; old taint in rax, rdx
pop rax, rcx, rdx
iret ; restores IA32_SHADOW_TAINT to registers
```

Listing 1: (Pseudo-)assembly for saving and restoring the taint MSR without destroying the taint of any other register during a context switch.

even gives additional protection against memory-corruption attacks [52].

Operating System. For ConTExT, the operating system is in charge of setting up *non-transient* memory mappings. As the operating system parses the binary, it can directly set up the *non-transient* memory mappings, which are marked as such by the compiler. The operating system requires additional small changes. The operating system has to save and restore taint values on context switches. The hardware already saves the current taint value of all registers into the `IA32_SHADOW_TAINT` MSR upon interrupts. Thus, the operating system only has to read this register and save it together with all other saved registers.

As interrupts can be interrupted by other interrupts, e.g., a normal interrupt can be interrupted by a non-maskable interrupt (NMI), there is a critical section between reading the MSR and saving the result. If registers are untainted in this section, a nested interrupt would lose the taint information as it overwrites the `IA32_SHADOW_TAINT` MSR. However, if registers are not untainted in this section, no taint information can be lost. Hence, we have to initialize the registers required to read the MSR in a way that does not destroy the taint. For this purpose, we define that the `rep` prefix for arithmetic and logical operations on registers preserves the taint. Section III-C shows (pseudo-)assembly code, which prepares the registers with the required immediate values. Generally, overwriting a register with an immediate or by using an idiom, e.g., `xor rax,rax`, untaints the register. However, the `rep` prefix prevents the untainting here.

In addition to the context switch, the operating system has to flush the cache when the content of a *non-transient* memory location is initially loaded from the binary. This is important as the initial data transfer to the memory page is not done through the *non-transient* user-space mapping. Thus, the operating system has to either disable the cache before this operation or flush the corresponding cache lines afterwards. This functionality is already present in the x86 ISA and supported by modern operating systems. Thus, there is no further change required.

IV. Implementation of ConTExT

In this section, we present our implementation of both ConTExT and ConTExT-light, which we use for the evaluation (cf. Section V). As we cannot change real x86 hardware or emulate the hardware changes required for ConTExT on commodity hardware, we opted for a hardware simulation of our changes using a full-system emulator (cf. Section IV-A). While this does not allow to measure performance by measuring the runtime, it allows measuring performance in the number of memory accesses, *non-transient* memory accesses, taint over-approximations, etc., for real-world benchmarks.

For ConTExT-light, we present a method to partially emulate the non-transient memory mapping behavior on commodity hardware by retrofitting uncacheable memory mappings. Thus, in Section IV-B, we present an open-source proof-of-concept implementation of ConTExT-light which can already be used and evaluated on commodity hardware. As ConTExT-light is running on a real modern CPU architecture, the results are more tangible than a simulation-based evaluation. Hence, the performance overhead is an over-approximation, and any real hardware implementation is expected to be more efficient than ConTExT-light, as the CPU has to stall in fewer cases.

ConTExT-light is not designed as a protection against Meltdown-type attacks. Mitigating Meltdown-type attacks, including MDS attacks, is orthogonal to our work, and we consider it out of scope.

A. Hardware Simulation

We simulated ConTExT using the open-source x86-64 emulator Bochs [55] to get as close as possible to functionally extending a real x86-64 processor with our features, *non-transient* memory mappings (cf. Section III-A) as well as secret tracking (cf. Section III-B). We incorporated hardware and behavioral changes in our ConTExT-enabled Bochs.

For the hardware simulation, we considered alternatives, such as the gem5 simulator [9] or an out-of-order RISC-V core. However, gem5, as Bochs, is a software-based emulation, and the overhead estimations from gem5 do not match the actual overheads in practice, as layouting and microarchitectural details have a huge influence on real hardware. Currently, there is also no open-source implementation of a last-level cache for RISC-V, and it would be difficult to reason about the performance overheads on x86 based on a RISC-V implementation. Hence, we implement the behavioral changes in Bochs to analyze the functionality and use ConTExT-light on a real CPU to approximate the performance overhead (cf. Section IV-B).

Hardware Changes. To support secret tracking, a few minor hardware changes are required. Mostly, these are single bits to track whether a register is *non-transient*. These bits are required in every page-table entry, TLB entry, and register. Furthermore, we introduce additional meta-data bits per cache line to minimize the performance cost of register spills (cf. Section III-B).

Page-Table Entry. To distinguish *non-transient* from normal memory mappings, we have to mark every memory mapping accordingly in the PTE. For backward- and future-compatibility, repurposing one of the ignored bits is the best choice (cf. Section III-A). Furthermore, repurposing a bit ensures that the change does not result in any runtime or memory overhead. If this bit is set, we treat the memory

mapping as a region which may contain secrets. The developer has to do that both for memory locations containing secrets, as well as memory locations where secrets are (temporarily) stored.

Translation Lookaside Buffer. For performance reasons, modern CPUs cache page-table entries in the TLB. Consequently, we need an additional *non-transient* bit in the TLB, caching the bit of the page-table entry. In Bochs, caching of page-table entries is also implemented as a TLB-like structure allowing the simulated hardware to automatically transfer the added bit from the PTE to the TLB. Thus, for cached page-table entries, memory accesses use the cached *non-transient* bit from the TLB.

Cache. Bochs only implements an instruction cache, but no data cache, which plays a vital role in our design to cache taint information (cf. Section III-B). Hence, we extended Bochs with data-cache emulation by implementing an 8-way (inclusive) last-level cache. As the exact eviction strategy is unknown [31], we used LRU as a good approximation as it has been used in Intel CPUs until Ivy Bridge [31]. In our emulated cache, we added 8 taint bits as metadata per cache line. Note that this change does not influence the architectural size of the cache or a cache line. While this sounds like a large amount of additional metadata, it amounts to less than $1.6\,\%$ increase of the size of the last-level cache. Considering that every cache line already holds a large amount of metadata (e.g., physical tag, cache-coherency information, possibly error-detection bits), these additional 8 bits of metadata do not result in a large hardware overhead, and are fully backward compatible.

Model-Specific Registers. As described in Section III-B, we added two new MSRs to Bochs. Accesses to `IA32_TAINT` are directly mapped to the taint bits of the architectural registers, allowing the operating system to read and write all at once. While the physical register file contains more registers [41], we still require only two MSRs, as they only provide access to the taint bits of the current architectural registers. As a typical x86 CPU already contains several hundred MSRs [42], [1], adding two new MSRs per CPU core is a negligible hardware overhead. To save the current taint state on interrupts (Section III-B1), we ensure data consistency between the two MSRs; a write to `IA32_TAINT` also (atomically) updates `IA32_SHADOW_TAINT`. This enables us to implement secure context switches (cf. Section III-C).

Behavioral Changes. All behavioral changes are only enabled if the operating system supports and enables ConTExT using the corresponding bit in the control register (cf. Section III-A). However, taint tracking is enabled unconditionally as it happens implicitly without additional cost. This applies to all operations which transfer data from memory to registers or from registers to registers. In our proof-of-concept implementation, we added the taint tracking to 368 out of 557 instructions implemented in Bochs. If no memory mapping is marked as *non-transient*, then no register can be tainted. Thus, taint tracking simply has no effect if there is no operating system support.

B. ConTExT-light

In addition to the hardware emulation for ConTExT, we implemented ConTExT-light (cf. Section III) for Linux. Our implementation of ConTExT-light consists of two parts, a kernel module, and a runtime library. For the full ConTExT, we provide a compiler extension that minimizes the performance penalties of register spills.

For the proof of concept, we emulate *non-transient* memory mappings via *uncacheable* memory mappings. Uncacheable memory can generally not be accessed inside the transient execution domain [21], [59] and we consider Meltdown-type attacks out-of-scope since they are already fixed on most recent hardware [59], [91], [78]. In contrast to ConTExT, ConTExT-light does not protect secrets while they are architecturally stored in registers of running threads. Thus, the security guarantees of ConTExT-light still hold in this case.

Kernel Module. We opted to implement the operating-system changes as a kernel module for compatibility with a wide range of kernels. The kernel module is responsible for setting up *non-transient* memory mappings. As our proof-of-concept implementation relies on uncacheable memory, we do not retrofit page-table bits but use the page-attribute table to declare a memory mapping as uncacheable.

The kernel module provides an interface for the runtime library (cf. Section IV-B) to set up *non-transient* memory mappings. This allows keeping the changes in the kernel space minimal as most of the logic and parsing can be implemented in user space. The kernel module ensures that the page-attribute table contains an *uncacheable* (UC) entry by reprogramming the page-attribute table if this is not already the case. If the runtime library requests a mapping to be marked *non-transient* via the kernel-module interface, the page-table entry is modified to reference the UC entry in the page-attribute table. Subsequently, the corresponding TLB entry is flushed. We do not flush all cache lines of the mapping, as this would incur additional overhead. Thus, the developer (or runtime library) has to take care that values stored on pages marked as *non-transient* are not cached before they are marked as *non-transient*.

Runtime Library. The runtime library sets up all static and dynamic *non-transient* memory mappings via the kernel-module interface. Our proof-of-concept runtime library supports C and C++ applications and can even be included as a single header file for simple projects. The header file provides a keyword, `nospec`, to annotate variables as secrets using the `__attribute__` directive. This keyword ensures that the linker allocates the variables in a dedicated `secret` section in the ELF binary. Moreover, the header file registers a constructor function which is executed before the actual application, to initialize ConTExT at runtime.

When the application starts, the runtime library identifies all memory mappings in the `secret` section from the ELF binary. These memory mappings are then set to *non-transient* (*i.e.*, uncacheable) using the kernel module.

The runtime library is only active on application startup and does not influence the application during runtime. During runtime, it is only used if the developer requests dynamic *non-transient* memory, *i.e.*, *non-transient* heap memory. For this

purpose, the runtime library provides a `malloc_nospec` and `free_nospec` function. These functions mark the allocated heap memory immediately as *non-transient*.

Compiler. For the full ConTExT with hardware support, we also require compiler support. We extend the LLVM compiler [54] in version 8.0.0 to not use the stack for local variables, but move them to a different part of the memory which we refer to as *unprotected stack*.[3] The normal stack is marked as *non-transient* to not leak temporary variables and function parameters the compiler puts on the stack. Thus, to reduce the performance impact, we allocate local variables which are defined by the developer in the unprotected stack, which is not marked as *non-transient*.

Our implementation is based on the already existing SafeStack extension [52]. We modify the heuristics to not move only specific but all user-defined variables from the *non-transient* stack to the unprotected stack (SafeStack in the original extension). Allocations coming from function parameters and registers spills are put on the *non-transient* stack.

V. Evaluation

In this section, we evaluate ConTExT and ConTExT-light with respect to their security properties and their performance. We evaluate ConTExT on our modified Bochs emulator, and ConTExT-light on a Lenovo T480s (Intel Core i7-8650U, 24 GB DRAM) running Ubuntu 18.04.1 with kernel version 4.15.0.

A. Security

We generally assume that the operating system is trusted as it handles the *non-transient* memory mappings. First, we explain how ConTExT can be used to protect against all Spectre attacks, and how current commodity hardware can be retrofitted to partially emulate ConTExT. Second, we show the limitations of ConTExT.

1) Security of ConTExT: The security guarantees of ConTExT are built on two assumptions: the application developer correctly annotated all secrets as such, and the application does not actively leak secrets (e.g., by writing them to memory locations not marked as *non-transient*). ConTExT guarantees for code that is leakage-free on a strict in-order machine that this code will also be leakage-free on an out-of-order or speculative machine with ConTExT, iff secrets are correctly annotated. For the evaluation, we distinguish two cases, based on whether the secret values are used architecturally in the application or not while an attacker mounts a transient-execution attack.

Security Argument. ConTExT eliminates leakage of secrets from transient-instruction execution into the microarchitectural state. It is trivial to see that allowing no transient-instruction execution eliminates any leakage. ConTExT allows the transient execution of instructions that do not influence the microarchitectural state. An implementation, e.g., our proof-of-concept, defines for each instruction whether it has to stall (e.g., branch instructions if the corresponding taint bit is set), use a dummy value instead of the secret value (e.g., operations with one or more secret input operands and one or more memory input or output operands, and operations that influence "uncore" or off-core microarchitectural elements), or can run in an unmodified way (e.g., pure on-core register operations). If an implementation correctly restricts these, the microarchitectural state cannot be influenced by a secret. Hence, in the extreme case where the entire memory is secret, it is straightforward to see that ConTExT would not allow any transient-instruction execution. More specifically, ConTExT allows exactly the subset of instructions in the instruction stream to run transiently that do not influence the microarchitecture based on secrets.

[3]The patches can be found in our GitHub repository https://github.com/IAIK/contextlight.

```
1 char oracle[256 * 4096];
2 // nospec for ConTExT-light
3 char /*nospec*/ secret = 'X';
4
5 if(speculate()) {
6     // LFENCE here for mitigation
7     oracle[secret * 4096]; // encode secret
8     oracle['E' * 4096]; // encode public value
9 }
```

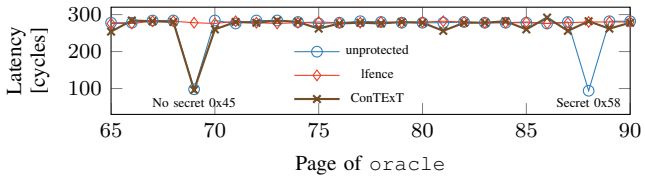

Fig. 3: Evaluation of Figure 1. The unprotected code snippet leaks the secret 'X' (0x58) and public value 'E' (0x45) to the cache (Lines 7 to 8). State-of-the-art lfence-based mitigation (`lfence` in Line 6) prevents both indices from being cached. A ConTExT-light annotation (Line 3) prevents the secret index from being cached but allows the public index to be cached, warming up the cache.

Architecturally Unused Secrets. A secret is architecturally unused if the secret is only stored in a *non-transient* memory region, *i.e.*, there is no part of the secret which is stored in a register, cache, or normal memory region. For example, this is the case if the secret was not used by the time of an attack. However, the application can also be in such a state, although the secret has already been used in the past. If all traces of the secret in normal memory or the cache are already overwritten (or evicted), the application returns again to the state where secrets are architecturally unused.

In this state, an attacker can only target the secret itself and not an unprotected copy of it. It is clear that such an attack cannot be successful, as—per-definition—transiently executed code cannot retrieve the value from a *non-transient* memory region. Hence, ConTExT is secure if its implementation fulfills this property.

Architecturally Used Secrets. If the entire secret, or parts of it, are stored in a register, cache, or a memory region not marked as *non-transient*, the secret is considered architecturally used. In this case, an attacker can target any unprotected copy of the secret, not only the original secret stored in the *non-transient* memory region. However, an attack fails if the target is marked as secret, e.g., by a *non-transient* memory mapping, tainted register, or tainted cache line.

If a *non-transient* memory region is loaded into a register, the register is tainted and, thus, it cannot be targeted. Moreover, the taint is also applied to the corresponding cache line and TLB entry. Any register-to-register operation which copies the secret also copies the taint. Similarly, an operation that copies the secret to a *non-transient* memory region is also secure. Such operations include, for example, register spills to the stack, temporary storage of registers in local variables, or secrets as function arguments (depending on the calling convention). Tainted registers can only be untainted by destroying their content, *i.e.*, overwriting them with non-secret values. Overwriting a register with an immediate or by using an idiom, e.g., `xor rax,rax`, generally untaints the register. Using the `rep` prefix on arithmetic or logical register operations preserves the taint.

Thus, registers cannot be untainted while containing a secret. However, over-approximation can lead to more tainted registers than necessary.

Operations that copy the secret to a memory region not marked as *non-transient* could be attacked. However, such operations are never implicitly generated by the compiler, as the compiler only uses the stack as a temporary memory. Thus, such an operation has to be explicitly defined by the application developer, which violates the assumption that the application does not actively leak secrets.

A remaining scenario is the context switch of the application with used secrets. In such a case, the application is stopped by the operating system, and the current register content is saved to the kernel. As the operating system is aware of register taints, and also considered trusted, it can leverage the taint saving mechanism described in Section III-B1. The registers can again be saved in a *non-transient* memory region to prevent transient-execution attacks on the saved registers. When returning from the kernel, all registers are first tainted (an over-approximation, as they are restored from a *non-transient* stack), but the original taint is restored just before the end of the context switch. Thus, registers containing secrets are always tainted and cannot be targeted.

2) Security Limitations of ConTExT-light: As ConTExT-light is implemented using uncacheable memory, we evaluated the security properties of uncacheable memory regarding transient execution. We use the transient-execution proof-of-concepts from Canella et al. [14] as test cases to verify that ConTExT-light prevents any leakage of secret data. For all proof-of-concepts which are applicable to our test system, we successfully leaked the secrets before deploying ConTExT-light. We furthermore used the AVX-based Spectre-PHT variant from Schwarz et al. [80] to verify that ConTExT-light also prevents Spectre attacks, which do not use the cache as a covert channel. To verify the effectiveness of ConTExT-light in our experimental setup, we mark the memory mapping containing the secret data as uncacheable using the PAT. Additionally, using Flush+Reload, we verified that the memory mapping is actually uncacheable. For all tested proof-of-concepts, ConTExT-light successfully prevented any leakage of the secret data (cf. Figure 3).

ConTExT-light cannot protect secrets while they are architecturally stored in registers of running threads. Furthermore, ConTExT-light is not designed as a protection against Meltdown-type attacks. Mitigating Meltdown-type attacks, including MDS attacks, is orthogonal to our work, and we consider it out of scope.

3) Limitations: ConTExT can only be effective if used correctly by the application developer, *i.e.*, if the developer marks all secrets as secret and does not actively leak secrets. However, even if used correctly, there are certain limitations which mostly result from a trade-off between performance and security. In the following paragraphs, we point out where application developers must take care to not accidentally leak secrets.

ConTExT does not allow taint to leave from registers to the microarchitectural state. Hence, we have to stall the pipeline if secret registers would influence the control flow, e.g., a modification of the instruction pointer based on the flags register.

Instructions such as CRC32 might also leak secrets if a secret value is used as input, either directly or in combination with an attacker-known value. However, as this is again a secret-dependent operation, the developer has to ensure that this does not leak any secrets.

Another responsibility of the developer is that secret values are not actively copied to memory locations not marked as *non-transient*. This cannot be prevented by either the compiler or the hardware, as it is often necessary, e.g., the tainted output of a crypto operation (ciphertext) is not secret anymore and can be written to normal memory.

ConTExT-light. As ConTExT-light is only a partial emulation of ConTExT, it comes with some limitations compared to ConTExT. The largest difference to ConTExT is that secrets in registers, the load buffer, the store buffer, and the line fill buffer are not protected. Thus, if a secret is in one of these microarchitectural structures, it remains susceptible to transient-execution attacks.

B. Performance

We evaluated the performance of ConTExT-light as an upper bound for the performance overhead of ConTExT. This upper bound is not tight, and the actual upper bound can be expected to be substantially lower. We also evaluate the performance overhead of ConTExT based on our full-system emulation in Bochs. The SPECspeed 2017 evaluation for the baseline and of the *unprotected stack* of ConTExT is performed on an i7-8700K machine while all other evaluations are performed on an i7-8650U machine. Both systems run Ubuntu Linux 18.04.1 with kernel 4.15.0.

We evaluated the software implications of our proposed hardware changes using our modified version of Bochs and a modified Linux kernel, based on kernel version 4.15. For the Linux kernel, we only had to modify 52 lines in 9 files to support the save and restore of register taints on context switches. These small changes result in a negligible performance overhead on context switches, e.g., for syscalls.

The latency of syscalls increases by a constant value, which is 48 cycles (averaged over $500\,000$ syscall invocations). On a standard Ubuntu Linux installation, we observed between 3000 and 5000 syscalls per second on average while performing regular office tasks. On our test system, we observe an

TABLE II: Performance evaluation of the *unprotected stack* of ConTExT using the SPECspeed 2017 integer benchmark. The baseline was compiled with the unmodified compiler, the ConTExT run uses our modified LLVM compiler.

Benchmark	SPEC Score		Overhead
	Baseline	ConTExT	[%]
600.perlbench_s	7.03	6.86	+2.42
602.gcc_s	11.90	11.80	+0.84
605.mcf_s	9.06	9.16	−1.10
620.omnetpp_s	5.07	4.81	+5.13
623.xalancbmk_s	6.06	5.95	+1.82
625.x264_s	9.25	9.25	0.00
631.deepsjeng_s	5.26	5.22	+0.76
641.leela_s	4.71	4.64	+1.48
648.exchange2_s	*would require Fortran runtime*		
657.xz_s	12.10	12.10	0.00
Average			+1.26

overhead on the system load of around $0.01\,\%$ at this syscall rate. The highest syscall rates observed for real-world use cases at Netflix was reported to be around $50\,000$ syscalls per second [28]. On our test system, we observe an overhead on the system load of around $0.13\,\%$ at this syscall rate.

1) Compiler Extension: We evaluated the impact of the *unprotected stack* of ConTExT using the SPECspeed 2017 integer benchmark. Table II shows that similarly to the original SafeStack implementation [52], the resulting performance overhead is $1.26\,\%$ on average and, in the worst case, $5.13\,\%$.

These results are not surprising as only addresses of variables change. This only requires very little runtime code for maintaining a second stack pointer. Thus, the small performance overhead is mostly due to the setup time for the additional *non-transient* stack.

We furthermore evaluated the performance impact introduced by the *non-transient* stack. As a baseline, we consider the case where we only have one *non-transient* stack and compare it to our design where the *non-transient* stack is only an additional stack to the regular unprotected one. Based on Intel Pin [61], we implemented our own plugin to trace all memory accesses. With the plugin, we evaluated how much memory the *non-transient* stack consumes. For this purpose, we ran the GNU Core Utilities, once compiled with the unmodified compiler, and once compiled with our extended LLVM compiler. Even for these lightweight applications, we measured a reduction of average *non-transient* stack memory by $42.74\,\%$. The modified LLVM compiler sustained an average *non-transient* stack usage of $4.7\,\mathrm{kB}$, whereas the applications compiled with a vanilla compiler consumed, on average, $8.2\,\mathrm{kB}$ on the single *non-transient* stack. Moreover, for 64 out of the 91 tested applications (*i.e.*, $70.3\,\%$), the compiler extension reduced the *non-transient* stack usage to only $3528\,\mathrm{B}$, which is below the smallest memory region that can be set *non-transient*, *i.e.*, the size of one virtual page ($4\,\mathrm{kB}$). The reason for these reductions is that the stack is not used anymore for storing user-defined variables. Hence, the compiler extension makes it practical to deploy ConTExT with the additional *non-transient* stack.

2) ConTExT-light: We evaluated the performance impact of ConTExT-light, both for unmodified applications as well as applications where we annotate secret values as such. For unmodified applications, we do not expect any runtime overhead, except for a constant initialization overhead.

We confirmed this assumption experimentally. The average initialization overhead when starting an application with our current non-optimized implementation is $0.15\,\mathrm{ms}$.

For applications with annotated secret values, there is a performance overhead for architectural accesses to the secret. Without ConTExT-light, the secret could be stored in the L1, L2, or L3 cache, or the main memory. Hence, the maximum overhead for a memory access is the difference between an L1 cache hit and a cache miss. The minimum overhead for a memory access is zero (*i.e.*, cache miss in both cases). In practice, we often see a cache miss instead of an L3 cache hit, which makes an average overhead of 100 cycles on our test system.

To evaluate the real-world performance, we applied ConTExT-light to various real and artifical applications.[4] We first evaluate ConTExT-light on pure cryptographic algorithms, as they are the main target for Spectre attacks and thus require protection. In addition to performance evaluations on pure cryptographic algorithms, we also evaluate the performance of real-world application when annotating secrets. In all cases, the effort to identify and annotate secrets only required changing between 3 and 27 lines in the source code.

OpenSSL RSA. We evaluated the performance by encrypting a message using OpenSSL's RSA. For this, we provide OpenSSL with the secure heap allocation functions of ConTExT-light. We verified that indeed all memory allocations in OpenSSL use the secure functions using `ltrace` and single-stepping. The performance overhead we measured when annotating all buffers that may (temporarily) contain secrets in an RSA encryption is $71.14\,\%$ ($\pm\ 4.66\,\%$, $n = 10\,000$). This is not surprising as RSA performs many in-place operations in one secure buffer, and hence, higher overheads are expected.

AES. As a second cryptographic algorithm, we evaluated AES, both in OpenSSL and in a custom AES-NI implementation. For our AES-NI implementation, we annotate the AES key as well as the intermediate round keys as secrets. For AES-NI, no other secret values, or values derived from secrets, have to be stored in memory. As AES-NI expects all values in the `xmm` registers, there is only the initial performance overhead of copying the ConTExT-light-protected keys to the registers. As this is a one-time operation, the overhead of 122 cycles ($n = 10\,000\,000$, $\sigma_{\bar{x}} = 0.00$), is negligible when performing multiple encryptions or decryptions. For the encryption and decryption step, there is no performance overhead at all. We verified this by encrypting and decrypting a block $10\,000\,000$ times. Both with and without ConTExT-light, the encryption and decryption took 46 cycles per 16-byte block. While the application is an artificial application, it shows that ConTExT-light-protected cryptographic algorithms can be implemented without any performance overhead.

To analyze the performance overhead of ConTExT-light on a state-of-the-art AES implementation, we used OpenSSL's

[4]The changes to existing applications and the artifical applications can be found in our GitHub repository https://github.com/IAIK/contextlight.

AES-128-CBC. Similarly to the AES-NI example, we measured the number of cycles it takes to encrypt and decrypt the same block. Without ConTExT-light, it takes on average 1371 cycles ($n = 100\,000$, $\sigma_{\bar{x}} = 36.90$). For the protected variant, we annotated the key as secret, and for simplicity, the entire internal encryption and decryption context `EVP_CIPHER_CTX` of OpenSSL. While this protects more variables than necessary, it ensures that all secrets in the context of the encryption and decryption are marked as uncachable. Even then, the overhead is not too drastic with an average number of cycles for encryption and decryption of 5196 ($n = 100\,000$, $\sigma_{\bar{x}} = 32.82$). This naïve approach only requires to provide ConTExT-light's implementation of the heap management to OpenSSL using `CRYPTO_set_mem_functions` and annotating the key using the `nospec` attribute. We verified using GDB that all occurrences of the secret key are only stored in uncachable memory. The result is that secret AES keys cannot be extracted anymore using Spectre attacks, with a performance overhead of 338 % ($n = 100\,000$, $\sigma_{\bar{x}} = 0.24$).

However, as we showed with the AES-NI example, this can still be improved by modifying the OpenSSL library itself, and ensuring that only sensitive data is marked as such.

OpenSSH. For OpenSSH, the main asset is the private key which is stored in memory and which is susceptible to Spectre attacks.[5] Hence, to evaluate the impact of protecting the private key with ConTExT-light, we evaluate OpenSSH with our modifications.

Conveniently, OpenSSH already encapsulates the private key into its own global variable `sensitive_data`. The variable is a structure of type `Sensitive` which can store an arbitrary number of SSH keys. The private keys are stored in `sshbuf`s and referenced by the `sensitive_data` variable. Hence, to apply ConTExT-light, we annotated the global variable and changed the heap allocations in the `sshbuf` functions to use the heap-manipulation functions provided by ConTExT-light. This resulted in a change of 14 lines of code.

To benchmark the impact of the modification, we analyzed the time it takes to connect to an SSH server, as well as how long it takes to transfer a file from a server. The connection time, which includes the initialization time of ConTExT-light, increased on average by 24.7 % ($n = 1000$, $\sigma_{\bar{x}} = 0.038$) from 369 ms to 459 ms. However, this amortizes when, e.g., transferring files. When copying a 128 MB file over SSH in a local network, this overhead is only 5.4 % ($n = 1000$, $\sigma_{\bar{x}} = 0.006$) anymore. Furthermore, as soon as the connection is established, there is no performance impact of ConTExT-light noticeable.

VeraCrypt. Gruss et al. [33] presented a Meltdown attack on the master password of VeraCrypt, the successor of TrueCrypt. As we expect this attack to be possible with Spectre assuming a suitable gadget is found, we show that ConTExT-light can protect the key material in VeraCrypt. VeraCrypt uses a `SecureBuffer` class to store sensitive data, such as the master password. Such a `SecureBuffer` is used, amongst others, for the header key and the encrypted volume header. Hence, it is sufficient to protect all instances of `SecureBuffer` using ConTExT-light. This requires only 3 lines of additional code.

[5] https://marc.info/?l=openbsd-cvs&m=156109087822676&w=2

As the password and keys are used for mounting and encrypting data, we analyze the performance overhead for these operations introduced by ConTExT-light. For mounting an encrypted container, the average time increases by 3.21 % ($n = 1000$, $\sigma_{\bar{x}} = 0.001$) from 1.59 s to 1.64 s. To test the encryption performance, we copy 4 files each with 128 MB to the mounted container. In this experiment, we measure an average overhead of 0.13 % ($n = 200$, $\sigma_{\bar{x}} = 0.006$), increasing the time for the file operations by 0.6 ms. The reason for this small overhead is that the bottleneck is the SSD and not the encryption. On our i7-8650U, we achieve an encryption speed with AES of 4.6 GB/s, which is significantly faster than the SSD write speed. Hence, for file operations, there is no observable performance overhead caused by ConTExT-light.

OATH One Time Password Tool. The OATH One Time Password tool `oathtool` is used to generate one-time passwords for second-factor authentication. This tool supports the Time-based One-time Password algorithm (TOTP), which is used e.g., for Google's or Facebook's two-factor authentication. Based on a shared secret between the user and the service, the tool calculates a cryptographic hash over the shared secret and the current time. A part of this hash is then used as the one-time password for the authentication. An attacker who can extract the shared secret can generate a one-time password at any time. Hence, we use ConTExT-light to protect this shared secret. We do not protect the one-time password, as this is just a temporary second factor that is valid for at most 30 s.

Adding ConTExT-light to `oathtool` requires only 27 lines of code changes in 7 files. The main changes ensure that the buffers storing the shared secret, as well as the buffers used for the hash calculations, are marked as uncachable. This is achieved by allocating them on the non-cachable heap using `malloc_nospec` instead of on the stack or normal heap. We verified the functional correctness of the changes by comparing the generated one-time passwords with the Google Authenticator application. As new passwords are only generated every 30 seconds, any performance overhead introduced by ConTExT-light is not relevant.

Password Manager. LastPass is a tool that can be used to generate and securely store passwords and other sensitive data. The command-line client, LastPass-cli, connects to a remote server with user-provided credentials and retrieves or stores the password and additional information on the remote server, e.g., notes or attachments. To access the data, a user requires the master password associated with an account for the first access. This will store an encrypted local version of the data from the server on the user's disk. The second access will use a key stored by an agent to decrypt the local version of the data. Hence, we protect the password as well as the decryption key as all other transmitted data is short-lived.

We enhanced LastPass-cli by adding ConTExT-light, which requires changing 19 lines of code. These changes ensure that buffers storing the master password, as well as the decryption key, are marked as uncacheable. We tested the application repeatedly to ensure functional correctness. To evaluate the performance slowdown of ConTExT-light, we repeatedly queried a password. In the experiment, we observed a slowdown from 0.162 s to 0.248 s for a slowdown of 53 % with ConTExT-light applied.

NGINX. NGINX is a web server that can also be used for a variety of other tasks, e.g., as a load balancer, mail proxy, and HTTP cache. Similar to other web servers, NGINX allows for secure connections to a client via HTTPS. To authenticate that the client is communicating with the server, the client verifies the server identity by checking its signature generated using the certificate key, *i.e.*, the server's private key, with the certificate, *i.e.*, the server's public key. Hence, when extracting the certificate key, the attacker can impersonate the server.

We modified NGINX to protect the certificate key using ConTExT-light, which requires changing 11 lines of code. We do not protect individual sessions as the session keys are short-lived and, hence, very hard to extract using a Spectre attack. To determine the effect of ConTExT-light on the performance of NGINX, we configured a local server with a generated certificate and used the *siege* load testing and bechmarking utility.[6] With *siege*, we simulate 255 clients for a duration of 300 s. With this test, we observe a decrease from 63 695 to 59 071 transactions, a decrease of 7.3 %. The average response time per transaction increased from 0.62 s to 0.65 s.

Protected Data and Overhead Comparison. In all evaluated applications, the amount of protected data is relatively small. Sensitive data with the highest value for an attacker is mostly either a password, passphrase, or key. Leaking a password usually gives an attacker full access to the application or the rest of the data. Hence, this is the preferable target for a Spectre attack. Especially given the leakage and error rate of Spectre attacks, it is only feasible to leak small amounts of data. In an artificial proof-of-concept, Spectre-PHT achieved up to 10 kB/s, whereas the fastest real-world attack only achieves 41 B/s with an error rate of 2 % [50]. Similarly, Spectre-BTB achieves 1809 B/s with an error rate of 1.7 % [50]. While these leakage rates are sufficient to extract a password or private key, it is not feasible to leak larger amounts of data, such as emails or databases. Moreover, Spectre attacks also require a specific knowledge of where the data is located in memory [50]. Hence, locating the targeted data might also require leaking other data first, e.g., pointers, reducing the effective leakage rate further.

State-of-the-art Spectre mitigations always have a performance impact on the software, regardless whether secrets are present: For instance serialization barriers, the recommended mitigation strategy for Spectre-PHT attacks, cause a high performance overhead, *i.e.*, 62–74.8 % [16]. Additional overheads are caused by Spectre-BTB mitigations, e.g., retpoline (5–10 %), or alternatively STIBP (30–50 %) [53] and IBRS (20–30 %) [87], as well as mitigations for other Spectre variants.

ConTExT reduces the overheads for non-annotated software to a minimum (cf. Table II). The performance overheads for annotated software when heavily using secrets is similar to the state-of-the-art Spectre mitigations. However, this is often just for a small period of time, e.g., for authentication. Hence, ConTExT is a viable alternative as its overhead is inherently lower than the ones we observe with ConTExT-light, and ConTExT-light already is in the range of state-of-the-art mitigation approaches. ConTExT improves the performance of ConTExT-light by regular caching and hiding the latency of register loads. Hence, the performance will be higher.

[6] https://github.com/JoeDog/siege

VI. Discussion

ConTExT is not a defense for commodity systems. ConTExT requires changes across all layers. Yet, compared to all other defenses, it is the first proposal to achieve complete protection [67], [14]. Concurrent to our work, NVIDIA patented a similar idea [10]. However, they focus solely on the protection of memory locations, *i.e.*, not speculating on memory that might contain secrets. As NVIDIA only provides a patent and no whitepaper or scientific paper, it does not discuss any changes required to the software level, e.g., the operating system, compiler, or applications. Hence, there is also no evaluation of the expected overheads. In contrast to their work, we do provide protection on a register-level, allowing speculatively cache and register fills. This clearly has a lower performance impact. However, the various patents in this area [45], [56], [10] give us additional confidence in the practicality of our approach.

Naturally, ConTExT is particularly interesting in cases where isolation is not clear, e.g., to protect a sandbox environment from the sandboxed code. There are different ways to select what are secrets to protect. One extreme would be to generally mark all data secret. As this is not practical, related works either restrict it to an architecturally already defined group, or let the user annotate secrets. Taram et al. [85] defined all userspace memory and user input as secret. However, this can be very expensive, and consequently, Yu et al. [101] proposed a less expensive annotation-based protection mechanism. While this is an important discussion, it is orthogonal to this work. In related work, Brahmakshatriya et al. [11] annotate secrets and modify LLVM to store the annotated and derived secrets in a separate memory area. This approach is similar to our approach. However, they do not try to mitigate Spectre attacks, but memory leaks caused by traditional vulnerabilities. Similarly, Carr and Payer [15] use data annotations to split memory into sensitive and non-sensitive memory ranges based on the data type. These papers show that annotating secrets is a feasible approach to protect against memory leaks. Our work shows that if we can mark secrets, we can provide complete protection against Spectre attacks. From a problem which is, according to Mcilroy et al. [65], currently not solvable in software, ConTExT shifts the landscape such that the problem is not *easy* to solve, but *solvable* in software. ConTExT is the foundation to research future proposals investigating how annotations can be automated, replaced, or simplified. Having a backward-compatible way to annotate secrets and propagate this information through the microarchitecture can be an alternative to something like a CHERI-based processor [95].

Inadvertent Untainting. In line with countermeasures against side-channel attacks, the countermeasure does not protect secrets if a developer actively exposes the secret, e.g., by writing it to memory not marked as *non-transient*. Even with ConTExT, it is the developer's responsibility to take care of secrets, *i.e.*, when temporarily storing them somewhere.

ConTExT only ensures that the *compiler* does not implicitly copy annotated secrets to insecure memory locations, e.g., when temporarily storing register values on the stack to free a register. The developer is assumed to have the domain knowledge on whether a particular variable is a secret. Hence, we expect the developer to correctly decide whether data can be moved to a normal memory location. If sensitive data has to

be copied to a different memory location, then the destination has to be marked as *non-transient* as well. Moving sensitive data only between registers is handled by the hardware taint tracking. This does not complicate the workflow of a developer. Currently, a developer has to decide for every *branch* whether it can leak a value, and whether this value is a secret.

Secret Aliases. Pointer aliases to secret values marked as *non-transient* are not a problem, as the pointer value itself (*i.e.*, the address) is not a secret. The check whether a memory area is marked as *non-transient* is done at the page-table entry (which might already be in the TLB). Pointer aliases still point to the same physical location, *i.e.*, the secret, and hence the same page-table entry is used in the access. The CPU detects the memory type upon this access and either stalls or continues, independent of which pointer was used for the access. For multiple mappings of the same memory location, *i.e.*, shared memory, all mappings must be marked *non-transient* unless the programmer intends to keep one of them non-secret.

Dealing with Edge Cases. There are many elements in a processor that generally could leak data such that a register contains a secret. No matter where the data was leaked from—the memory, the cache, the line fill buffer, the load buffer, the store buffer, or just another register—if the register is tainted, ConTExT does not execute any operation that depends on the value from that register. Hence, under the assumption that the secret has to move through a register (or already be in a register), the protection ConTExT provides is complete. Only violating this assumption would allow bypassing ConTExT. To the best of our knowledge, there is no mechanism on x86-64 that would allow performing an indexed array access without loading the index into a register. This supports our assumption.

As ConTExT prevents the value from being passed on from the tainted register, we do not have any edge cases around the various microarchitectural elements.

Microcode. ConTExT likely cannot be implemented (efficiently) in microcode or microcode updates. The reason is that the behavior in the critical path when forwarding a value from a register to a dependent instruction has to be modified. To the best of our knowledge, there is no microcode involved in this part for performance reasons.

Virtualization. Our approach is oblivious to virtualization. EPTs equally contain *non-transient* bits. Identical to the way several other page table bits are combined (e.g., the *non-executable* bit), if any bit in the hierarchy is set to *non-transient*, the page is *non-transient*. Naturally, the extensions we implemented on the operating system level would have to be identically implemented on the hypervisor level. We leave this implementation effort for future work.

Implementation of the Microarchitectural Changes. While a microarchitectural implementation would be interesting, this is not necessary to see the practicality of our work. We already have the uncacheable memory mapping, which is marked in the page table. Uncacheable memory is not used during speculative execution, although if it is already in a cache, line fill buffer, load buffer, or store buffer, it might be leaked. Hence, there is already a mechanism in current processors, which is very similar to the one we propose. While uncacheable memory is much slower than what we propose with ConTExT, it clearly shows that an implementation is possible and provides an upper bound for the performance overhead.

VII. Conclusion

In this paper, we presented ConTExT, a technique to effectively and efficiently prevent leakage of secrets during transient execution. The basic idea of ConTExT is to transform Spectre from a problem that cannot be solved purely in software [65], to a problem that is not easy to solve, but solvable in software. For this, ConTExT requires minimal modifications of applications, compilers, operating systems, and hardware. We implemented these in applications, compilers, and operating systems, as well as in a processor simulator.

Mitigating all transient-execution attacks with a principled approach of course costs performance. We provide an approximative proof-of-concept for ConTExT which we use on commodity systems to obtain an upper bound for the performance overhead. We argue why the actual performance overhead for ConTExT can be expected to be substantially lower. As seen in our security evaluation, ConTExT is the first proposal for a principled defense tackling the root cause of transient-execution attacks. ConTExT has no performance overhead for regular applications. Even with the over-approximation of ConTExT-light, namely between $0\,\%$ and $338\,\%$ for security-critical applications, it is still below the combined overhead of recommended state-of-the-art mitigation strategies. The overhead with ConTExT will be substantially lower for most real-world workloads. Our work shows that transient execution can be made secure while maintaining a high system performance.

Acknowledgments

We thank our anonymous reviewers for their comments and suggestions that helped improving the paper. The project was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402). It was also supported by the Austrian Research Promotion Agency (FFG) via the K-project DeSSnet, which is funded in the context of COMET - Competence Centers for Excellent Technologies by BMVIT, BMWFW, Styria and Carinthia. This work has additionally been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the Province of Styria and the Business Promotion Agencies of Styria and Carinthia. This work has also been supported by the Austrian Research Promotion Agency (FFG) via the competence center Know-Center (grant number 844595), which is funded in the context of COMET – Competence Centers for Excellent Technologies by BMVIT, BMWFW, and Styria. Additional funding was provided by generous gifts from ARM and Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] AMD, *Software Optimization Guide for AMD Family 17h Processors*, Jun. 2017.

[2] AMD, "AMD64 Technology: Speculative Store Bypass Disable," 2018, revision 5.21.18.

[3] AMD, "Software Techniques for Managing Speculation on AMD Processors," 2018, revison 7.10.18.

[4] AMD, "Software techniques for managing speculation on AMD processors," 2018.

[5] ARM Limited, "Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism," 2018.

[6] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, 2014.

[7] S. Bhattacharya, C. Maurice, S. Bhasin, and D. Mukhopadhyay, "Template attack on blinded scalar multiplication with asynchronous perf-ioctl calls," *Cryptology ePrint Archive, Report 2017/968*, 2017.

[8] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: exploiting speculative execution through port contention," in *CCS*, 2019.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, 2011.

[10] D. D. Boggs, R. Segelken, M. Cornaby, N. Fortino, S. Chaudhry, D. Khartikov, A. Mooley, N. Tuck, and G. Vreugdenhil, "Memory type which is cacheable yet inaccessible by speculative instructions," 2019, uS Patent App. 16/022,274.

[11] A. Brahmakshatriya, P. Kedia, D. P. McKee, D. Garg, A. Lal, A. Rastogi, H. Nemati, A. Panda, and P. Bhatu, "Conflvm: A compiler for enforcing data confidentiality in low-level code," in *EuroSys*, 2019.

[12] R. Branco, K. Hu, K. Sun, and H. Kawakami, "Efficient mitigation of side-channel based attacks against speculative execution processing architectures," 2019, uS Patent App. 16/023,564.

[13] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking Data on Meltdown-resistant CPUs," in *CCS*, 2019.

[14] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium*, 2019.

[15] S. A. Carr and M. Payer, "Datashield: Configurable data confidentiality and integrity," in *AsiaCCS*, 2017.

[16] C. Carruth, "RFC: Speculative Load Hardening (a Spectre variant #1 mitigation)," Mar. 2018.

[17] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution," in *EuroS&P*, 2019.

[18] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in *ISCC*, 2006.

[19] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *USENIX Security*, 2004.

[20] T. Downs, "Where do interrupts happen?" Aug. 2019. [Online]. Available: https://travisdowns.github.io/blog/2019/08/20/interrupts.html

[21] ECLYPSIUM, "System Management Mode Speculative Execution Attacks," May 2018. [Online]. Available: https://blog.eclypsium.com/2018/05/17/system-management-mode-speculative-execution-attacks/

[22] D. Evtyushkin and D. Ponomarev, "Covert channels through random number generator: Mechanisms, capacity estimation and mitigations," in *CCS*, 2016.

[23] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *MICRO*, 2016.

[24] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "BranchScope: A New Side-Channel Attack on Directional Branch Predictor," in *ASPLOS*, 2018.

[25] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers," 2016.

[26] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware," *Journal of Cryptographic Engineering*, 2016.

[27] A. F. Glew and G. J. Hinton, "Method and apparatus for processing memory-type information within a microprocessor," 1995, european Patent Office EP0783735A4.

[28] B. Gregg, "KPTI/KAISER Meltdown Initial Performance Regressions," 2018.

[29] D. Gruss, D. Hansen, and B. Gregg, "Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer," *USENIX ;login*, 2018.

[30] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *ESSoS*, 2017.

[31] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA*, 2016.

[32] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[33] D. Gruss, M. Schwarz, and M. Lipp, "Meltdown: Basics, Details, Consequences," *BlackHat USA*, 2018.

[34] M. Guri, M. Monitz, Y. Mirski, and Y. Elovici, "Bitwhisper: Covert signaling channel between air-gapped computers using thermal manipulations," in *IEEE CSF*, 2015.

[35] J. Horn, "speculative execution, variant 4: speculative store bypass," 2018.

[36] Intel, "Intel Analysis of Speculative Execution Side Channels," Jul. 2018. [Online]. Available: https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf

[37] ——, "Retpoline: A Branch Target Injection Mitigation," Jun. 2018, revision 003.

[38] ——, "Speculative Execution Side Channel Mitigations," 2018, revision 3.0.

[39] ——, "Deep Dive: CPUID Enumeration and Architectural MSRs," May 2019. [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-cpuid-enumeration-and-architectural-msrs#MDS-CPUID

[40] ——, "Deep Dive: Intel Analysis of Microarchitectural Data Sampling," 2019.

[41] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," 2019.

[42] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," 2019.

[43] A. Ionescu, "Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER)." 2017. [Online]. Available: https://twitter.com/aionescu/status/930412525111296000

[44] G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cross processor cache attacks," in *AsiaCCS*, 2016.

[45] J. Jaeyeon and Y. Zhu, "Sensitive data tracking using dynamic taint analysis," 2014. [Online]. Available: https://patents.google.com/patent/US9548986B2

[46] kernel.org, "Documentation: Document array_index_nospec - kernel version v4.16-rc1," 2018. [Online]. Available: https://www.kernel.org/doc/Documentation/speculation.txt

[47] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation," in *DAC*, 2019.

[48] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," *ePrint 2018/418*, May 2018.

[49] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *arXiv:1807.03757*, 2018.

[50] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[51] E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *WOOT*, 2018.

[52] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," in *OSDI*, 2014.

[53] M. Larabel, "Bisected: The Unfortunate Reason Linux 4.20 Is Running Slower," Nov. 2018.

[54] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *IEEE / ACM Inter-*

national Symposium on Code Generation and Optimization – CGO 2004, 2004, pp. 75–88.

[55] K. P. Lawton, "Bochs: A portable PC emulator for UNIX/X," *Linux Journal*, 1996.

[56] E. N. Leake and G. Pike, "Taint tracking mechanism for computer security," 2013. [Online]. Available: https://patents.google.com/patent/US8875288B2

[57] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *USENIX Security Symposium*, 2017.

[58] J. Levin, *Mac OS X and IOS Internals: To the Apple's Core*. John Wiley & Sons, 2012.

[59] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.

[60] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *S&P*, 2015.

[61] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN notices*, 2005.

[62] LWN, "The current state of kernel page-table isolation," Dec. 2017. [Online]. Available: https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/

[63] G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution Using Return Stack Buffers," in *CCS*, 2018.

[64] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. Alberto Boano, S. Mangard, and K. Römer, "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud," in *NDSS*, 2017.

[65] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv:1902.05178*, 2019.

[66] Microsoft, "Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer," Jan. 2018.

[67] M. Miller, "Mitigating speculative execution side channel hardware vulnerabilities," Mar. 2018.

[68] J. Newsome and D. X. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software." in *NDSS*, 2005.

[69] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer, "You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass," *arXiv:1805.08506*, 2018.

[70] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: the Case of AES," in *CT-RSA*, 2006.

[71] A. Pardoe, "Spectre mitigations in MSVC," 2018.

[72] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security Symposium*, 2016.

[73] F. Pizlo, "What Spectre and Meltdown mean for WebKit," Jan. 2018.

[74] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "LIFT: A low-overhead practical information flow tracking system for detecting security attacks," in *MICRO*, 2006.

[75] C. Reis, A. Moshchuk, and N. Oskov, "Site Isolation: Process Separation for Web Sites within the Browser," in *USENIX Security Symposium*, 2019.

[76] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *S&P*, 2010.

[77] M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs," *arXiv:1905.05725*, 2019.

[78] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *CCS*, 2019.

[79] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript," in *FC*, 2017.

[80] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," in *ESORICS*, 2019.

[81] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qualifiers." in *USENIX Security Symposium*, 2001.

[82] A. Slowinska and H. Bos, "Pointless tainting?: evaluating the practicality of pointer tainting," in *EuroSys*, 2009.

[83] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, 2008.

[84] K. Sun, R. Branco, and K. Hu, "A New Memory Type Against Speculative Side Channel Attacks," 2019.

[85] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *ASPLOS*, 2019.

[86] The Chromium Projects, "Actions required to mitigate Speculative Side-Channel Attack techniques," 2018.

[87] V. Tkachenko, "20-30% Performance Hit from the Spectre Bug Fix on Ubuntu," Jan. 2018.

[88] C. Trippel, D. Lustig, and M. Martonosi, "MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols," *arXiv:1802.03802*, 2018.

[89] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," 2018. [Online]. Available: https://support.google.com/faqs/answer/7625886

[90] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security Symposium*, 2018.

[91] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-flight Data Load," in *S&P*, May 2019.

[92] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *IEEE HPCA*, 2008.

[93] L. Wagner, "Mitigations landing for new class of timing attack," Jan. 2018.

[94] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Still: Exploit code detection via static taint and initialization analyses," in *Annual Computer Security Applications Conference*, 2008.

[95] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *S&P*, 2015.

[96] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution," 2018.

[97] Z. Wu, Z. Xu, and H. Wang, "Whispers in the Hyper-space: High-bandwidth and Reliable Covert Channel Attacks inside the Cloud," *IEEE/ACM Transactions on Networking*, 2014.

[98] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, "An exploration of L2 cache covert channels in virtualized environments," in *CCSW'11*, 2011.

[99] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *MICRO*, 2018.

[100] Y. Yarom and K. Falkner, "Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *USENIX Security Symposium*, 2014.

[101] J. Yu, L. Hsiung, M. El Hajj, and C. W. Fletcher, "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing," in *NDSS*, 2019.