WHISTLE: CPU Abstractions for Hardware and Software Memory Safety Invariants

Sungkeun Kim, Farabi Mahmud, *Student Member, IEEE,* Jiayi Huang, *Member, IEEE,* Pritam Majumder, *Student Member, IEEE,* Chia-Che Tsai, Abdullah Muzahid, Eun Jung Kim, *Member, IEEE*

Abstract—Memory safety invariants extracted from a program can help defend and detect against both software and hardware memory violations. For instance, by allowing only specific instructions to access certain memory locations, system can detect out-of-bound or illegal pointer dereferences that lead to correctness and security issues. In this paper, we propose CPU abstractions, called WHISTLE, to specify and check program invariants to provide defense mechanism against both software and hardware memory violations at runtime. WHISTLE ensures that the invariants must be satisfied at every memory accesses. We present a fast invariant address translation and retrieval scheme using a specialized cache. It stores and checks invariants related to global, stack and heap objects. The invariant checks can be performed synchronously or asynchronously. WHISTLE uses synchronous checking for high security-critical programs, while others are protected by asynchronous checking. A fast exception is proposed to alert any violations as soon as possible in order to close the gap for transient attacks. Our evaluation shows that WHISTLE can detect both software and hardware, spatial and temporal memory violations. WHISTLE incurs 53% overhead when checking synchronously, or 13% overhead when checking asynchronously.

Index Terms—Hardware Defense, Hardware-Assisted Security, Memory Safety, Program Invariants, Cache Architecture

✦

1 INTRODUCTION

Memory safety violation is considered one of the most critical software vulnerabilities leading to both correctness and security problems. In 2020, Common Weakness Enumeration (CWE) community listed three types of memory safety violation among the five most impactful and serious software issues [1]. Memory safety violation can manifest from software or hardware behavior. For illustration purpose, let us consider the examples in Figure 1. It shows how a memory safety violation (buffer overflow) can be the result of a vulnerability in software or in hardware. A software (or *software-induced*) memory safety violation can be prevented by software defenses such as bounds-checking [2]. However, a hardware (or *hardware-induced*) memory safety violation can bypass such defenses in software within the CPU pipeline as a result of misprediction or out-of-order optimizations. For example, a Spectre-PHT attack [3] will mistrain the branch predictors to temporarily bypass the bounds-checking in software within speculative execution, and then leak the out-of-bound data through a side channel. Many works either in software or hardware have addressed memory safety violations. However, existing works suffer from the following major shortcomings:

- Most existing works have focused on defense against only software memory safety violations [4], [5], [6], or defense against only hardware memory safety violations [7], [8], or defense against software violations with partial defense against hardware violations [9], [10]. Cryptographic Capa-
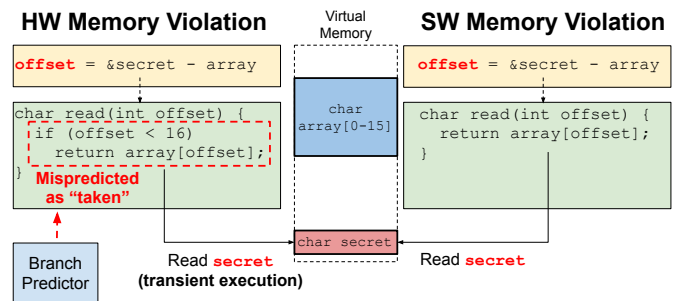

Fig. 1: A comparison of hardware and software memory safety violations. Due to a branch misprediction or lack of bounds check, respectively, a malicious input (`int offset`) can cause memory safety violations in hardware or in software.

bility Computing (C^3) [11] provides uniform defense against both software and hardware memory safety violations, yet it requires memory encryption which may not be necessary in some scenarios. Both No-FAT [12] and HeapCheck [13] performs bounds checking on both non-speculative and speculative memory access and raise exceptions on violations. An uniform, general hardware defense against both software and hardware violations not only incurs lower access overheads but also provides economy of mechanisms and wide coverage of defense.

- Existing works have addressed software memory safety violations based on either blocking the malicious behaviors (i.e., *blocklisting*) [9], [14] or allowing the benign behaviors (i.e., *allowlisting*) [5], [15]. However, for hardware memory safety violations, most existing defense works only focus on detecting or preventing specific malicious behaviors [9] or their consequences [16]. These defenses for hardware violations

- *All authors except Jiayi Huang are associated with Department of Computer Science and Engineering, Texas A&M University, Texas, TX-77843. E-mail: (ksungkeun84, farabi, pritam2309, chiache, Abdullah.Muzahid, ejkim)@tamu.edu*

- *Jiayi Huang is associated with Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106 USA. E-mail: jyhuang@ucsb.edu*

are specific to the exploits and can be considered *ad-hoc* solutions. If any future exploit exhibits different behaviors, the attacker can circumvent the defense mechanism.

To address the limitations, we propose WHISTLE, a set of CPU abstractions for memory safety violation detection inside the microarchitecture. It is capable of handling both software and hardware violations and allows program-specific policies to check synchronously or asynchronously. WHISTLE provides schemes to detect violations in stacks, heaps, and global objects of a program, and can prevent spatial attacks as well as temporal attacks such as *Use-After-Free*. WHISTLE is based on program-specific invariants stemming from a common pattern that *most of the memory locations of a program are accessed (transiently or not) by only a handful of instructions during normal executions*. These "good" instructions can be formulated for the corresponding memory locations as *invariants* of the program. By allowing only memory accesses within the invariants, WHISTLE can defend against future, unknown software or hardware vulnerabilities as exploiting these vulnerabilities will trigger the alarms by accessing disallowed memory locations. We propose a hardware implementation of WHISTLE, with the following contributions:

- *Uniform Defense:* We develop an effective defense mechanism against both software and hardware memory violations using invariants. As a proof-of-concept (PoC), we demonstrate how to generate invariants using profiling in hardware and store them in program binaries.
- *Invariant Cache with Flexible Checks:* We propose a small dedicated cache, namely Top Invariant (TI) cache, to make the invariant accesses faster. TI cache works alongside the L1 cache with the rest of the memory hierarchy. During a load memory request, WHISTLE checks if it is accessing a location protected by the invariants. If so, WHISTLE accesses the TI cache to check whether the invariants are satisfied. Memory accesses to invariants are distributed along the memory hierarchy based on the access frequency. Thus, the most frequently accessed invariants reside in the TI cache, while others reside in the L2, memory, or disk. TI cache along with the rest of the memory hierarchy provides the functionality to check if the invariants are satisfied. WHISTLE provides two modes of invariant checking — synchronous and asynchronous. WHISTLE uses synchronous checking for high security-critical programs, while others are protected by asynchronous checking.
- *A Fast Exception*: When a memory location is accessed by any instruction outside the invariants, WHISTLE raises the security exception immediately (i.e., without waiting for the offending instruction to reach the head of the reorder buffer) to prevent Meltdown-type [17] attacks. The OS handles the exception by immediately terminating the process.

We implement WHISTLE in `gem5` [18] and evaluate it with SW and HW violations by using four programs of Spectre variants Spectre (PHT/BTB/RSB/STL) [19], BugBench [20] and NIST [21]. We also evaluate overheads of WHISTLE using SPEC CPU2017 [22]. A thorough security and performance analysis shows that WHISTLE can detect both HW and SW memory safety violations with 13%-53% performance overhead across a mix of synchronous and asynchronous checks.

The rest of the paper is organized as follows: §2 provides a background, §3 describes the threat model; §4 discusses security analysis; §5 presents the main ideas of WHISTLE; §6 shows the detailed implementation; §7 and §8 evaluate WHISTLE's security

and overhead; §9 provides some related work, and finally, §10 concludes the work.

2 BACKGROUND & MOTIVATION

In this section, we introduce the basic concept of allowlisting and blocklisting as the strategy of defense and our approach toward hardware-based solution.

2.1 Allowlisting or Blocklisting?

All access control mechanisms can be categorized as either *allowlisting* and *blocklisting*. *Allowlisting* defines the policies based on the "known good" behaviors of the target, and block everything else by considering them potentially harmful. On the other hand, *blocklisting* defines the policies based on the "known bad" behaviors of the target and explicitly blocks them in the system. Take memory safety for an example. An allowlisting approach adds disjoint [5], [15], [23], [24] or co-joint metadata [25] to keep track of the memory locations which can be safely accessed. A blocklisting approach may trigger alarm from generated token or tripwire [9], [14], or detect memory content corruption. Both approaches have pros and cons. Allowlisting systematically defends against a class of attacks, even if the attack factors are unknown. Blocklisting blocks a known threat until the threat is removed systematically, and thus can't mitigate unknown threats.

2.2 Why Allowlisting in WHISTLE?

A lesson from the recent discoveries of hardware and software vulnerabilities is that *anything that can go wrong will go wrong.* We cannot assume even the hardware to be immune from the classes of vulnerabilities previously found inside software. In 2018–2019, numerous variants of the Spectre and Meltdown attacks were discovered. In 2020, CWE reported more than 2000 memory safety violations in various popular software [1]. Despite individual patches in software or hardware, no systematic solution has been proposed so far to prevent software and hardware memory safety violations as a class of vulnerabilities. Therefore, we choose to adopt allowlisting policy in WHISTLE as a stepping stone towards mitigating all of these attacks.

2.3 Invariant Generation

WHISTLE uses invariants to distinguish malicious and benign behaviors. Just like all access control mechanisms, the composition and enforcement of security rules are both sophisticated topics. Fortunately, WHISTLE's mechanism for invariant enforcement is not tied to any method of composition, and thus allows us to focus on the former and leave the latter for future work.

For now, WHISTLE uses profiling (i.e., dynamic analysis) for early, unintervened invariant generation, but *profiling is not inherent to our solution*. Nevertheless, using profiling can cause false positives and false negatives. For example, some cases of misspeculation can be potentially benign, especially for the buffer overrun that commonly happens after a program loops through the elements. Using hardware profiling can prevent the false positives by including these accesses as part of the invariants. The caveats of doing so is that any protected "secret" cannot be near the bounds of any buffer. These cases are not violation in regards to any of the invariants collected, but do pose a risk of being exploited by the attackers. In all of our program samples, we have not observed this scenario. We consider this a reasonable caveat, as a compiler

can straightforwardly distance buffers from other variables with padding. Other false positives can occur if profiling is not done sufficiently (e.g., until convergence). In that case, new accesses might appear as false violations. A programmer can analyze and confirm the false violations and subsequently update the invariant section by releasing some type of invariant patch.

A careful reader might wonder what happens if the profiling runs are buggy or under attack. In order to prevent buggy/attacked profiling runs from corrupting the invariants, the standard practice in debugging community is to use some well-known bug detection tools with each profiling run to make sure that the execution is bug-free [26]. On top of that, an isolated machine is used to prevent any attack during profiling.

Other potential alternative methods to profiling include synthesization (i.e., static analysis) and human composition. Synthesization can guarantee invariant coverage. However, it may require sophisticated algorithm, and is less scalable. Human composition requires significant efforts and is subject to human error. In practice, profiling can be useful for the initial collection of raw policies, which can be further refined with synthesization or human intervention. In software security, profiling has been used by other papers [26], [27] especially when the work is focused on expression and enforcement. There is a long line of research on improving dynamic analysis, such as symbolic execution, fuzzing [28], and AI-based collection, which we consider complementary to our work and out of scope.

3 THREAT MODEL

In-scope Attacks: WHISTLE prevents violations to memory safety rules defined by the invariants. The violations can be the results of exploiting either software or hardware vulnerabilities, including the existing Spectre attacks [3]. Besides the known attacks, WHISTLE is also designed as a defense for future, unknown attacks, including future Spectre-type attacks that exploits speculative optimizations to violate memory safety rules, as well as future Meltdown-type attacks which violates hardware protections but can still be temporarily executed in the pipeline. WHISTLE prevents memory safety violations to variables in stacks, heaps, and global regions, and prevents spatial as well as temporal violations such as Use-After-Free.

Trusted Components: WHISTLE assumes that the software is trustworthy but may contain vulnerabilities to be exploited by the attackers; both the protected program and the OS will not contain malicious code that deliberately violates the memory safety rules. WHISTLE also trusts the integrity of invariants stored in the program binaries, which can be protected by page tables or other hardware protections. The OS is also trusted to handle exceptions raised by the hardware during invariant violations.

Out-of-scope Attacks: WHISTLE only prevents violations for memory safety rules, and does not protect other data structures such as registers. WHISTLE does not enforce control flow integrity but can detect memory safety violations (such as buffer overflows) that are either prerequisites or outcomes of control flow violations. WHISTLE also cannot prevent attacks from attacker-forged code such as Javascript or eBPF gadgets, WHISTLE does not protect the correctness and integrity of memory contents and cannot prevent or detect semantics-based attacks that do not violate memory safety rules. In addition, WHISTLE does not prevent data leakage through side channels, including side channels through structures added by WHISTLE (e.g., TI Cache),

Memory Type	Effective Period	Invariants $Invrs = \{Key \rightarrow PCs\}$
Global	$Load \rightarrow Unload$	$(CallContext, Addr - BinaryBase)$ $\rightarrow \{PC_1, PC_2, \cdots, PC_n\}$
Stacks	$Call \rightarrow Return$	$(CallContext, Addr - FrameBase)$ $\rightarrow \{PC_1, PC_2, \cdots, PC_n\}$
Heaps	$Malloc \rightarrow Free$	$(CallContext_{Malloc}, Addr - ObjectBase)$ $\rightarrow \{PC_1, PC_2, \cdots, PC_n\}$

TABLE 1: The memory types, effective periods, and invariants definitions in WHISTLE.

but rather prevents illegal access to data before leaking through side channels. Although WHISTLE may introduce new side channels through its structures, the side channels do not reveal more information than what L1 or L2 cache already reveals (i.e., which memory blocks are recently accessed).

Synchronous vs. Asynchronous Checking: WHISTLE allows flexible security policies, for each program to choose between (1) blocking the memory operations until the check is finished (*synchronous checking*); and (2) letting the memory operations finish but raising an exception immediately after the violation is detected (*asynchronous checking*). A similar design choice has been adopted by REST [14], to delay STORE commits (*Debug* mode) until acknowledgement or to proceed and issue imprecise exceptions (*Secure* mode). Synchronous checking provides stronger security guarantee because there is no transient window where the CPU pipeline has access to the data and is able to leak through consequential cache operations. Synchronous checking is necessary if the attacker can retrieve the secret with one attempt, such as fetching a single bit from an encryption key. In other cases where the attacker needs several attempts or iterations, the program can be prompted by the exception as soon as the first violation is detected by WHISTLE. One example where asynchronous checking is appropriate is when the attacker is using Spectre to dump the kernel memory, which will be stopped by WHISTLE immediately.

4 DEFINITION AND SECURITY ARGUMENTS

In this section, we describe the invariants used in WHISTLE and our security arguments.

4.1 Memory Safety Invariants

WHISTLE detects memory safety violations based on program invariants. We define invariants as Program Counters (PCs) allowed to read or write a memory location in a program, either as global, stack, or heap objects. Table 1 defines the invariants of three different memory types as well as their effective periods:

1) A *global object* resides inside the program's data segment. The invariant would include all the PCs that can access the *virtual address* of that object, relative to the base of the binary. Even if the global objects have a static life time, we define the invariant of the global objects with the calling context for the finer grained protection.

2) A *stack (local) object* resides inside the stack of each thread. The invariant would include all the PCs that can access the *offset* of that current stack frame. The offset is distinguished by the *calling context* of the current frame; The calling context changes when entering a function and gets restored when returning. This is to differentiate the local objects in two functions that share the same offset.

3) A *heap object* resides inside the heap and is created by routines such as `malloc`. Since the heap can be reused,

(a) Software Violation

(b) Hardware Violation

```
1  // Global Data
2  char dict[256];
3  /* other variables */
4  int secret;
5  // Vulnerable Code
6  int input[]=/* from user */;
7  for (i=0; input[i]; i++)
8    printf("%d\n", dict[
       input[i]]);
```

```
1  // Global Data
2  char dict[256];
3  char hex[512];
4  /* other variables */
5  int secret;
6
7  // Vulnerable Code
8  int x=/* from user */;
9  if (x>=0 && x<256)
10   printf("%c%c\n", hex[
       dict[x]*2], hex[dict[x
       ]*2+1]);
```

Fig. 2: Examples of memory safety violations.

the invariants of a heap object is related to the timing of allocation and deallocation. We identify the invariant by the calling context when `malloc` is called, and unload the invariant when the object is freed.

4.2 Security Arguments

Here we describe the security argument for the defense of WHISTLE against software and hardware violations.

Detecting Software Violations: We use buffer overflow as an example (Figure 2(a)) for software violations. At line 8, the value of each element in `input` is not checked to be within the bounds of `dict`. A malicious `input` may contain values that can load beyond the bounds of `dict` and read `secret`. To detect this attack, WHISTLE must check the invariant that *line 8 should never load data beyond the bounds of* `dict`. When this invariant is generated, WHISTLE will see that the PC(s) of line 8 only access memory up to the bound of `dict` during normal and attack-free executions. Therefore, the PC(s) of line 8 will only be included in the invariants for `dict` and `input`, and will not be permitted to read `secret`.

Detecting Hardware Violations: We use Spectre (Bounds Check Bypass) as an example for hardware violations as a result of speculative execution (Figure 2(b)). The attack speculatively accesses beyond the bound of `dict` at line 10, even reaching `secret` with specific x. Based on the return value of `dict[x]`, specific elements of `hex` are loaded and create a side channel that can leak the `secret`. To detect this attack, WHISTLE must check the invariant that *line 10 should never speculatively load* `secret`, *even though it might still speculatively load beyond the bounds of* `dict`.

5 INVARIANT-BASED MONITORING

This section presents the details of different components of WHISTLE and its end-to-end workflow.

Invariant Generation: As a proof-of-concept system for invariant-based detection, we choose to use profiling for invariant generation. This choice is influenced by numerous prior works [26], [27] that show that profiling can be an effective technique to collect various types of invariants. WHISTLE extends the hardware to support in-microarchitecture event tracing for both speculative and non-speculative executions. During profiling, the PCs of all the memory instructions are recorded irrespective of their execution status (transient or not). These recorded memory locations and PCs are then processed by a software tool to generate the invariants. Each invariant is associated with a protected memory object and contains the PCs of the memory instructions that access the corresponding the memory object. WHISTLE extends
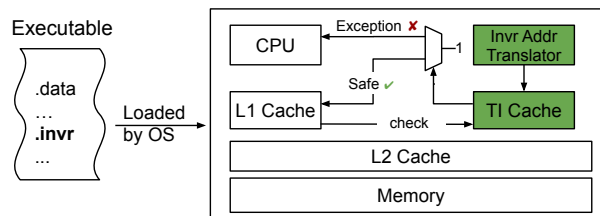

Fig. 3: Invariant-based monitoring system workflow.

the program binary with a special invariant section, to store the invariants for global, heap, and stack objects.

Invariant Cache: We propose a fast address translation mechanism to obtain the invariant addresses. Invariant information is stored in a fully associative shadow cache structure, named Top Invariant (TI) cache, alongside the L1 data cache. It is introduced to avoid any interference with the demand data. The TI cache stores PCs of the most frequent instructions of the invariant sets, while other less frequent PCs of the invariant sets reside in L2 cache or memory. Frequency of accesses is collected during profiling. We provide both synchronous and asynchronous modes for invariant checking operations. The synchronous operation checks invariants before the memory content is accessed, thereby providing the highest security level. Thus, synchronous checking is suitable for preventing attacks that can cause leakage/damage in a single attempt. On the other hand, the asynchronous mode can carry out invariant checking lazily off the critical path without blocking the load instruction execution. Thus, asynchronous checking is suitable for applications with less stringent security requirements (e.g., less sensitive data or cases where multiple attacks are needed for practical purposes). The detailed implementation of these designs is presented in §6.

Fast Security Exceptions: To handle security violations appropriately, we introduce a faster exception mechanism. WHISTLE raises this exception when a memory location is accessed by any instruction outside the invariant set. We argue that having a specialized and fast exception is crucial for security. This exception should be raised as soon as possible in the CPU pipeline. In other words, the pipeline should deliver the exception even before the offending instruction becomes the head of the reorder buffer. Thus, the window for launching a meltdown-type attack will become smaller. Finally, the OS should immediately terminate the program and report the violating PC and the accessed memory location (even though the instruction may or may not be squashed by the CPU). Thanks to the invariant information, programmer can easily reason about the violation and take appropriate remedy.

End-to-end: Figure 3 shows the end-to-end workflow for WHISTLE. A program is compiled and executed with trusted inputs and environments for hardware-based profiling. Once profiling finishes, the collected microarchitecture events are further processed in software to generate the invariant sets. The program binary is augmented with an invariant section that stores the invariant sets. During subsequent executions, the operating system loads the binary, reads the binary header, and initializes global invariant registers with range information. When a memory location is accessed by a load instruction, WHISTLE compares the virtual address of the location with the invariant registers to determine whether an invariant check is needed. If the check is not needed, the data is accessed as normal. Otherwise, WHISTLE feeds the memory address to the fast invariant address translator
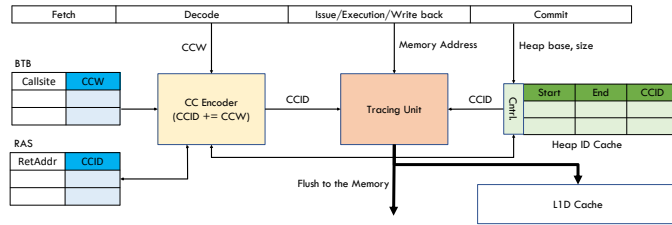
Fig. 4: Out of order core with additional hardware for profiling, maintaining calling contexts, and heap information. CCID= Calling Contex ID and CCW = Calling Context Weight.

Fig. 5: Convergence of invariants. Invariants are saturated as more profiles are collected.

to generate the invariant pointer address. WHISTLE uses this pointer address to access the TI cache and check the invariants. In cases of TI cache misses, the check request is sent to the next level in the memory hierarchy for further checks. Note that the invariant check can be either synchronous or asynchronous with the data access depending on the application security level. In synchronous check, data is not returned back to core until the check is completed. When asynchronous check is applied, data can be returned right away to minimize the performance overhead. An exception is raised if there is any violation. The exception is raised right away (without waiting for the instruction to be at the head of the reorder buffer) and returns the control to the operating system. The exception handler terminates the process immediately and reports the instruction and memory address.

6 IMPLEMENTATION

6.1 Overview of the Design

WHISTLE adds extra hardware to enable hardware-based profiling. Figure 4 shows a typical out-of-order core with the extra hardware. §6.2 explains the tracing support for profiling. WHISTLE profiles three major memory areas used in a program - global, stack, and heap. Owing to different characteristics of distinct memory allocations, it is imperative to profile and record them separately. To profile data objects, we leverage the calling contexts to differentiate accesses to the same address (§6.3). Unlike stack, a conventional CPU is not aware of heap object. We extends CPU and OS to keep track of heap allocations and deallocations (§6.4). Once profiling is finished, invariants are generated and embedded into the executable binary offline (§6.6). To implement invariant checks efficiently, we use a specialized cache like structure, namely, TI Cache (§6.8).

6.2 Profiling Support

We augment the out-of-order core with a tracing unit (Figure 4). Special memory regions are allocated in each core to record the traces. These accesses can bypass the caches and there is no need to check its coherence and consistency during profiling as each core has its private profiler. For multithreaded programs, all the per-core profiles are merged (offline) into one profile. To be more specific, the timings of allocating and deallocating a heap object will be recorded to be associated with the accesses to the virtual address of the object during this period of time.

As explained in §5, WHISTLE uses multiple bug/attack-free inputs to collect invariants. WHISTLE collects invariants until no new invariants are found by profiling more. Figure 5 shows the convergence of invariants over profiling runs. In this figure, we profile with different inputs given by SPEC CPU2017 [22] and randomly generated for real applications respectively. As profiling
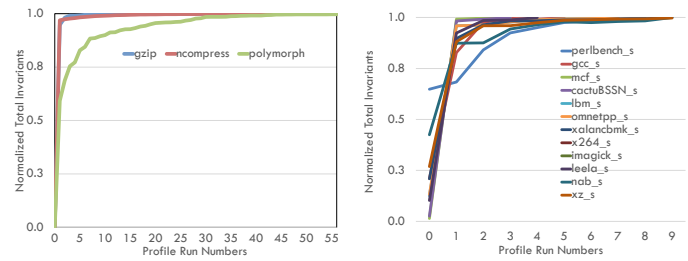
more, increment of total invariants are saturated. This clearly demonstrates the convergence of invariants. Of course, there is still no guarantee that all possible invariants are captured. Therefore, as suggested in prior works [26], coverage enhancing techniques [28] need to be applied during profiling runs.

6.3 Calling Context Encoding

Memory addresses in stack/heap are reused by many objects as program runs. To differentiate these objects, we use a calling context that is a sequence of call sites. Previous works [29] use xor-folding of the last few call-sites, but PCCE [30] proposed to efficiently encode/decode precise calling context. PCCE calculates weights (CCW) of edges on a call-graph. Then, "CCID += CCW" and "CCID -= CCW" instructions where CCID is a calling context ID initialized with zero are inserted before/after every call instruction. WHISTLE adopts this idea to maintain CCID at runtime. Instead of inserting two extra instructions, we extend call/return instructions of Intel x86 ISA to deliver CCW to the processor and extend the processor to update CCID with very simple logic (add/subtract). Updating CCID is not dependent on existing design in the processor pipeline stages. Branch Target Buffer (BTB) and Return Address Stack (RAS) are extended to store CCW or CCID (64 bits each) associated with a predicted call (Figure 4).

6.4 Hardware Support for Heap Objects

We extend the compiler, instruction set architecture, and the processor to trace heap object's creation and deletion. Compiler inserts the extended instruction to *malloc* and *free* function so that the processor updates the *Heap ID* cache as shown in Figure 4. *Heap ID* cache is Content Addressable Memory (CAM) and uses two tags–begin and end address of heap objects. The data portion of the cache stores the allocation CCID of heap objects. During an access to a heap object, tag matching is done by checking if the heap address lies in between the two tags. If so, the CCID of the matched line is returned as data. In case the cache does not have available space, requests are issued to extended memory controller that manages designated memory to keep information of additional heap objects [1]. Energy and space overheads are evaluated in § 8.

The current *Heap ID* cache is shared among all the cores to ensure coherence. This is crucial since temporal memory violations can occur across multiple cores. For example, core 1 can deallocate an object while core 2 continues to access the object. Therefore, as soon as a heap ID is unloaded in the *Heap ID* cache, the object is considered invalidated by all the cores.

---

1. In the evaluation, we limit the area of heap profile to store every heap information in *Heap ID* cache.

6.5 Binary Compatibility

The extension of WHISTLE for the compiler, the ISA, and the processor does not break existing applications that do not provide their program invariants. For programs that are augmented for memory safety violation detection, no other instruction needs to be modified besides only two special instructions added— call_cc/ret_cc for delivering weights for updating the CCID, and add_heapobj/remove_heapobj for updating the heap ID for tracking heap objects at *malloc* and *free*. The extension to *malloc* and *free* should only impact the system library that implements these functions (e.g., *libc*), unless the application binary is statically linked against the library. Although WHISTLE does require recompilation of the program binary, the recompilation is mostly only for the purpose of embedding the weights for CCIDs. The program sections for storing the invariant sets are injected directly into the program binary without the need of recompilation and can even be populated into a separated binary if necessary.

Portability to Other Platforms: Our extension for the compiler and the ISA is general enough to be ported to other CPU and microarchitecture with minor adjustments. For an ISA with fixed-length instructions (such as ARM), we can add a new instruction for embedding the weights for CCIDs instead of extending call/return. The extension is also neutral to microarchitectural design since it only requires CPU changes. For other compilation frameworks, such as a runtime for an interpreted language or a runtime with just-in-time compilation, invariant collection with profiling may not be possible, so we will have to rely on static analysis or programming APIs.

6.6 Invariant Section and Memory Hierarchy

WHISTLE extends the binary with a new section for invariants (.invr) as shown in Figure 6. Invariant section has two subsections—one for invariant blocks (GInvrBlks, SInvrBlks, HInvrBlks) and one for invariant pointers (GInvrPtrs, SInvrPtrs, HInvrPtrs).

Invariant Blocks store sets of PCs that access to the same memory address in the same context. These PCs will be used to check if a requested memory access by a PC is legitimate. First two 8 bytes in an invariant block are reserved to store access frequency of the memory address and the number of cache blocks to store the entire invariant block. Access frequency is used for replacement policy of TI cache and the number of cache blocks are used to multicast requests from TI cache. PCs in each invariant block are ordered by access frequency of each PC so that the most frequently used PCs are installed in TI cache. TI cache is a shadow cache structure used to reduce performance impact of invariant checks. It stores the most frequently used PCs in an invariant block as a Top Invariant Block as described in §6.8.

Invariant Pointers store addresses of the corresponding invariant blocks. WHISTLE uses indirect addressing to reduce fragmentation of invariant section. Note that size of an invariant block is not fixed and dependent on the number of PCs that access to the same memory address. To access an invariant block directly, the size of the invariant blocks should be uniform resulting in an internal fragmentation.

6.7 Invariant Access

We store invariant blocks in the separated section of the binary. One of the challenges of invariant memory management is to determine the address of an invariant block for a particular
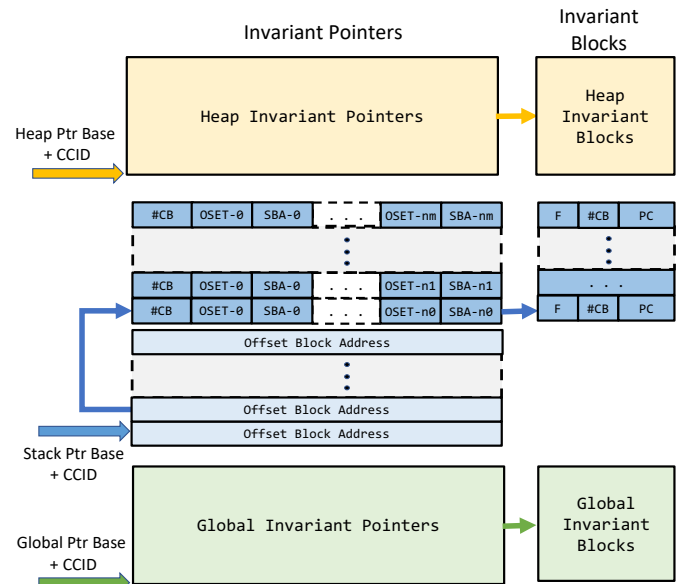

Fig. 6: Memory layout of invariant section (.invr) in executable. F=the access frequency of invariant block, #CB=the number of cache blocks, OSET=offset of stack or heap object, SBA=Stack Invariant Address, HBA=Heap Invariant Address.
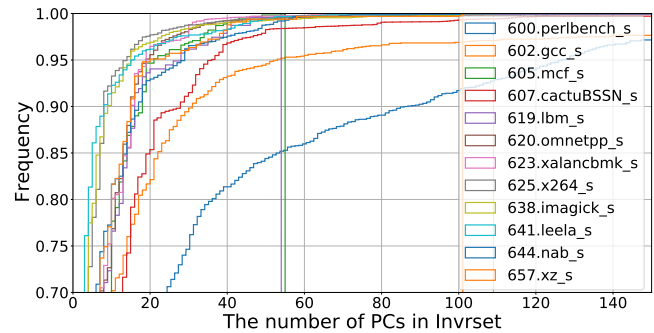

Fig. 7: Cumulative distribution of Invariant set size.

memory location. We propose efficient indirect invariant access mechanism for global, heap, and stack objects.

We describe the indirect invariant access mechanism with stack invariants for example. Stack invariants are grouped according to CCIDs. During the access of a stack object, WHISTLE uses CCID of the corresponding function as an offset from Stack Ptr Base to find Offset Block Address as shown in Figure 6. Offset Block Address points to a region that contains SInvrBlk Addrs for all offsets associated with the particular CCID. WHISTLE reads the first block of this region to determine the number of cache blocks. TI cache issues read request to all of those blocks. Each cache block contains a number of $< offset, SBA >$. As each cache blocks arrives to the TI cache, it finds the block with an offset that matches the offset of the stack object. The SInvrBlk Addr associated with this offset is used to find the invariant block of the stack object. Invariants of global and heap objects are identified in a similar fashion except that the CCIDs used for heap objects will be the allocation CCIDs. Note that only for Heap objects, the *Heap ID* cache is used to find the allocation CCIDs, whose hardware design presented in §6.4.

6.8 Top Invariant Cache

The major challenge of invariant based approach is the volume of the profiled invariants. It not only causes huge storage overhead,
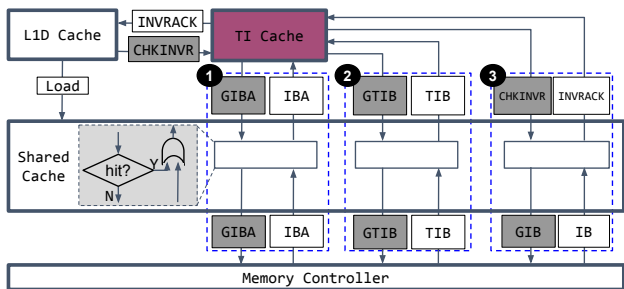
Fig. 8: Message flow of Invariant check across the memory hierarchy. GIBA/IBA, GTIB/TIB/TIB/IB, and CHKINVR/INVR(N)ACK are request/response messages for invariant block address, top invariant blocks, remaining invariant blocks, and invariant checks, respectively.
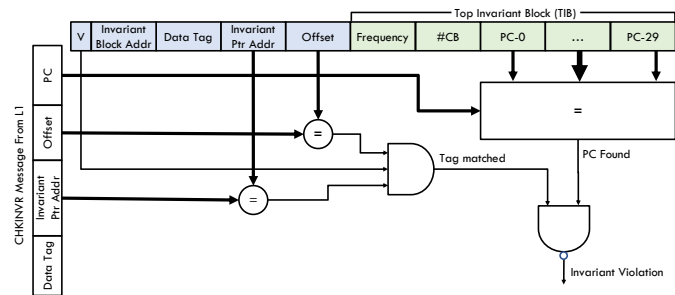

Fig. 9: Tag matching mechanism between invariant pointers and offsets, and invariant checking mechanism inside TI cache. #CB: the number of conventional cache blocks for the entire invariant sets. In this work, TI cache installs 256 bytes of them (four 64 bytes conventional cache blocks).

but also incurs performance overhead. Since caching invariants in the conventional data cache may pollute by evicting actual demand data, we introduce a special cache with a separate cache controller, Top Invariant (TI) Cache, for caching and checking the invariants.

6.8.1 Top Invariant Block and Least Frequently Accessed Replacement Policy

The numbers of PCs in each invariant sets are different and the sets have different access frequency. We first survey the range of invariant set sizes and decide size of TI cache block. Figure 7 shows cumulative distribution of invariant set size. We observe that 90% of invariant sets have less than 32 PCs which can be stored in four conventional cache block size (64 bytes). Therefore, we configure TI block size as 256 bytes. To read all 256 bytes effectively, the invariant section is generated with invariant blocks that are at least 256 byte long and aligned to conventional cache block size. Upon an invariant check, TI cache loads the first four conventional cache blocks in the invariant block. Then, it merges and installs them in one TI cache block. Remaining part of the invariant block will be installed in a shared cache and checked by CHKINVR and GIB messages described in the following Section. Second, we leverage the knowledge during profile for the efficient placement of PCs in the invariant block and TI cache replacement policy. By placing the most frequently accessed PCs first in the invariant block, the hit rate of TI cache block increases. In addition, TI cache selects a victim block that is the least frequently accessed among cache blocks for replacement.

6.8.2 Indirect Tag for TI Cache Access

Conventionally, a cache tag is part of the address for the cache line. Global, stack, and heap objects are associated with their unique invariant pointer addresses. Instead of using the conventional tag, TI cache uses the invariant pointer address as the tag as shown in Figure 9. Note that since stack and heap objects are associated with CCID and offset (see Figure 6), both Base + CCID and Offset are used for tagging. Also, TI cache adds one extra metadata to store address of invariant block (InvrBlkAddr).

6.8.3 Message Flows of Invariant Check

Figure 8 demonstrates the interaction in the memory hierarchy involved in invariant checks. Several messages are introduced to handle invariant check. Upon a memory access to a protected data in L1 cache, a CHKINVR message carrying the instruction PC and the invariant pointer address is sent to TI cache. The invariant pointer address is calculated with CCIDs. If it is a miss, TI cache initiates a sequence of steps to load top invariant block as follows: ❶ A GIBA request with the invariant pointer address is issued to get the actual address of the invariant block. The address of global invariant block is retrieved with one request but the addresses of stack and heap invariant blocks are be retrieved with at least two requests. ❷ Then the returned invariant block address is encapsulated in the GTIB messages to fetch the Top Invariant Block (TIB) from the shared cache. Note that size of TIB could be bigger than conventional cache line size (i.e., 64 bytes) depending on the configuration. In that case, TI cache loads multiple cache lines to install the entire TIB. If it hits in the shared cache, the TIB is returned and installed in the TI cache. In case of a miss, the request is forwarded to the memory controller to load it from memory. After TIB is installed in TI cache, a check is done to inspect if the accessed PC is in the block. If it is in the block, a INVRACK is sent back to L1 cache to acknowledge the safety of the access. If it is not in the block, further inspection is initiated. ❸ When the PC is not in TIB, a CHKINVR request is forwarded to the shared cache to scrutinize the remaining invariant blocks. If they miss in the shared cache, a GIB request is generated to load them to the shared cache to finish the check. After inspection, an INVRACK or INVRNACK is replied to the TI cache depending on the success of the check. If a violation happens, the INVRNACK triggers a security exception. If the type of CHKINVR is synchronous, the data supply to CPU from L1 is delayed until INVRACK. Otherwise, the data is supplied to CPU immediately and CHKINVR inspects in parallel.

7 SECURITY EVALUATION

We implement the hardware supported invariant profile and check using the gem5 simulator [18]. Table 2 summarizes the baseline configuration and additional structure in microarchitecture. WHISTLE uses TI and Heap ID cache structure to hold the invariants and CCID of heap creation on the core side. Also, WHISTLE extends branch target buffer (BTB) and return stack buffer (RSB) to store CCW and CCID. All the invariants are profiled based on cache line granularity. To profile invariants of each benchmark until no more invariants are found, we use all the inputs given by SPEC2017 [22], downloaded extra input data from online source [31], and changed the input parameters until no more invariants are found.

To emulate invariant embedding, we modify the source code of target benchmarks to allocate additional global memory to hold the invariant section. We extract information from the binaries

Parameter	Value
Core	2.0 GHz, Out-of-Order, no SMT, 32 Load Queue, 32 Store Queue entries, 192 ROB entries, Tournament branch predictor, 4096 BTB entries, 16 RSB entries.
L1-I $	Private, 64B line, 4-way, 32KB, 1 cycle access lat.
L1-D $	Private, 64B line, 8-way, 64KB for Baseline, 32KB for WHISTLE 1 cycle access lat.
HeapID $	8 B line, 1024 entries, 1 cycle access lat. Fully associative.
TI $	256B line, 256 blocks, 32KB or 64KB, 1 cycle access lat. Least Frequently Used (LFU) replacement policy, fully associative.
L2 $	Shared, inclusive, 64B line, 2 cycles access lat. 2MB, 16-way.
DRAM	Built-in memory model in gem5.

TABLE 2: Parameters of the simulated architecture. HeapID Cache and TI Cache are not included in baseline system. 64KB size of TI cache used for the fully synchronous check and 32KB size of that used for the fully asynchronous check.

	Source	Application	BOGO	IS	WHISTLE
Spatial	BugBench [20]	gzip1.2.4	✓	✗	✓
		ncompress	✓	✗	✓
		man1.5h1	✓	✗	✓
		polymorph-0.4.0	✓	✗	✓
Temporal	NIST [21]	ID 102226	✓	✗	✓
		ID 102247	✓	✗	✓
		ID 102618	✓	✗	✓
		ID 2151	✓	✗	✓
Transient	Spectre [19]	Spectre-PHT	✗	✓	✓
		Spectre-BTB	✗	✓	◑
		Spectre-RSB	✗	✓	✓
		Spectre-STL	✗	✓	✓

TABLE 3: Evaluation results with spatial, temporal, and transient memory violations in BOGO [5], InvisiSpec(IS) [16], and WHISTLE validation. ✓ means that the violation is detected. ✗ means that the violation is not detected. ◑ means that the violation is circumstantially detected, since WHISTLE only detects Spectre-BTB when there is a preceding memory corruption to mistrain the BTB.

(ELF format) such as regions of data segments (.data, .rodata, and .bss section) and code segment (.text) as well as addresses of malloc and free functions. Then, these binary layout information is referred by gem5 during simulation. This enables us to simply reflect extensions to compiler and operating systems.

We evaluate WHISTLE for both HW and SW violations. We write four programs of Spectre variants (Spectre-PHT/BTB/RSB/STL) [19] with eviction based cache side-channel to evaluate HW violations and use BugBench [20] and test cases from NIST [21] for SW violations. After profiling with bug-free inputs, the test programs are executed again with bug-triggering inputs. We also run SPEC CPU2017 [22] for both security and overhead evaluation. We use the *reference* input size and simulate for 1 billion instructions after warming up microarchitecture states with 1 billion instructions in system-call emulation mode[2].

7.1 SW and HW Violations

Table 3 lists the applications and validation results. BugBench provides simplified real-world applications (gzip, man, ncompress,

---

2. System call emulation has one-to-one page mapping and requires no TLB translation. Also, invariants are stored continuously in virtual and physical spaces, and invariant address are directly translated using offsets and CCIDs. We envision that both the program data and invariant addresses should be translated with page tables managed by OS, and CPU will perform TLB lookup for both. The existing TLB and Page Miss Handler can be reused for invariant addresses, with potentially larger buffer to reduce the overhead. Due to simulation limitations and significant workload for implementing OS-level handler, we leave this experiment for future work.

and polymorph) with buffer overflow bugs in the stack and global objects, and NIST provides test cases to evaluate the Use-After-Free bugs in heap objects. Buffer overflow bugs are detected by WHISTLE and it also detects the Use-After-Free bugs because WHISTLE keeps track of allocation/deallocation of heap objects using Heap ID cache. We do not observe false positives.

WHISTLE can fully detect three out of four Spectre variants. First, in Spectre-PHT [3], transient instructions are exploited to access a secret using an array out-of-bound access. Since this access was not observed during profiling, WHISTLE raises an exception and the program stopped. Second, Spectre-RSB exploits Return Stack Buffer to hijack return flow. PoC program mimics the attacker's behavior using a gadget function and malicious code resides after the gadget function call. Gadget function is invoked only during the attack and WHISTLE detects the violation from the malicious code. Three, Spectre-STL exploits memory disambiguator. PoC program inserts malicious load instruction after naive store instruction clearing secret data so that the load instruction reads the secret before clearing it. Again, this malicious load did not appear in the profile and WHISTLE detects this variant as well.

The only exception is Spectre-BTB, which WHISTLE can only detect under specific circumstances. Spectre-BTB, unlike other Spectre variants, exploits control flow violations instead of data access violations. Since WHISTLE does not check instruction fetching, it cannot detect control flow violations. However, to cause Spectre-BTB, the attacker needs to mistrain the BTB in order to change the control flow. The attacker may use a buffer overflow to corrupt a code pointer or return address, which can be detected by WHISTLE. WHISTLE cannot detect Spectre-BTB if the attacker uses other mistraining methods, such as mistraining from another thread. Potentially, WHISTLE can extend the invariant profiling and checking to instruction cache. That way, WHISTLE will be able to detect control flow violations that cause invariant violations in the instruction cache. We leave this extension for future work.

7.2 Comparison against with BOGO and InvisiSpec

We run PoC programs for both BOGO [5] and InvisiSpec [16] that are SW and HW memory violation detection techniques respectively. As shown in Table 3, neither BOGO nor InvisiSpec detect all the violations. BOGO provides full memory safety on top of MPX-enabled [15] processors, but it is limited to committed load or store instructions resulting in failure to detect the transient attacks. InvisiSpec defends against the transient attacks by blocking cache side channels. However, InvisiSpec is not designed to defend SW violations. We discuss more related works in §9.

7.3 Coverage of HW vs. SW Profiler

We implement both HW and SW profilers, and evaluate the coverage of using gem5 simulator with out-of-order core. HW profiler records every memory access, either transient or non-transient. We profile the first billion instructions for collecting the calling contexts, and the second billion instructions for collecting both the calling contexts and the invariants. On the other hand, SW profiler records only committed memory accesses, which can miss hardware vulnerabilities that rely on transient executions, such as recent speculation-based attacks, Spectre and Meltdown. Figure 10 shows the coverage of HW profiler in terms of number of invariant sets compared to SW profiler. HW profiler covers 60%
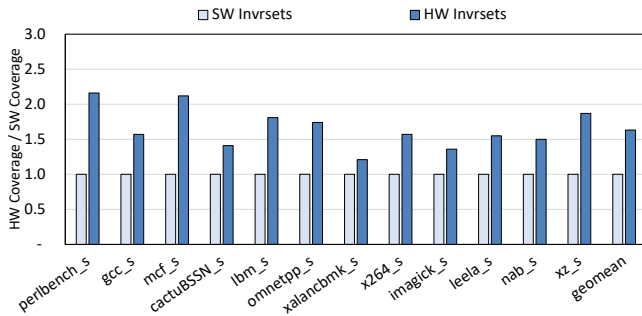
Fig. 10: Coverage Comparison in terms of the total number of invariant sets between SW and HW Profilers.

Fig. 11: Reduction of security exception latency in asynchronous check.

more invariant sets. *perlbench_s* and *mcf_s* have higher coverage than other benchmarks. The number of squashed memory instruction is dependent on program characteristics, such as number of branches, indirect jumps, and/or HW components associated with speculative execution, such as branch predictor.

7.4 Attack Surface Reduction

We also measure the reduction of attack surface in terms of software and hardware memory safety violation, based on how many rogue memory accesses in a program will be accepted by the system. Here, we define the memory-specific attack surface as *the number of PCs allowed to access a specific memory location under a specific calling context*. In TABLE 4, we show that for each invariant set, there are 2.60–20.01 PCs in average allowed to access the memory. However, without WHISTLE, a Spectre-BTB attack can change the control flow speculatively to allow any memory accessing PC to read/write any memory location. Considering that for each program in CPU2017, there are at least 1,220–25,405 unique PCs during the profile that access memory, the attack surface reduction by WHISTLE is 99.80–99.99%.

Benchmarks	# Unique PCs	# Invariant Sets	Avg. # PCs / Inv. Set	Attack Surface Reduction
perlbench_s	18,482	6,685	14.83	99.98%
gcc_s	16,975	2,671	20.01	99.96%
mcf_s	1,258	1,234	5.27	99.91%
cactuBSSN_s	25,405	298,649	6.54	99.99%
lbm_s	1220	517	4.76	99.80%
omnetpp_s	7,934	21,605	5.00	99.99%
xalancbmk_s	4,326	16,139	4.23	99.99%
x264_s	3,827	4,200	3.32	99.97%
imagick_s	2,865	7,819	2.60	99.98%
leela_s	2,678	5,755	2.79	99.98%
nab_s	2,515	5,118	4.93	99.98%
xz_s	1,305	897	3.81	99.88%

TABLE 4: Assessment of attack surface reduction in SPEC CPU2017 using WHISTLE, based on the number of PCs allowed to access each memory location.

7.5 Exception Latency Reduction

We measure how fast security exception is raised before instructions are committed. We collect the number of cycles elapsed between memory request, invariant check and instruction retirement, and calculate how much earlier the proposed exception is raised, compared to the number of cycles elapsed between memory request and instruction retirement with the assumption that the exceptions in the baseline without any mitigation for memory safety violations occur at retirement of the corresponding instruction. In Figure 11, a light red bar represents the cycle
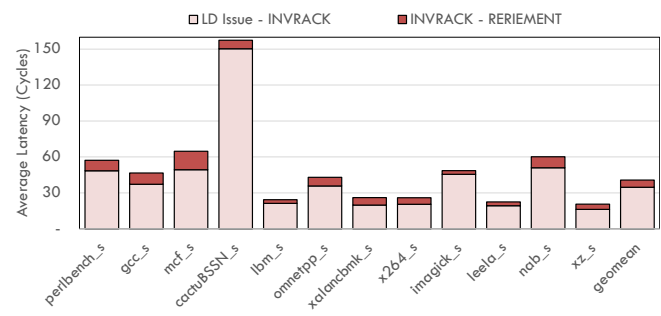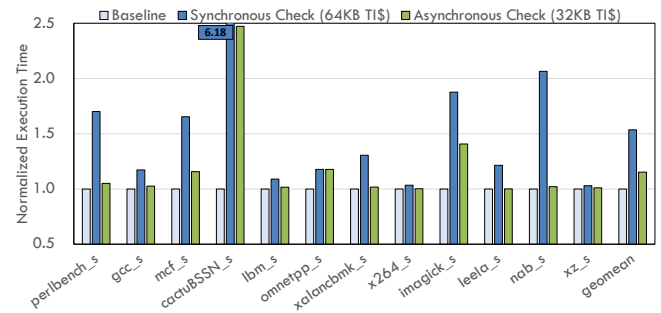

Fig. 12: Execution time for CPU2017. Invariants are profiled as cache line granularity.

difference between memory request and invariant check, which is the exception latency with asynchronous check. The entire bar with light and dark red bar represents the cycle difference between memory request and the retirement, which is the exception latency with baseline. On an average, the security exception requires 15% less time than that of the baseline system. Since, in asynchronous check, data could be supplied to the core before invariant check is finished, there may exist a small window of exploitation. Note that for applications with strong security requirement, we can enable synchronous checking.

8 OVERHEAD EVALUATION

We first show the performance overhead of the proposed microarchitecture with invariant check over the baseline, analyze the source of the overhead, and discuss how to overcome. Then, we describe the trade-off between different invariant check policies. Last, we evaluate overhead of area and energy. We observe that *cactuBSSN_s* and *lbm_s* allocate the large number of heap objects and few heap objects with large size respectively. We limit the number of heap objects and the maximum heap size to 500 objects and 100MB respectively. The reason is that the heap size can be up to gigabytes and causes the invariant size to explode. We believe that it can be improved by applying compression or deduplication techniques. We leave this work for future work. After adjustment, benchmarks generate invariant set with maximum size 251MB and 27MB on an average.

8.1 Performance Overhead

Figure 12 shows the normalized execution time of CPU2017 over baseline. For each program, we check the invariants based on cache line granularity and simulated for 1 billion instructions after warming up microarchitecture states with 1 billion instructions in system-call emulation mode. Average performance overheads of
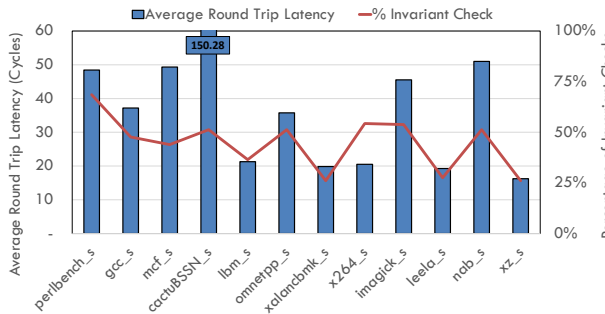
Fig. 13: Average Roundtrip Latency (Left Y Axis) and Total Number of Invariant Check (Right Y Axis).
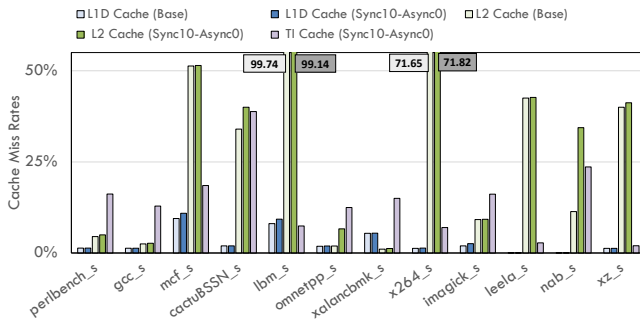

Fig. 14: Miss rates of L1D, L2, and TI Cache.

Fig. 15: Sensitivity of TI cache block numbers and size. TI Cache are configures with 256B block size. *perlbench_s* is configured differently with 512 block size and 128 cache blocks for 64KB size TI cache.

Fig. 16: Increment of binary size after embedding the invariants.

synchronous and asynchronous invariant check are 53% and 15% respectively. We use 64KB size of the TI cache for synchronous checking and 32KB of that for asynchronous checking to efficiently use the cache capacity. The main sources of performance degradation are round-trip latency and the number of invariant checks out of total L1D cache accesses, which are shown in Figure 13[3]. Benchmarks with greater latency and more number of invariant check have higher performance overhead compared to other benchmarks. For instance, we observe the overhead of *cactuBSSN_s* as an outlier, which can be attributed to extremely large number of invariant sets (298,649) resulting in high average round trip latency shown in Figure 13. Note that the number of *Top Invariant Cache Blocks (TIB)* are 128 that is not sufficient size for *cactuBSSN_s*. On the other hand, *x264_s and xz_s* have negligible overheads (3%) due to small number of invariant sets and good locality making small TI cache miss rate as shown in Table 4 and Figure 14 respectively. Because the number of invariant checks are the property of benchmarks, we cannot reduce them. Instead, we focus on latency of invariant check which depends on the performance of TI cache. As shown in Figure 14, benchmarks with high overhead have high miss rate in TI cache. We consider a hit in TI cache if PC is found in the TI cache block. In other words, even if the TI cache block is installed, if the PC is not found, it is miss because TI cache should forward CHKINVR message to lower level cache. For example, *cactuBSSN_s and imagick_s* suffer from in low performance because of the high miss rate in TI cache with 38% and 27% respectively.

8.2 Sensitivity of TI Cache

We study the sensitivity of TI cache size to understand the performance impact with different size of TI cache and its configuration. Figure 15 shows the miss rate of TI cache with different
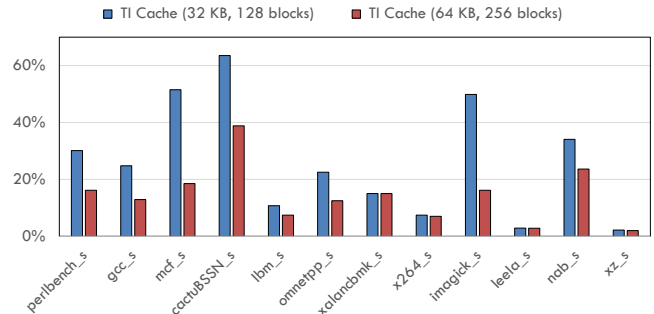
number of blocks (128 and 256 blocks for every benchmark except *perlbench_s*) and wider blocks (512 byte that can store 64 most frequently used PCs for *perlbench_s*). We observe that miss rates are reduced with more number of TI cache blocks but not *perlbench_s*. This is because *perlbench_s* has 10% of invariant sets with more than 32 PCs as shown in Figure 7 and we observe that accesses from 10% is still significant. Therefore, we doubled the block size for *perlbench_s* instead increasing the number of blocks and the miss rate decreased. Doubling the cache size does not incur area overhead as discussed in §8.5 so we can improve the performance with even larger than 64KB TI cache. On the other hand, asynchronous is not much sensitive than synchronous check as shown in Figure 12. This shows the high performance performance overhead with 32KB TI cache but not huge reduction of the performance with asynchronous check. Another optimization can further improve the round trip latency for invariant check. For example, becuase WHISTLE uses indirect Tag for TI cache block access (§6.8.2), it requires extra memory access that increases the miss penalty. We could use hash function with tags (invariant pointer address and offset) for getting addresses of invariant blocks. We leave this work for future work.

8.3 Performance Impact of Invariant section

We evaluate size of invariant section and its impact on performance. Figure 16 shows the increment of binary sizes of each benchmark after the invariants are embedded. Size of invariant section is mainly determined by the number of CCID and the number of invariant blocks for global, stack, and heap as described in Figure 6. For example, *cactuBSSN_s* is profiled with 251MB size of invariant section due to greater number of invariant blocks compared to other benchmarks. We observed 27MB size of invariant section on an average.

---

3. We observe that there are many memory accesses to sections of ELF binary during libc library functions calls. That is the reason why the number of checks are not mostly full even if WHISTLE checks all the memory access to global, stack, and heap objects.
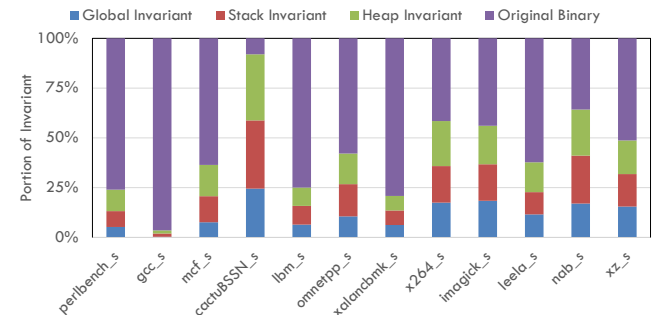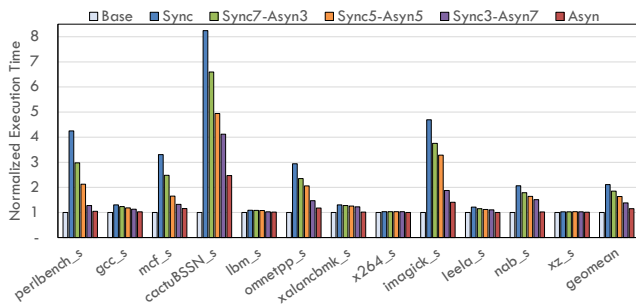
Fig. 17: Performance changes across different ratios of synchronous and asynchronous. TI Cache with 32KB size used. There are spectrum of synchronous-asynchronous checks - 100%-0% (Sync), 70%-30% (Syn7-Async3), 50%-50% (Syn5-Async5), 30%-70% (Syn3-Async7), and 0%-100% (Async).

	Area (mm²)	Energy/Access (nJ)
TI Cache (32KB/64KB)	3.19561/3.36907	1.077067/1.090471
HeapID Cache	0.323398	0.0991182
BTB & RAS	0.1977129	0.0517251

TABLE 5: Area and energy overhead of each component added by WHISTLE.

8.4 Comparison of Invariant Check Policies

We conduct an experiment to evaluate the performance overhead of WHISTLE with mixture of synchronous and asynchronous invariant checks. In order to see the performance trade-off between them, all the targeted memory accesses are randomly marked whether it is either synchronously or asynchronously checked based on a given ratio. We configure the ratios as 70%-30%, 50%-50%, and 30%-70% for synchronous-asynchronous checks, respectively, and have one run for each configuration. Figure 17 shows the performance overhead decreases as the portion of asynchronous check increases. Depending on the security level, WHISTLE can adjust the ratio of synchronous-asynchronous check for better performance.

8.5 Area and Energy Overhead

We estimate hardware budget using CACTI-7 [32] at 22nm. WHISTLE uses TI Cache to hold invariant and it has two tags–data and invariant pointer and extra 8 bytes metadata to store address of invariant block and its block size is 256 bytes. Heap ID cache uses both start and end address of corresponding heap object for tag matching to find CID of heap creation on the core side. Also, WHISTLE extends branch target buffer (BTB) and return stack buffer (RSB) to store 8 byte CCW and CCID. TI cache with 32KB size takes 3.19561 mm^2 of area and 1.077067 nJ of energy and 64KB size of TI cache incurs 5% more area and 1% more energy. Heap ID cache takes 0.323398 mm^2 of area and 0.0991182 nJ of energy. Extended BTB increase 0.197777129 mm^2 of area and 0.517251 nJ of energy. We consider that parallel tag matching logic in TI cache is being implemented using Content Addressable Memory (CAM), which has very low area, energy, latency implication.

9 RELATED WORK

In this section, we discuss hardware defenses for memory safety. We summarize the prior works in Table 6.

Dynamic Information Flow Tracking (DIFT): One of the challenges of DIFT is the runtime overhead. To reduce this overhead, LIFT [36] eliminates unnecessary checks by dynamic binary

	Title	SP	TP	TR	SE	FS	AB
Spatial	DataSafe [33]	✓	✓	✗	✗	✓	○
	DIFT [34]	✓	✓	✗	✗	✗	○
	Rakhsa [35]	✓	✓	✗	✓	✓*	○
	LIFT [36]	✓	✓	✗	✗	✗	○
Bounds Check	HardBound [23]	✓	✗	✗	✓	✗	○
	Intel MPX [15]	✓	✓	✗	✓	✗	○
	BOGO [5]	✓	✓	✗	✗	✗	○
	CHERIvoke [24]	✓	✓	✗	✓	✗	○
	REST [14]	✓	✓	✗	✗	✗	⊘
	Caliform [9]	✓	✓	✓*	✓	✗	⊘
	CHEx86 [10]	✓	✓	✓*	✗	✗	○
	AOS [4]	✓	✓	✗	✓	✗	○
	HeapCheck [13]	✓	✓	✓	✗	✗	○
	No-FAT [12]	✓	✓	✗	✓	✗	○
Monitoring	Adjustable Monitoring [37]	✓	✓	✗	✗	✓	⊘
	FADE [38]	✓	✓	✗	✗	✓	⊘
	FlexCore [39]	✓	✓	✗	✗	✓	⊘
	Harmoni [40]	✓	✓	✗	✗	✓	○
	HDFI [41]	✓	✓	✗	✗	✗	○
	MemTracker [42]	✓	✓	✗	✗	✗	○
	NILE [43]	✓	✓	✗	✗	✗	○
	PHMon [44]	✓	✓	✗	✓	✗	○
	Watchdog [6]	✓	✓	✗	✗	✗	○
CFI	SpecCFI [7]	✓	✗	✓	✓	✗	○
	C³ [11]	✓	✓	✓	✓	✗	○
	WHISTLE	✓	✓	✓	✓	✓	○

TABLE 6: Summary of Prior Works. **SP**: Spatial memory safety, **TP**: Temporal memory safety, **TR**: Transient memory safety, **SE**: Security exception, **FS**: Flexible Security, **AB**: Allowlisting or Blocklisting approach (✓: Fully Supported, ✗: Not Supported, ✓*: Partially Supported, ○: Allowlisting, ⊘: Blocklisting.).

inspection. Later, FPGAs are used for low overhead information tracking. *Compared to DIFT,* WHISTLE *focuses on detecting the very instruction which accesses the sensitive variables, rather than tracking the information flow beforehand or afterwards.*

Bounds Checking: Bounds checking [2] detects memory access that exceeds the expected lower or upper bound. Architectural stupports are proposed for bound checking in recent works [5], [15], [23], [24]. Several other works apply coloring to implement allowlisting policies [25], which fail to support intra-object memory protection. Recently, REST [14] and Caliform [9] employ blocklisting policies to detect memory safety violation. CHEx86 proposes a speculative pointer tracking mechanism to track pointers and support bounds checking by intercepting malloc function [10] while AOS instruments malloc function to propagate pointer information to hardware for heap object bounds checking [4]. HeapCheck [13] enforces bounds checking on memory requests from the CPUs, based on object bounds provided from hooked allocation and deallocation routines. No-FAT [12] uses statically transformed instructions to enforce bounds checking on heap objects, with object bounds determined from memory locations. *Compared to bounds checking,* WHISTLE *provides a more general approach to check memory safety rules, including rules that are within objects.*

Monitoring Based Solutions: Other works focus on monitoring memory violations at runtime based on given policy [6]. Nile [43] and PHMon [44] are recent works which provide hardware assisted frameworks for general monitoring. Flexible support for different security levels can be realized through different policies and extensions, or allocating various security budgets [37], [38]. However, none of aforementioned works considers transient execution memory safety threats as hardware vulnerabilities exploited by Spectre and Meltdown (except CHEx86, which defends against Spectre-v1). Recently, hardware defenses are proposed to isolate the impact of speculative execution before the changes

become permanent in cache hierarchy [8], [16]. *The design of WHISTLE is meant to detect the violating instruction, instead of mitigating the consequence (e.g., side channel) of violation in cache, TLB, or other components.*

Similar to this work, SpecCFI [7] takes allowlisting approach and uses in-architecture checks for jump, call and return targets within transient execution, to prevent Speculative control-flow attacks [3]. SpecCFI generates the CFI rules using the existing compiler support. WHISTLE *focuses on data access but can be extended for CFI.*

Cryptographic Capability Computing (C^3) [11] encrypts both the values and the corresponding pointers using encryption keys generated from the sizes, the size-aligned base addresses, and versions of the pointers. C^3 can prevent both spatial and temporal memory safety violations, since any of these violations will lead to wrong encryption keys and garbled plaintexts. WHISTLE *also offers uniform protection against spatial, temporal, and speculative memory violations, but does not require memory encryption.*

10 CONCLUSIONS

We proposed WHISTLE, a program invariant-based technique to detect HW and SW memory violations. Our proposed hardware profiler can construct memory invariants from both transient and non-transient instructions. The proposed TI cache enables fast checking of invariants when loading data. TI cache works with the memory hierarchy to store invariants at different levels based on access frequency. WHISTLE provides both synchronous and asynchronous checking of invariants; the latter includes a fast security exception to alert the OS about an attempted access that violates the invariants. We believe WHISTLE to be a stepping stone towards a systematic solution to prevent both HW and SW memory safety violations.

ACKNOWLEDGMENTS

REFERENCES

[1] C. W. Enumeration, "2020 cwe top 25 most dangerous software weaknesses." https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.

[2] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *SIGPLAN Not.*, vol. 44, p. 245–258, June 2009.

[3] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[4] Y. Kim, J. Lee, and H. Kim, "Hardware-based always-on heap memory safety," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1153–1166, IEEE, 2020.

[5] T. Zhang, D. Lee, and C. Jung, "BOGO: buy spatial memory safety, get temporal memory safety (almost) free," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 631–644, 2019.

[6] S. Nagarakatte, M. M. Martin, and S. Zdancewic, "Watchdog: Hardware for safe and secure manual memory management and full memory safety," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 189–200, IEEE, 2012.

[7] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "SpecCFI: Mitigating spectre attacks using cfi informed speculation," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 39–53, IEEE, 2020.

[8] S. Kim, F. Mahmud, J. Huang, P. Majumder, N. Christou, A. Muzahid, C.-C. Tsai, and E. J. Kim, "ReViCe: Reusing Victim Cache to PreventSpeculative Cache Leakage," in *2020 IEEE Secure Development Conference (SecDev)*, September 2020.

[9] H. Sasaki, M. A. Arroyo, M. T. I. Ziad, K. Bhat, K. Sinha, and S. Sethumadhavan, "Practical byte-granular memory blacklisting using califorms," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 558–571, 2019.

[10] R. Sharifi and A. Venkat, "CHEx86: Context-sensitive enforcement of memory safety via microcode-enabled capabilities," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 762–775, IEEE, 2020.

[11] M. LeMay, J. Rakshit, S. Deutsch, D. M. Durham, S. Ghosh, A. Nori, J. Gaur, A. Weiler, S. Sultana, K. Grewal, and S. Subramoney, "Cryptographic capability computing," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, (New York, NY, USA), p. 253–267, Association for Computing Machinery, 2021.

[12] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, R. Piersma, and S. Sethumadhavan, "No-fat: Architectural support for low overhead memory safety checks," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, p. 916–929, IEEE Press, 2021.

[13] G. Saileshwar, R. Boivie, T. Chen, B. Segal, and A. Buyuktosunoglu, "Heapcheck: Low-cost hardware support for memory safety," *ACM Trans. Archit. Code Optim.*, vol. 19, jan 2022.

[14] K. Sinha and S. Sethumadhavan, "Practical memory safety with REST," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, (Piscataway, NJ, USA), pp. 600–611, IEEE Press, 2018.

[15] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: A cross-layer analysis of the intel mpx system stack," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, pp. 1–30, 2018.

[16] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 428–441, IEEE, 2018.

[17] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 973–990, 2018.

[18] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, 2011.

[19] C. Canella, J. V. 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 *28th USENIX Security Symposium (USENIX Security 19)*, (Santa Clara, CA), pp. 249–266, USENIX Association, Aug. 2019.

[20] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the evaluation of software defect detection tools*, vol. 5, 2005.

[21] NIST, "Software Assurance Reference Dataset (SARD) project." https://samate.nist.gov/SARD, 2017. Last accessed 10 Mar 2022.

[22] "SPEC releases major new CPU benchmark suite." https://www.spec.org/cpu2017/press/release.html.

[23] J. Devietti, C. Blundell, M. M. Martin, and S. Zdancewic, "Hardbound: architectural support for spatial safety of the c programming language," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 103–114, 2008.

[24] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. Watson, *et al.*, "CHERIvoke: Characterising pointer revocation using cheri capabilities for temporal memory safety," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 545–557, 2019.

[25] "Hardware-assisted checking using silicon secured memory (ssm)." https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html.

[26] A. Muzahid, N. Otsuki, and J. Torrellas, "Atomtracker: A comprehensive approach to atomic region inference and violation detection," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, December 2010.

[27] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha, "Securecore: A multicore-based intrusion detection architecture for real-time embedded systems," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 21–32, IEEE, 2013.

[28] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. L. Traon, "Validity fuzzing and parametric generators for effective random testing," in *Pro-*

ceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019* (J. M. Atlee, T. Bultan, and J. Whittle, eds.), pp. 266–267, 2019.

[29] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas, "Accmon: Automatically detecting memory-related bugs via program counter-based invariants," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, pp. 269–280, IEEE, 2004.

[30] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang, "Precise calling context encoding," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1160–1177, 2011.

[31] "Sensei's libray." https://senseis.xmp.net/?GoDatabases.

[32] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[33] Y.-Y. Chen, P. A. Jamkhedkar, and R. B. Lee, "A software-hardware architecture for self-protecting data," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 14–27, 2012.

[34] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," *ACM Sigplan Notices*, vol. 39, no. 11, pp. 85–96, 2004.

[35] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 482–493, 2007.

[36] 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 *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pp. 135–148, IEEE, 2006.

[37] D. Lo, T. Chen, M. Ismail, and G. E. Suh, "Run-time monitoring with adjustable overhead using dataflow-guided filtering," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 662–674, IEEE, 2015.

[38] S. Fytraki, E. Vlachos, O. Kocberber, B. Falsafi, and B. Grot, "Fade: A programmable filtering accelerator for instruction-grain monitoring," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 108–119, IEEE, 2014.

[39] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh, "Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 137–148, IEEE, 2010.

[40] D. Y. Deng and G. E. Suh, "High-performance parallel accelerator for flexible and efficient run-time monitoring," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pp. 1–12, IEEE, 2012.

[41] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-assisted data-flow isolation," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 1–17, IEEE, 2016.

[42] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic, "Memtracker: Efficient and programmable support for memory access monitoring and debugging," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pp. 273–284, IEEE, 2007.

[43] L. Delshadtehrani, S. Eldridge, S. Canakci, M. Egele, and A. Joshi, "Nile: a programmable monitoring coprocessor," *IEEE Computer Architecture Letters*, vol. 17, no. 1, pp. 92–95, 2017.

[44] L. Delshadtehrani, S. Canakci, B. Zhou, S. Eldridge, A. Joshi, and M. Egele, "PHMon: A programmable hardware monitor and its security use cases," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 807–824, USENIX Association, Aug. 2020.
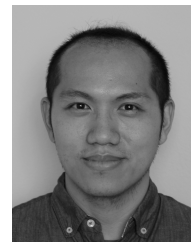
Farabi Mahmud received the BSc degree in Computer Science and Engineering from Bangladesh University of Engineering & Technology in 2017. He is a PhD student in the Department of Computer Science and Engineering, Texas A&M University. His research interests are the fields of computer architecture and systems, especially on networks-on-chip and hardware security. Before starting a PhD, he worked as a lecturer at United International University, Dhaka.
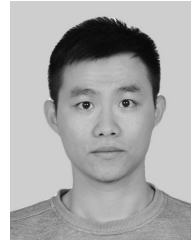
Jiayi Huang (Member, IEEE) received the BEng degree in information and communication engineering from Zhejiang University, China, in 2014, and the PhD degree in computer engineering from Texas A&M University, in 2020. He is currently a postdoctoral researcher with the Department of Electrical and Computer Engineering, UC Santa Barbara. His research interests include computer architecture, computer systems, and security. He is a member of the ACM and the IEEE Computer Society.

Pritam Majumder received the B.Tech degree in Computer Science and Engineering from WBUT, India, in 2011. He received his MS degree in Computer Science and Engineering from Indian Institute of Technology, Madras, in 2015. He is a Ph.D. student in the Department of Computer Science and Engineering, Texas A&M University. His research interests lie in the fields of computer architecture and systems, and machine learning. He is a student member of the ACM.

Chia-Che Tsai received the BS degree in computer science and information engineering from National Taiwan University, Taiwan, the MS degree in computer science from Columbia University, and the PhD degree in computer science from Stony Brook University. He is an assistant professor in the Department of Computer Science and Engineering at Texas A&M University. His research interests include operating systems, software and hardware security, and cloud computing.

Abdullah Muzahid received his BS in Computer Science from Bangladesh University of Engineering and Technology. He received his MS and PhD in Computer Sceince from University of Illinois at Urbana-Champaign. He is an assistant professor at the Department of Computer Science and Engineering of Texas A&M University. His research broadly focuses on various aspects of computer architecture and systems. More specifically, he is interested in multiprocessor architecture, parallel programming, programming models, debugging, program analysis and synthesis. Recently, he is interested in applying machine learning to solve various system-related issues.

Eun Jung Kim received the BS degree in Computer Science and Engineering from KAIST, Korea, the MS degree in computer science from Pohang University of Science and Technology, Korea, and the PhD degree from the Department of Computer Science and Engineering, Pennsylvania State University. She is an associate professor in the Department of Computer Science and Engineering, Texas A&M University. Her research interests include computer architecture, power efficient systems, parallel/distributed systems, cluster computing, and hardware security. She is a member of the IEEE Computer Society. More information about her research is available at http://faculty.cse.tamu.edu/ejkim.

Sungkeun Kim received the B.Eng degree in Computer Science and Engineering from Kyungpook National University, Republic of Korea, in 2011. He is a Ph.D. student in the Department of Computer Science and Engineering, Texas A&M University. His research interests are the fields of computer architecture and systems, especially on networks-on-chip, memory systems and near-data processing, and hardware security. Before starting a Ph.D., he worked as a software engineer at Samsung Electronics, Suwon, Republic of Korea.