PoisonIvy: Safe Speculation for Secure Memory

Tamara Silbergleit Lehman, Andrew D. Hilton, Benjamin C. Lee
Electrical and Computer Engineering
Duke University
{tamara.silbergleit, andrew.hilton, benjamin.c.lee}@duke.edu

***Abstract*—Encryption and integrity trees guard against physical attacks, but harm performance. Prior academic work has speculated around the latency of integrity verification, but has done so in an insecure manner. No industrial implementations of secure processors have included speculation. This work presents `PoisonIvy`, a mechanism which speculatively uses data before its integrity has been verified while preserving security and closing address-based side-channels. `PoisonIvy` reduces performance overheads from 40% to 20% for memory intensive workloads and down to 1.8%, on average.**

I. Introduction

Modern computing models provide physical access to untrusted entities, such as datacenter operators, who might examine sensitive code or data. Hardware mechanisms that guard against physical attacks must ensure confidentiality and integrity as data leaves the secure processor. For confidentiality, the memory controller encrypts and decrypts data to and from memory. Best practices combine one-time pads and counter-mode encryption [12], [48].

For integrity, the memory controller hashes data [9], [18], [19], [37], [49], [50] and then hashes the hashes to construct a tree [11], [20], which can determine whether data has been altered since it was last written into memory. Trees use keyed Hash Message Authentication Codes (HMACs). Best practice builds the integrity tree over counters used to encrypt data, instead of the data itself, which ensures integrity with reduced space requirements [12], [25].

Despite recent advances, security mechanisms are expensive and harm performance as they manipulate several types of metadata, which are stored in memory along with data. When requesting data, the memory controller must also request counters for decryption and hashes for verification. In the worst case, a last-level cache miss could trigger up to ten memory transfers—counters for decryption and hashes from the tree, up to the root, for integrity verification—significantly increasing request latency and bandwidth demand.

Metadata caches mitigate performance overheads. Academic designs place metadata in the L2 cache along with the data [11], [25], [48], [49], while industry designs use a dedicated cache in the memory encryption engine [12]. Caches mitigate, but do not eliminate, overheads. With a 32KB cache dedicated to metadata, we find that security harms performance by 4% on average and 98% in the worst case.

Speculation further reduces performance overheads by supplying data to the core for computation before retrieving its metadata and verifying its integrity. Some academic mechanisms speculate quite eagerly, using data immediately and assuming integrity violations are detected before they produce any ill effects [25], [38]. Others, such as `authen-then-write`, speculate more carefully for smaller yet still significant performance benefits [32].

However, speculation has a number of limitations despite academic progress and demonstrated potential. To date, none of the proposed mechanisms are completely safe and all of them open an avenue of attack that breaches chip security. Even the conservative `authen-then-write` mechanism is vulnerable to side-channel attacks, which are identified in the same study [32]. Industrial designs, such as Intel's Software Guard eXtensions (SGX) [4], rely primarily on caches, not speculation, to balance security and performance. To encourage the adoption of speculative mechanisms, we must close its security vulnerabilities.

Contributions. We propose `PoisonIvy`, a means of safe speculation. It guarantees that no information affected by unverified data escapes the chip. `PoisonIvy` restricts not only data but also addresses sent to memory. The first restriction ensures integrity as stores cannot write results from speculative computation to memory. The second closes side-channels, discovered in prior work [32], as requests with potentially affected addresses cannot traverse the memory bus.

`PoisonIvy` uses poison bits to track parts of the processor affected by unverified data. It draws inspiration from poison in Continual Flow Pipelines (CFP) [35], but differs in several ways. Unlike CFP, `PoisonIvy` tracks poison not only for registers, but also for instructions, control flow, and address translation. Whereas CFP uses poison to track instructions that cannot compute until an unknown value is supplied by a long-latency load, `PoisonIvy` tracks computation on known but unverified values. Because it permits computation to progress as long as data and addresses are restricted to the chip, `PoisonIvy` can track and clear poison at a coarser granularity than CFP—in epochs rather than per load.

We begin by surveying the state of the art in memory security, detailing mechanisms for confidentiality, integrity, and performance (Section II). We then present the `PoisonIvy` mechanism for safe speculation, describing poison propagation and epoch-based poison management (Section III). We find that `PoisonIvy` reduces overheads from memory security and performs as well as unsafe speculation, in which unverified data is used without regard for safety (Section IV).

978-1-5090-3508-3/16/$31.00

II. Background

We consider a well studied threat model in which an attacker has physical access to the machine [6], [15], [47]. When the processor is the trusted computing base, it must safeguard its interactions with off-chip memory. It ensures confidentiality with encryption and ensures integrity with Merkle trees [4], [6], [10], [38]. Security adds overheads, motivating performance optimizations that cache metadata and speculate around safeguards [11], [26], [27], [33], [34], [37], [49]. We survey recent progress to motivate our solution, `PoisonIvy`, which builds atop best practices to address remaining performance challenges.

System Model. An application's owner uses a tool chain to protect her binary before sending it to a vulnerable system. While industry-strength tools, such as those of Intel's SGX, are complex, they can be broken down into a simplified model that has three steps [4], [7]. First, the tool creates a random Advanced Encryption Standard (AES) key and encrypts the binary image, which includes both code and data. Second, the tool encrypts the AES key with the RSA public key of the target system. Finally, the AES-encrypted image and the RSA-encrypted key are paired and sent to the system for computation.

The secure processor runs the application by unpacking the image in two steps. First, the processor uses its matching RSA private key to decrypt the AES key. With the AES key, the processor decrypts code and data as they are read from disk, and re-encrypts them with its own key, then writes them to memory. A separate AES key, known and used only by the processor, is used to read data from memory into caches and vice-versa. When code and data are swapped out, the processor decrypts it with its own key, re-encrypts it with the application AES key, and writes it to disk

When combined with an integrity tree, encryption guards against a number of attacks. Examples include (i) replay, in which old data is injected to manipulate the system; (ii) snooping, in which off-chip communication is observed; (iii) tampering, in which data into or out of the processor is modified.

A. Confidentiality: Counter-Mode Encryption

Keyed encryption is required for confidentiality, which prevents adversaries from observing data. The processor could use XTS-mode,[1] which combines data with portions of its memory address before encrypting with AES. However, XTS serializes data access and decryption, lengthening the critical path by the AES latency, which ranges from 70 to 100 processor cycles. Haswell's AESDEC performs one round in 7 processor cycles and performs the 10-14 required rounds in 70-98 cycles, depending on key length [13]. Prior work assumes 80 cycles for decryption [48].

Counter-Mode Encryption. Alternatively, counter-mode encryption XORes plaintext with a one-time pad (OTP) to produce the ciphertext. The OTP is generated by encrypting a combination of a counter and the memory address with AES [37], [49], [48]. During decryption, the ciphertext is XORed with the same OTP to produce the plaintext. If counters are cached, the processor can generate the OTP for decryption and retrieve the ciphertext from memory in parallel. In this situation, the decryption latency overhead is only the XOR latency, which is less than one cycle.

The OTP, and by extension the counter, is used only once. The counter corresponding to a memory address is incremented after every write. If a counter were to overflow, the AES key used to encrypt counters and produce OTPs must be regenerated. Then, counters must be reset and memory contents must be re-encrypted. Because counter overflow is expensive, best practice increments counters at multiple data granularities.

Block and Page Counters. Large counters avoid overflow and expensive re-encryption, but small counters fit in cache more easily. Researchers resolve this tension with separate counters for memory blocks and pages [25], [48]. Every block write increments the per-block counter. Overflow in the per-block counter increments the per-page counter corresponding to the block's page. Finally, each block's input to OTP generation is the concatenation of per-block and per-page counters.

Block and page counters reduce the frequency and cost of re-encryption. When the per-page counter increments, the memory controller loads each block from the page, decrypts it, and re-encrypts it with a new counter that concatenates the incremented per-page counter and a reset per-block counter. Although page re-encryption is moderately expensive, it happens infrequently and is much less expensive than re-encrypting all memory contents. The combined counter (7-bit per-block and 64-bit per-page counters) never overflows in practice. It would overflow in 75,000 years if the same block were written once per nanosecond.

B. Integrity: Bonsai Merkle Trees

Integrity mechanisms prevent adversaries from modifying data without a user's knowledge. One approach uses hashes to verify memory integrity. For every write, the memory controller computes and stores a keyed hash of the data. For every read, it loads and compares the previously computed hash against the current one. When hashes are equal, integrity is verified. When hashes differ, the system detects tampering and halts.

Integrity Trees. Hashes alone do not guard against replay attacks in which adversaries inject previously observed data and hashes onto the memory channel. As illustrated in Figure 1, Merkle Trees guard against replay attacks by building a hash tree over memory contents [20]. Leaves of the tree contain one hash per data block. Further up, each tree node hashes its children's hashes. The tree's root holds a hash that encompasses the whole memory space. The tree's root is held on chip, where it cannot be attacked, and is always trusted. A data write propagates hash writes up the tree to the root.

Bonsai Merkle Trees (BMTs) guard against replay attacks at lower cost than Merkle Trees. BMTs construct a hash tree to

[1] Xor Encrypt Xor (XEX) Tweakable Block Cipher with Ciphertext Stealing

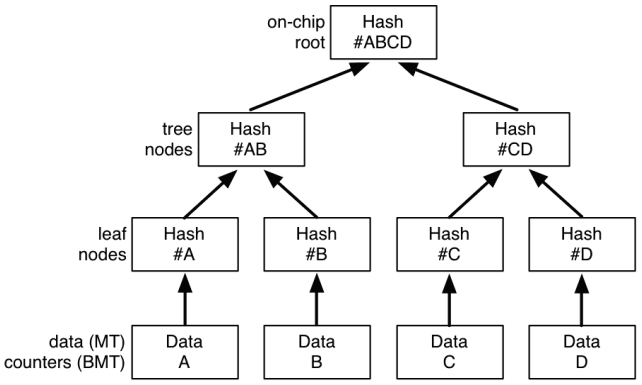

Fig. 1: Trees ensure integrity by hashing data and then hashing the hashes. Merkle Trees (MT) protect data whereas Bonsai Merkle Trees (BMT) protect counters.

protect only the counters used to generate OTPs instead of the entire memory space. This approach reduces space overheads from 50% to 14–34% [25]. Moreover, BMTs perform better because counters are often cached on chip and need not be verified by the tree. Suppose the processor requests data and its corresponding counter is cached, the counter's integrity was verified when it was first loaded from memory and BMT (re-)traversal is unnecessary. Given BMTs to protect counters, simple keyed hashes are sufficient to protect the integrity of the data.

C. Performance: Unsafe Speculation

Security increases average memory access time. Data en route from memory to the last-level cache must wait for decryption plus integrity verification, which may need data hashes and tree nodes from memory. Speculating on integrity removes the second set of overheads from the critical path. The memory controller supplies data to the processor before verifying integrity, permitting computation on decrypted but unverified data.

Performance versus Security. Speculation exposes tensions between performance and security. For performance, speculation requires a permissive mechanism that computes far ahead of verification to hide its latency, which is tens of cycles when counters are cached but hundreds (to thousands) of cycles when counters and tree nodes are in memory. Unfortunately, instructions will compute on unverified data, posing risks to confidentiality and integrity.

For security, speculation must restrict the visibility of computation on unverified data. A conservative mechanism for speculation does not modify architected state [32], but it performs poorly as the instruction window fills quickly and exposes long verification latencies. Furthermore, guarding architected state alone is insufficient because speculative computation may issue loads and stores that send unencrypted addresses across the memory channel.

Prior work delivers performance or security but rarely both. A less conservative approach allows the memory controller to return unverified data to the core, assuming that the window of vulnerability between data use and verification is too short for attackers to exploit [6], [25], [27], [38], [47]. Shi *et.al.*, close this vulnerability by stalling stores until all outstanding verifications complete ("`authen-then-write`") [32]. This approach prevents an application from being "tricked" into writing data to memory, but puts pressure on the store buffer, constrains speculation, and exposes verification latency. Although researchers have sought to scale store buffer capacity [29], [30], [31], [36], no modern processor has a large enough store buffer to tolerate thousands of cycles of latency.

Side-Channel Attacks. Shi *et.al.* present a series of side-channel attacks, in a system that speculates, that leak secrets via the memory address bus before verification fails [32]. These attacks exploit the fact that, when using OTP encryption, each bit of ciphertext corresponds to exactly one bit of plaintext. If an attacker wishes to flip a specific bit of plaintext, she can flip the corresponding ciphertext bit without breaking any cryptographic system.

In one attack, Shi *et.al.*, assumes that the attacker knows the contents of instruction memory (*e.g.*, has a copy of the program). First, the attacker alters bits in instruction words to cause secret data to be used in a load's address, taking care to place the secret into bits unaffected by address translation but not within the block offset. Then, the attacker observes addresses on the memory bus to learn the secret. Other attacks include manipulating pointers in linked data structures, and searching for a secret value by repeatedly altering constants that are compared to the secret. These attacks show that safe speculation must safeguard both data and instructions.

Some might argue that changes to the encryption algorithm could close these side-channels and permit safe speculation. Although XTS-mode encryption avoids vulnerabilities from OTPs, it penalizes performance by serializing data access and decryption. Diffusive ciphers avoid the one-to-one mapping of ciphertext and plaintext bits, ensuring a bit flip in ciphertext changes multiple bits in plaintext. However, diffusion only reduces odds of a successful attack. If an attacker targets N bits—and does not care about the remaining $(512-N)$ bits—in a cache block, diffusion reduces the odds of success to 1 in 2^N. For example, 1 in 256 attacks are successful when 8 bits are targeted, which is far from secure.

D. Architectures for Secure Memory

Table I summarizes several representative architectures that combine the building blocks for secure memory—confidentiality, integrity, and performance. SGX is Intel's industrial design whereas AEGIS and AISE are academic designs. `PoisonIvy` provides safe speculation that could be used with any of these schemes. Being built atop state-of-the-art design decisions makes `PoisonIvy` immediately relevant.

Intel SGX. Software Guard eXtensions (SGX) provides secure regions of memory (*i.e.* enclaves) for programmer managed secrets. Instruction set extensions allow programmers to create, load, and enter an enclave. The Memory Encryption Engine (MEE), a microarchitectural extension to the memory

	Confidentiality	Integrity	Performance
SGX [4]	AES-Ctr-Mode; 56-bit counters per 512-bit blocks	Modified 4-Level Bonsai Merkle Tree; Carter-Wegman 56-bit MAC	Dedicated metadata cache; Cache size unknown No speculation
AEGIS [38]	AES-CBC; 128-bit	Merkle Tree 128-bit MAC	Shared LLC for data, tree nodes Unsafe speculation
AISE [25]	AES-Ctr-Mode; 64-bit counter per 4KB page; 7-bit counter per 512-bit block	Bonsai Merkle Tree; 128-bit MAC tags; HMAC SHA-2	Shared LLC for data, counters Unsafe speculation
`PoisonIvy`	AES-Ctr-Mode 64-bit counter per 4KB page; 7-bit counter per 512-bit block	Bonsai Merkle Tree 64-bit MAC tags; truncated 128-bit HMAC SHA-2	Dedicated 32KB metadata cache; Safe speculation

TABLE I: Comparison of Memory Security Mechanisms.

controller [12], encrypts and hashes enclave contents to ensure confidentiality and integrity.

SGX and `PoisonIvy` share fundamental security mechanisms with small implementation differences. For confidentiality, both SGX and `PoisonIvy` use counter-mode encryption, in which OTPs are generated by AES-128. SGX uses 56-bit per block counters whereas `PoisonIvy` uses 7-bit per block counters and 64-bit per page counters to reduce overflow costs and facilitate caching. For integrity, both SGX and `PoisonIvy` use Bonsai Merkle Trees (BMTs). While the length of the hashes—for both data and tree—used in SGX and `PoisonIvy` are the same (64 bits), SGX uses the Carter-Wegman algorithm, whereas `PoisonIvy` and other academic works use SHA-2.

AEGIS. Suh *et.al.* propose new instructions to allow an application to execute in a tamper resistant environment [38]. To protect the system against physical attacks, AEGIS uses AES-Cipher Block Chaining (CBC) for encryption and a Merkle tree with 128-bit MAC for integrity. AEGIS caches tree nodes in the last-level cache (LLC) along with data. In addition, it allows the processor to compute on unverified data and assumes that, if integrity fails, the system will halt before leaking information—an assumption we call unsafe speculation.

AISE. Rogers *et.al.* protect a system from physical attacks with a modified encryption and integrity mechanisms [25]. For encryption, they use AES-counter mode with two counters: a 64-bit counter per 4KB page and a 7-bit counter per 512-bit blocks. The counters are concatenated when producing the one-time pad. For integrity, they modify the traditional Merkle tree to produce the Bonsai Merkle Tree. BMTs protect counters, instead of the data, thereby shortening the tree height and decreasing its size. Counters are cached in the LLC along with data. This scheme uses unsafe speculation as well.

III. PoisonIvy: Safe Speculation

`PoisonIvy` makes speculation safe with a few key principles. First, unverified data may be used for any purpose within the processor, but it cannot affect any information leaving the processor before its integrity is verified. Second, any instruction affected by computation on unverified data is poisoned by the speculation. Instructions could be affected via input register values, preceding instruction words, preceding control flow, or values used in address translation. Speculation built around these principles is not only safe but also efficient. The only operations blocked by integrity verification are those that require off-chip communication.

`PoisonIvy`'s fundamental mechanism is *poison*, which is inspired by the key mechanism of Continual Flow Pipelines (CFP) [35]. CFP poison bits indicate a load's value is missing due to a long-latency cache miss. In contrast, `PoisonIvy`'s poison bits indicate an unverified value is being used speculatively. The microarchitecture uses poison to determine what information must be restricted to the processor.

Whereas CFP requires a mechanism to recover from misspeculation, such as checkpoints, `PoisonIvy` does not expect to recover from misspeculation. Computing on data for which integrity verification fails is a drastic problem, indicating a physical attack against the system. The only safe outcome is halting the system. Indeed, SGX halts the system whenever verification fails even though it does not speculate.

A. Poison Propagation and Use

Whenever the memory controller has data to handle an LLC miss but cannot verify it immediately, it returns the data to the LLC and core speculatively. When a cache receives speculative data, it marks the line as poisoned. When the core receives speculative data, it poisons the output of the load that requested it. This poison propagates, marking instructions and data that are affected by the unverified value.

Registers. Poison bits propagate through registers as in CFP. When an instruction reads its input registers (or bypasses), it reads the corresponding poison bits. It then ORs these poison bits with the instruction's IW and CF poison bits, which are discussed next, to determine whether its output register is poisoned. Then, the poison information for its output register is propagated on the bypass network and written into the register file along with the instruction's output value. In out-of-order processors with register renaming, we require one poison bit per physical register. The poison bits for all registers are cleared once all outstanding verifications are completed successfully.

Figure 2 shows an example in which `i1` misses at the LLC and memory returns data speculatively, setting the poison bit on the output register `r1`. When `i2` reads its input registers, it finds that `r1` is poisoned, so it poisons its output register,

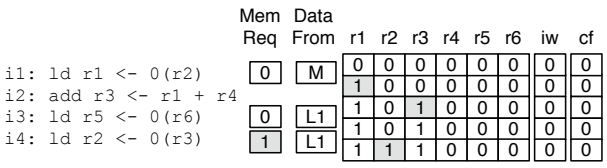

Fig. 2: The output register poison bit is set when poison bits for any of an instruction's input registers are set. Memory instructions send poison bits along with their request to the memory hierarchy.

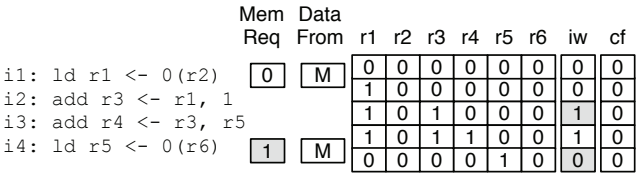

Fig. 3: The instruction word poison bit is set when an instruction cache miss is filled with speculative data. Instruction `i2` misses at the instruction cache and all the way down to memory. Instruction `i4` waits at the memory controller to be sent off-chip until all verifications are completed.

`r3`. `i4`'s input register `r3` is poisoned, so `i4`'s request to the memory hierarchy is also marked as poisoned. If the request misses at the LLC, the memory controller stalls the request until verification completes.

Instruction Words. Whereas CFP cannot speculate around an instruction cache miss and has no notion of poisoning instruction words, `PoisonIvy` must track poisoned instructions with an instruction word (IW) poison bit. If a fetch misses all cache levels, memory speculatively supplies a poisoned instruction word. An attacker may have tampered with the memory blocks holding these words and computation with these instructions cannot be trusted until verification completes. The IW poison bit is set at the processor front-end and cleared only when outstanding speculations are verified.

Figure 3 shows an example in which the IW poison bit is set. Instruction `i2` is retrieved from memory and the instruction cache miss is resolved with speculative data. This instruction and all subsequent ones carry IW poison along with them through the pipeline. When a memory request misses at the LLC, like `i4`, the memory controller stalls the request until verification completes. At this point all poison bits, including IW, are cleared. Register `r5` is now poisoned by the newly speculative load returned from memory.

After fetching an instruction speculatively, all subsequent fetches (even those that hit in cache) must be poisoned, as the instruction stream may be corrupted, changing data and control dependences. This requirement, combined with the rarity of instruction cache misses that are satisfied by memory, means there is little advantage to tracking poison bits for each line in the instruction cache. Instead, `PoisonIvy` uses one poison bit to track speculation on any instruction word.

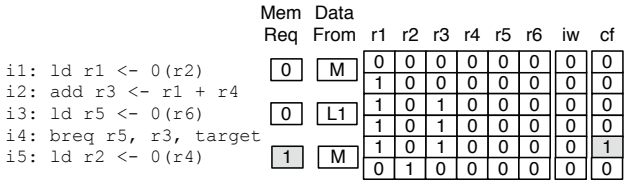

Fig. 4: The control flow poison bit is set when a branch is executed and when a poison bit for any of its input registers is set. The load on instruction `i5` has to wait at the memory controller to be fulfilled until all verifications complete.

Control Flow. Speculative data can affect the program's control flow by poisoning data values that dictate branches. `PoisonIvy` tracks poisoned control flow (CF) with one poison bit per thread. The CF poison bit is set in the register-read stage of the pipeline when any poisoned instruction might modify the program counter. A poisoned branch sets the bit whether it was correctly predicted or not. The CF poison bit is cleared when verification completes or instructions older than CF-poisoned instructions are squashed.

This method of setting the CF poison bit produces correct outcomes. First, any instruction that executes after a poisoned branch observes poisoned control flow. Second, an older instruction that executes after the branch, due to out-of-order scheduling, may observe a set CF bit even though its control flow was not actually poisoned. This is a safe and conservative outcome. Third, a younger instruction that executes before the branch observes a cleared CF bit. This is a correct outcome as anything the attacker did, absent other poison reaching this instruction, did not affect its computation. These rules greatly simplify implementation, eliminating the need to propagate poison through younger instructions that have already executed out-of-order.

Figure 4 shows an example with poisoned control flow. Suppose the value of `r3` is affected by speculation and instruction `i4` compares `r3` to `r5`. Unverified values could cause the program to execute different instructions. This vulnerability corresponds exactly to the binary search attack devised by Shi *et.al.* For this reason, `PoisonIvy` must track when control flow has been affected by poisoned values.

Although there is logically one CF poison bit per thread, an implementation can safely have multiple physical copies (*e.g.*, one per superscalar lane) if desired. These copies need not be updated at exactly the same time. Rather, they can be updated with the same latency that is required for branch misprediction signals to propagate to that part of the execution core.

Address Translation. The processor may need to access memory when translating a virtual address to a physical address. For example, it may miss in the TLB and fail to find the page table entry in cache. In such situations, the

translation itself may be returned speculatively from memory. Whenever PoisonIvy has a speculative translation, any memory request that uses the translation produces poisoned values and, in the event that it misses at all cache levels, cannot be allowed off the chip until speculation is verified.

When the DTLB receives a speculative translation, the poison bit in the entry is set. Memory instructions OR the poison bit of their translation with the poison bits of their other inputs to determine whether the instructions are poisoned. When the ITLB is filled with a speculative translation, the IW poison bit is set, poisoning all instruction words.

Memory Hierarchy. In PoisonIvy, the memory hierarchy must propagate poison and enforce invariants that ensure security. For poison propagation, every line in the L1 data cache and in lower-level caches has a poison bit added to the tags. When a line is filled speculatively, its poison bit is set. Additionally, whenever a poisoned store writes to a cache line, its poison bit is set. As previously mentioned, the instruction cache does not have per line poison bits. Instead, the IW poison bit is set whenever the instruction cache receives a speculative fill, whether directly from the memory controller or indirectly from a lower-level cache.

PoisonIvy's memory hierarchy ensures that (i) no poisoned data is written off-chip and (ii) no memory requests are made with poisoned addresses. The first of these is enforced by the LLC, which may not evict a dirty and poisoned block. When the LLC must evict a block, it treats dirty, poisoned lines as locked and selects a non-poisoned victim. If no such line exists—*i.e.*, all lines in the set are poisoned and dirty—then eviction must stall until speculation completes. Note that such stalls are extremely rare.

Second, the memory controller ensures that no request with a poisoned address escapes the chip. Memory requests carry their poison information with them down the memory hierarchy. If a poisoned request misses at the LLC and reaches the memory controller, the request stalls until speculation completes. Enforcing this rule at the memory controller, and not higher in the cache hierarchy, delays only off-chip requests. Those that hit in cache are satisfied with no performance penalty. Note that non-cacheable memory and IO operations must always stall until all poison bits are cleared.

PoisonIvy accounts for prefetchers, which are prevalent in memory systems. Each prefetcher maintains a poison bit, which is set whenever it is influenced by poisoned information. The prefetcher then includes this poison in each request that it generates. Poisoned prefetches may bring data through the cache hierarchy but stall at the memory controller until verification completes.

Timing Information. PoisonIvy accounts for one more subtle piece of state in the processor—timing information. Suppose a victim computes on speculative, unverified data and modifies the processor's shared resources such as the LLC. An attacker on another core observes some timing effect due to the victim's speculative computation, like higher average memory access time, as the attacker's data is evicted by the victim's writes to the LLC. Thus, the attacker has timing information derived from the victim's speculative execution that could be leaked from the processor. Closing this side-channel requires poisoning timing information.

PoisonIvy protects against this new timing side-channel attack with a Timing Poison (TP) bit. Whenever a program executes an instruction that can observe time (*e.g.*, rdtsc) or performance counters, the TP bit is ORed with the instruction's poison information. A rdtsc instruction poisoned by TP produces the correct time when executed, but its output register is poisoned. Time measurements or values computed from them cannot leave the chip until verification completes. Note that the TP bit does *not* close any existing timing side-channels that an application may be vulnerable to.

Setting TP chip-wide when any core speculates is simple and conservative. Normal programs do not read time often, if ever, so the performance impact is minimal. More fine-grained management could set TP when (i) any thread on the same core has poisoned data, (ii) the core issues request to shared cache that holds poisoned data, or (iii) the core issues request to memory controller when it is stalling another request with a poisoned address.

Clearing Poison Bits. Once the memory controller has received all outstanding verifications, it will notify the respective structures—*i.e.* all levels of the cache hierarchy, the DTLB, the register file, issue queue, memory controller queue—to clear poison bits. In each cache, including the DTLB, the poison bits are flash cleared. Clearing the poison bits in the LLC implicitly unlocks the lines. The memory controller flash clears poison bits from its request queue, allowing memory requests that have been prevented from accessing memory to proceed.

While LLC cache misses do not happen frequently, waiting for all outstanding verifications to complete might unnecessarily introduce additional delays. If a program has a large memory footprint with many LLC misses, poison spreads and eventually halts processing until verification catches up. To reduce delays from verification, we clear poison bits in batches with an epoch-based policy.

B. Epoch-Based Poison Management

Thus far, we have described PoisonIvy with one poison bit per structure (*e.g.*, one bit per register, one bit per cache line, etc). In such a scheme, all outstanding verifications must be completed before clearing the poison bit. This scheme works well if the program has small, natural bursts of LLC misses, followed by many cache hits—when the burst ends, all requests are verified and speculation is cleared. However, if the program accesses memory uniformly over a longer period, it may provide no natural break for verification to complete—more and more state is poisoned until structures fill, the processor stalls, and verification catches up.

Epochs. PoisonIvy resolves this difficulty by replacing each poison bit with a pair of poison bits—one per *verification epoch*. When the memory controller first returns speculative data, it does so in epoch 0 and sets the first bit in the pair for each affected structure. As more requests are returned

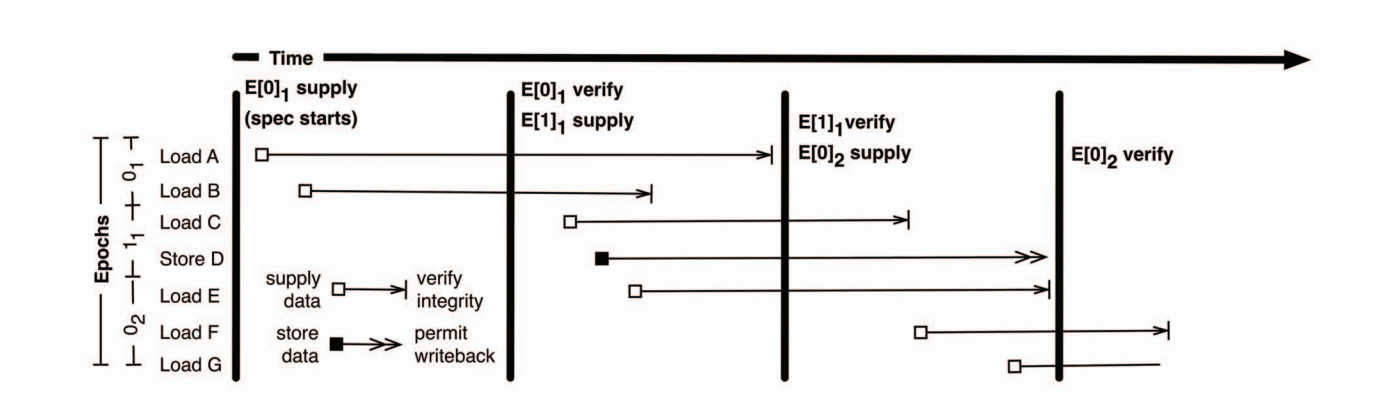

Fig. 5: At any point in time, two epochs are active—one in which data is supplied and another in which verifications are completed. PoisonIvy verifications do not need to complete in the same order that memory requests were generated. They may complete in any order before or after the epoch closes. $E[x]_n$ denotes the nth instance of epoch x. Only when all verifications of epoch $E[0]_i$ have completed, can a new instance of epoch 0, $E[0]_{i+1}$, begin.

speculatively in epoch 0, they set the first bit in each appropriate poison pair. After some time—measured in number of cycles, speculative requests, or other conditions—the memory controller transitions to verification epoch 1, in which new speculative requests set the second poison bit for each affected structure.

While new requests arrive in epoch 1, epoch 0's requests are verified as memory cycles become available for counters, hashes, and tree nodes. No new requests are added to epoch 0 and its number of outstanding verifications decrease over time. When all of epoch 0's requests are verified, the memory controller notifies the processor and clears epoch 0's poison bits in all structures. Note that epoch 1's bits are unaffected.

After epoch 0 completes verification, the memory controller closes epoch 1 to new requests and opens a new instance of epoch 0. To ensure proper semantics, the memory controller waits as long as needed to completely clear poison bits across the processor before initiating the new instance of epoch 0. This small delay does not affect the latency of requests to memory since the controller releases requests as soon as their poison bits are cleared. New memory requests may be performed speculatively in epoch 1 while the clear signals for epoch 0 propagate.

Figure 5 shows how the speculation mechanism pipelines data supply and verification across epochs. Our definition of epochs permits variable durations and allows memory accesses to dictate the cadence of data supply and verify. The first instance of epoch 0, denoted $E[0]_1$, begins when loads A and B supply unverified data to the LLC. When $E[0]_1$'s length exceeds the minimum duration, the first instance of epoch 1, denoted $E[1]_1$, begins. Memory accesses C through E are attributed to $E[1]_1$ until its length exceeds the minimum duration and $E[0]_1$'s data is verified, which starts the second instance of epoch 0, denoted $E[0]_2$. Accesses F and G are attributed to $E[0]_2$. PoisonIvy could support more epochs with correspondingly more poison bits, but we find no significant advantage to having more than two epochs.

Loads can be verified out of order, which gives the memory controller flexibility when scheduling metadata requests. Because PoisonIvy can tolerate very high verification latencies without stalling the pipeline, the memory controller should prioritize demand requests over verification requests. The controller should schedule verification over demand requests only when its verification queues fill. Verification requests affect performance when metadata transfers saturate memory bandwidth, which is rare, or when dependent requests have to wait for verification to complete to go out to memory.

Poisoned Cache Lines. The memory controller supplies unverified data to the cache hierarchy and performs two additional tasks. First, the controller increments a counter for the number of pending verifications in the current epoch. Second, the controller marks the outstanding requests as poisoned by setting the bit corresponding to the current epoch number.

The LLC controller confines poisoned cache lines to the chip. First, poisoned lines cannot be evicted since evictions release unverified data to memory. The eviction policy accommodates this constraint with little performance impact. The processor pipeline stalls only when an LLC set fills with dirty lines during an epoch. In practice, epochs clear within a few thousand cycles and such stalls never occur.

Second, poisoned lines cannot be shared by the cache or inter-socket coherence protocol. Stalling coherence requests typically risks deadlock, but our mechanism poses no such risk—coherence and verification are independent. The memory controller verifies integrity and clears poison bits even if the rest of the system stalls.

Verifying a Load. When the memory controller verifies a load's data integrity, it also decrements the counter associated with the load's epoch. The memory controller consults the outstanding queue entry to identify the epoch.

An epoch ends when its last pending verification is resolved and its counter decrements to zero. The memory controller

Structure	Size	Poison (bits)
Registers	196	392
Issue queue	60	120
dTLB size	64	128
DL1 cache	32KB	1024
L2 cache	256KB	8192
L3 cache	2MB	65536
Mem controller queue	32	64
IW	–	2
CF	–	2
Pipeline latches, etc	–	$\leq$4096
Poison bits	–	**79556** $\approx$ **9.5KB**

TABLE II: Poison Storage

Pipeline width	4 (dispatch+commit), 8 (issue)
ROB size	192
Issue Queue size	60
Store buffer size	42
Load queue size	72
Clock Frequency	3GHz
L1 I & D Cache	32KB 8-way
L2 Cache	256KB 8-way
L3 Cache	2MB 8-way
Memory Size	4GB
Hash Latency	80 processor cycles
Hash Throughput	1 per DRAM cycle

TABLE III: Simulation Configuration

clears the poison bit for that epoch for all outstanding verifications in the queue. When a memory request's poison bits have been cleared, it is also released from the memory controller to proceed off-chip. The memory controller also sends a message up the memory hierarchy and to the core to clear their poison bits. When all poison bits in a LLC line are clear, the controller releases the cache line and permits eviction to memory.

Area Overhead. Adding poison bits for each epoch adds a small amount of area overhead throughout the processor. Table II shows the area breakdown of the different structures that require poison bits. The total additional area is $\approx$9.5KB.

The logic and propagation overhead is very small. Nearly all poison bits are added to regular, dense SRAM structures (registers, caches) in which wire area is proportional to capacity [45]. Poison bits follow instruction/data through existing datapaths, avoiding irregular and area-intensive wires.

IV. Evaluation

We evaluate `PoisonIvy` on a system with counter-mode encryption and BMTs [25]. We use HMAC with SHA-2 for all hashes and truncate the resulting hash to 8B. We use the 8B hash for both tree nodes, which ensure counter integrity, and data hashes, which ensure data integrity. A larger hash would only make speculation more important. The arity and height of the hash tree are eight and nine. We use 7b per-block counters and 64b per-page counters to cover 4KB of data in one 64B block.

Simulators. We combine MARSSx86 and DRAMSim2 for cycle-level, full-system simulation of the processor and memory [23], [28]. Table III summarizes simulation parameters. We use a 2MB LLC to match the per-core LLC capacity seen in commercial muti-core processors (*e.g.*, 8MB for 4 cores). We modify the memory controller model to implement metadata caches. We also implement a timing model to estimate costs of data en/decryption and integrity verification. All experiments include stream prefetching.

Workloads. We evaluate all benchmarks in PARSEC [5], SPLASH2 [46] and SPECCPU2006 [1]. We simulate regions of interest for PARSEC and SPLASH. We fast-forward 1B user instructions for SPEC. For all three suites, we run simulations for 500M user instructions. We present results for memory-intensive benchmarks that have more than 10 misses per thousand of instructions (MPKI) in the LLC. Graphs show results with a 32KB metadata cache unless otherwise specified.

A. Performance from Speculation

Figure 6—and all other figures—shows performance overheads normalized to that of an unsecure system (*i.e.*, neither integrity nor encryption). We compare four security mechanisms:

- **No Speculation.** The memory controller waits for verification before supplying data to the LLC. This inherently safe mechanism is implemented in industrial designs (*i.e.*, Intel's SGX).
- **`Authen-then-write`.** Stores cannot write the L1 cache until all outstanding speculative loads are verified [32]. This mechanism guards against speculative data escaping the core, but does not guard against address-based side-channel attacks.
- **Unsafe Speculation.** The memory controller supplies data to the LLC assuming verification will succeed. No mechanism prevents results of unverified computation from leaving the chip [25], [38].
- **`PoisonIvy`.** Data is supplied unverified and speculative computation is enabled. Poison prevents results of unverified computation from leaving the chip. It also guards against address-based side-channel attacks.

In Figure 6, we show the average overhead for all benchmarks in each suite (parsecAvg, specAvg, and splashAvg). These averages are quite low as they include the many benchmarks in each suite that exhibit few LLC misses. Such benchmarks have inherently low overhead as schemes for memory integrity only add latency to off-chip accesses. To better show the overall trends, we also include the average of 11 memory-intensive benchmarks from the three suites (memAvg).

Comparison to No Speculation. `PoisonIvy` (and the other schemes) significantly outperform No Speculation—generally exhibiting about half as much overhead. Without speculation, verification latencies are exposed on the critical path when returning load data, which prevents dependent instructions from executing and filling the out-of-order window.

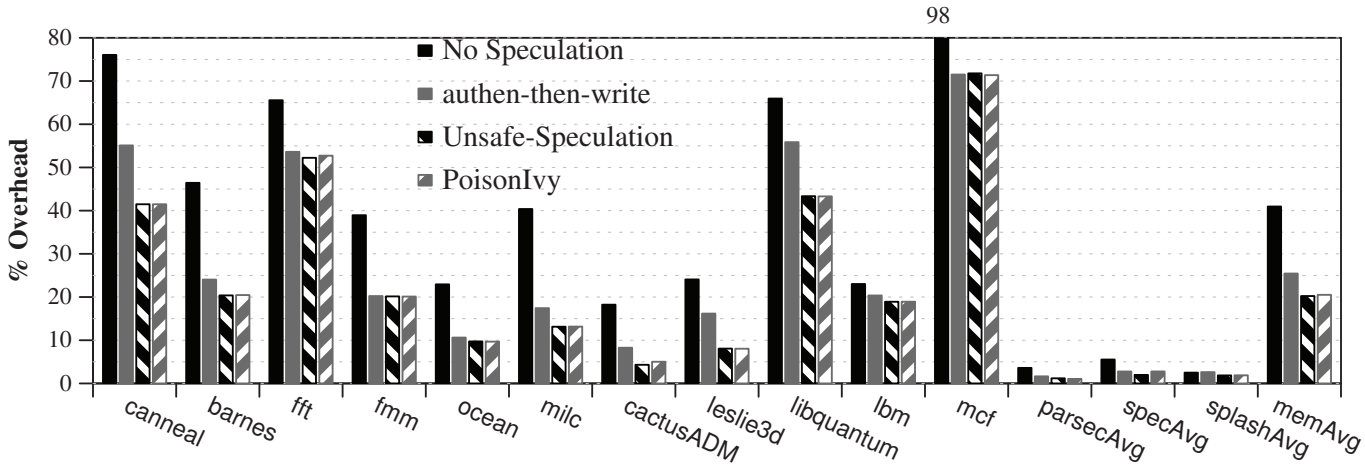

Fig. 6: PoisonIvy hides security overheads. For memory-intensive workloads, it performs as well as unsafe speculation, outperforms no speculation by 20%, and outperforms authen-then-write by 5%. PoisonIvy also guards against address-based side-channel attacks.

Comparison to Authen-then-write. Although authen-then-write performs much better than No Speculation, PoisonIvy outperforms or matches it. Benefits are most notable for *canneal* and *libquantum*. PoisonIvy's performance advantage comes from the fact that authen-then-write is constrained by the store buffer capacity during speculation. After the memory controller responds to an LLC miss with unverified data, the core can execute and commit instructions. However, the next store cannot complete and modify the L1 cache until verification completes. Waiting stores quickly fill the store buffer and stall instruction dispatch.

By the time metadata returns and integrity is verified, the datapath has been waiting for hundreds of cycles and the out-of-order window has filled. Because of the limited capacity of the store buffer, the datapath has few opportunities to continue computation when verification latencies are long. Latencies are at least 80 processor cycles when hashing data to check integrity, assuming hash and counter are cached, and are much higher when loading metadata from main memory.

In contrast, PoisonIvy permits far more computation during integrity verification by allowing stores to commit data to the L1. Dirty blocks are poisoned and can escape to the L2 and LLC but cannot leave the secure chip. The pipeline only stalls when the LLC must stall eviction due to poisoned dirty lines. PoisonIvy not only outperforms authen-then-write, it also improves security by guarding against address-based side-channel attacks, which were presented in that study [32].

Comparison to Unsafe Speculation. Unsafe Speculation, which does nothing to restrict computation on unverified data, performs best but is least secure. Figure 6 shows that PoisonIvy generally matches this best-case performance. In effect, our system has the performance of unsafe speculation while guaranteeing security. A few benchmarks experience trivially higher overheads ($< 1\%$) than unsafe speculation.

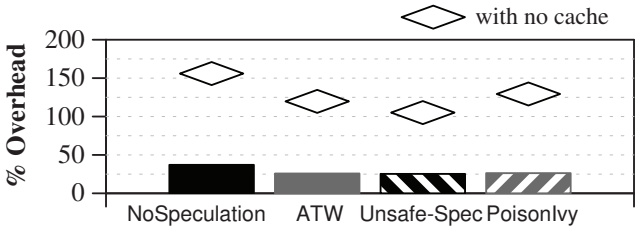

Fig. 7: Performance comparison for a pointer chasing μbenchmark with and without a metadata cache.

Surprisingly, PoisonIvy performs as well as a system with unsafe speculation for *mcf*, which is known for pointer chasing. This result arises from the fact that many metadata requests hit the cache, resulting in short verification latencies—shorter than the time required for a load to return data to the core, flow through dependent instructions to another dependent load and for that dependent load's miss to reach the memory controller.

For further insight, we implement a pointer chasing microbenchmark—which does no computation with the output of a load beyond using it as the input to the next load—and evaluate all three systems without metadata caching. Figure 7 shows the result of this experiment with and without a metadata cache. With a metadata cache, the results are similar to what was observed for *mcf*. However, without a metadata cache, PoisonIvy performs 24% worse than unsafe speculation and 10% worse than authen-then-write. authen-then-write performs well because the microbenchmark is dominated by loads and has no stores, thereby avoiding pressure on its bottleneck, the store buffer. In contrast, PoisonIvy performs well for normal workloads that mix loads and stores. Thus, PoisonIvy performs well but pays the price for security when given challenging memory accesses and no metadata cache.

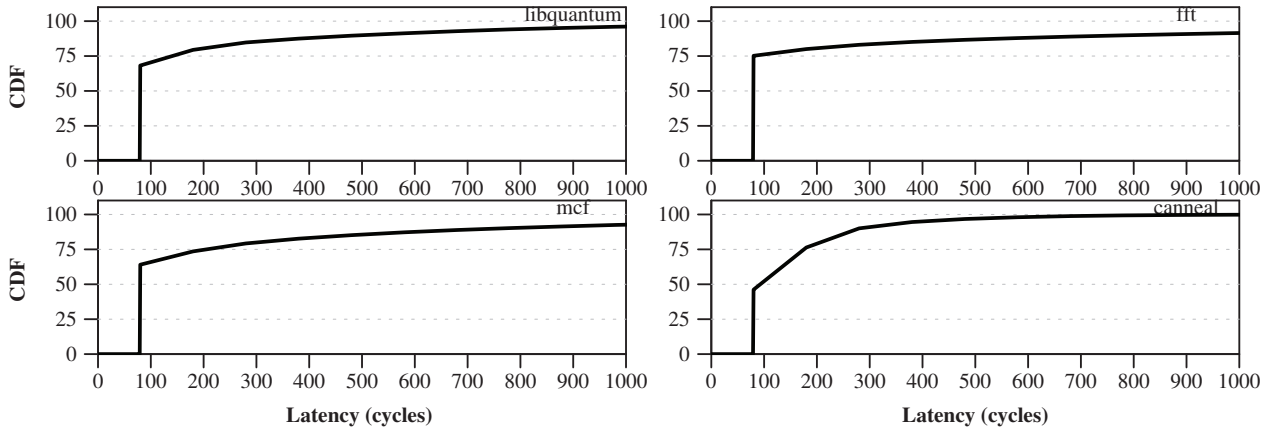

Fig. 8: Cumulative distribution function (CDF) of verification latency from cache line fill to verification. Minimum and average latencies are 80 and 273 cycles.

B. Verification Latency

To further understand the performance characteristics of these schemes, Figure 8 shows the cumulative distribution functions (CDFs) for verification latency when filling a cache line from main memory. The minimum value is the hash latency—80 processor cycles in our study. Many requests are verified in this short latency, indicating that the required metadata is in the memory controller's dedicated cache. For example, in *mcf*, 64% of requests require only this minimum latency due to cache hits, leading to the behavior described above.

These short latencies are quite easy to speculate around and rarely impact dependent memory requests. When a memory request hits in the metadata cache, its verification latency is hidden by the time it takes to supply data to the core, execute dependent instructions, and issue another memory request through the cache hierarchy to the memory controller. By the time a dependent request reaches the memory controller, verification is almost finished and the dependent request stalls for a very short time, if at all.

Although the minimum latency is modest, the CDFs reveal large means and long tails. Average latency is approximately 273 cycles, much too long for a processor to cover in its store buffer (*e.g.*, with `authen-then-write`). Furthermore, the distributions have long tails with verifications that require as many as 3000 cycles, arising from multiple DRAM requests to retrieve metadata that are delayed behind higher priority data requests. These long tails motivate `PoisonIvy`'s epoch-based speculation.

C. Hash Latency

Figure 9 evaluates sensitivity to hashing latency. Without speculation, the hashing latency is on the critical path when returning data to the core, even when metadata is in the memory controller's cache; a non-speculative design is quite sensitive to hash latency. In contrast, `PoisonIvy` removes the hashing latency from the critical path to service a memory request. As hashing latency decreases, so do benefits from speculation. But even at the lowest latency that we examined, 10 processor cycles, speculation reduces performance overheads from security from 28% down to 20%, on average, for the memory-intensive benchmarks.

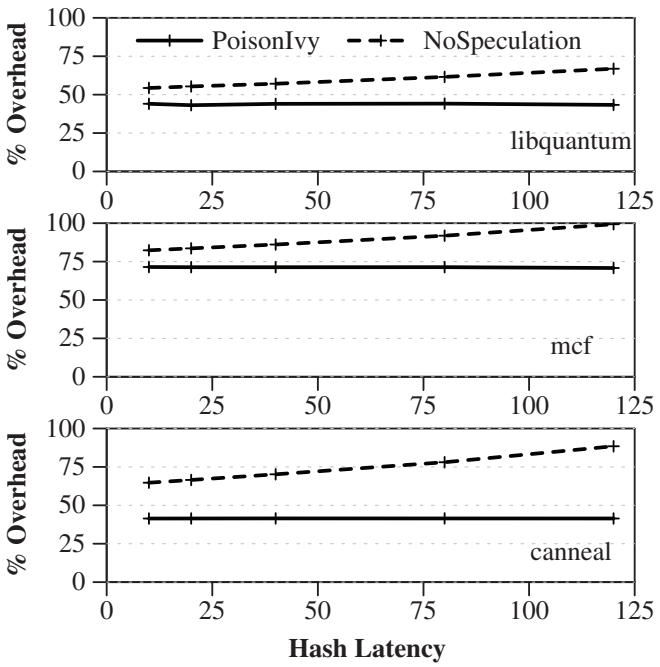

Fig. 9: Performance sensitivity to hash latency with and without `PoisonIvy`. `PoisonIvy` removes hashing from the critical path and its performance is insensitive to hashing latency.

D. Metadata Cache Size

Metadata caches matter for two reasons. First, with speculation, verification latency determines for how long structures are poisoned. Metadata caching reduces verification latency as requests for counters, tree nodes, and hashes might avoid main

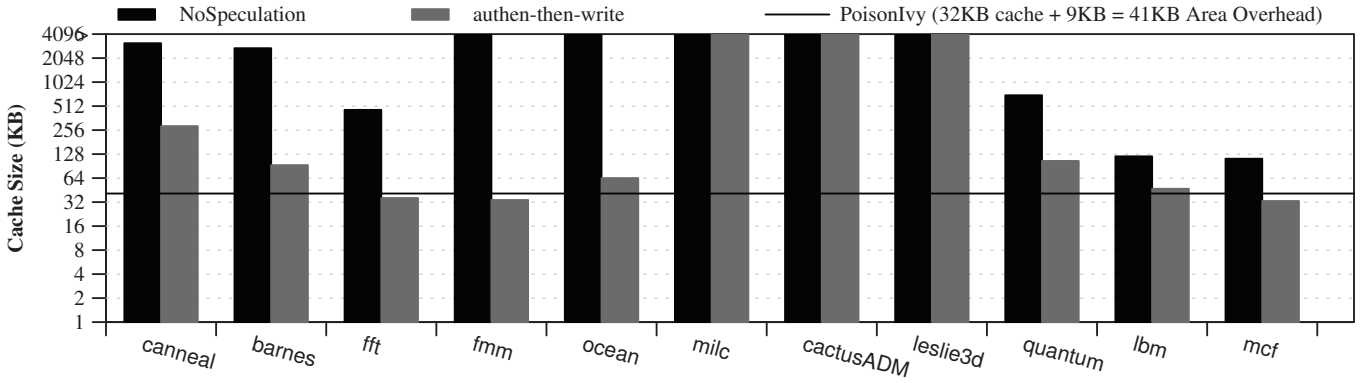

Fig. 10: Cache sizes needed for NoSpeculation to match the performance of PoisonIvy with 32KB metadata cache. The horizontal line shows storage for PoisonIvy (32KB cache + poison bits). Note the log-scale y-axis.

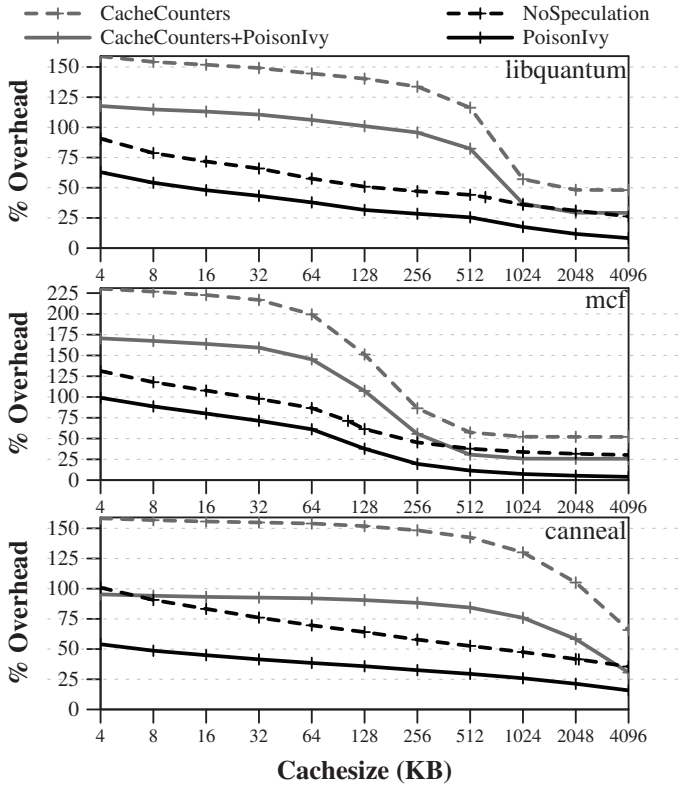

Fig. 11: Comparison of PoisonIvy and NoSpeculation when caching counters versus all metadata types. Speculation's advantage increases as metadata cache size decreases. Regardless of cache size PoisonIvy outperforms all other configurations by at least 15%. Note the log-scale x-axis.

memory. Second, metadata caching might be an alternative to speculation if area overheads do not matter.

Exploring the effects of metadata cache size allows us to consider an important question—how does metadata caching alone compare to PoisonIvy when poison storage is included? As poison bits are added to many large structures, they require a noticeable amount of storage—about 9.5KB in the system described by Table II.

Figure 10 shows the metadata cache size required before NoSpeculation and authen-then-write match PoisonIvys performance with a 32KB metadata cache. The horizontal line shows PoisonIvys total storage requirements—41.5KB of which 32KB is metadata cache and 9.5KB is poison information.

A system with no speculation always requires much more storage to match PoisonIvy's performance. For *canneal*, a system without speculation requires 2048KB of metadata caching for PoisonIvy's performance. In the best case, for *lbm*, twice as much storage is required. For more than six benchmarks, more than 4MB of storage is required to perform as well as PoisonIvy's use of 41.5KB.

authen-then-write's performance comes much closer but still requires metadata caches that are much larger than PoisonIvy total storage overheads. For authen-then-write, three-quarters of the benchmarks require at least twice the storage to match PoisonIvy's performance. The remaining benchmarks—*fft, fmm, lbm and mcf*—can match PoisonIvy's performance with a 32KB metadata cache. It is important to note that while authen-then-write performs well for a few benchmarks, it does not protect against address-based side-channel attacks. PoisonIvy provides more complete security guarantees.

Figure 11 shows sensitivity to metadata cache design, presenting performance overheads under No Speculation and PoisonIvy while varying the metadata cache size and its contents; caches may hold only counters or all metadata. Although prior work caches only counters [25], we find caching all metadata types is worthwhile. For an average of memory-intensive benchmarks, holding all types of metadata in a 32KB cache reduces PoisonIvy's overheads from 64% down to 20%.

As would be expected, performance overheads decrease with larger cache sizes. Overheads significantly and rapidly drop off when the cache accommodates the metadata working set. As the cache size increases and the latency of verifying data integrity decreases, speculation becomes less important. However, speculation with PoisonIvy provides significant

benefits even with a 4MB metadata cache.

V. Related Work

System Software. When multiple users share resources, one could exploit vulnerabilities in the hypervisor to uncover secret data [24]. HyperWall controls the hypervisor's memory accesses [39]. NoHype removes the hypervisor from the software stack and relies on existing processor features to perform virtualization tasks [15]. HyperCoffer [47] introduces a virtual machine shadow that implements encryption for privacy and hash trees for integrity. In contrast to these system solutions, SGX and ISO-x extend the instruction set to isolate an application from others [3], [10]. These mechanisms are orthogonal to `PoisonIvy`, which applies whenever encryption and trees are used for privacy and integrity.

Taint Tracking. Information flow security—also known as taint—tracks untrusted data whereas poison tracks microarchitectural effects of latency tolerance schemes (CFP, etc). Information flow security has been studied extensively at all levels of the abstraction stack: from applications [8], [2], [16], [43], [44], to OS/privileged software [21], and even to logic gates [40], [42], [22], [14], [41], [17]. Most of these systems track information flow to prevent code execution (*e.g.*, branches) that could leak secret information. Gate Level Information Flow Tracking (GLIFT) uses taint tags for each bit at the gate level to track the information flow through the system [14].

The main difference between taint and poison used in `PoisonIvy` is that the latter hides verification latency for memory integrity (*e.g.*, SGX). This difference matters. `PoisonIvy` (i) halts the system when verification fails and cannot use software to handle security exceptions; (ii) guards against physical attacks (*e.g.*, memory tampering) instead of unsafe application data (*e.g.*, SQL injection); (iii) distrusts memory and cannot write poison/taint tags to DRAM since unprotected tags break security and protected tags negate performance from speculation; (iv) uses poison only to restrict unverified data to chip and cannot exploit programmable propagation/checks.

VI. Conclusions

A trusted processor employs encryption and integrity trees to guard against physical attacks. We propose a new security architecture that ensures security yet significantly lowers performance overheads. We architect a speculation mechanism that uses poison to track speculative data and addresses throughout the processor and memory hierarchy. An epoch-based management policy allows the core to compute on speculative data while clearing speculation in batches. The last-level cache prevents data from leaving the processor before it is verified. Poison tracks speculative dependencies throughout the processor (*i.e.*, core, register, IW, CF and DTLB) and ensures that no memory operations are sent across the memory's command and address buses before speculative data has been verified. `PoisonIvy` reduces performance overheads of security to 1.8%, on average.

VII. Acknowledgements

We would like to thank the anonymous reviewers for their thoughtful comments and suggestions. We would also like to extend a special thanks to Carlos Rozas, Siddhartha Chhabra, Frank Mckeen, and other members of Intel Labs, and Brian Rogers for their valuable discussions and insights.

This work is supported by the National Science Foundation under grants CCF-1149252 (CAREER), CCF-1337215 (XPS-CLCCA), SHF-1527610, and AF-1408784. This work is also supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these sponsors.

References

[1] Standard Performance Evaluation Corporation (SPEC) CPU^{TM} 2006 https://www.spec.org/cpu2006/.

[2] "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones, author=Enck, William and Gilbert, Peter and Han, Seungyeop and Tendulkar, Vasant and Chun, Byung-Gon and Cox, Landon P and Jung, Jaeyeon and McDaniel, Patrick and Sheth, Anmol N, journal=Transactions on Computer Systems (TOCS), pages=5, year=2014, publisher=ACM."

[3] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013, p. 10.

[4] I. Anati, F. Mckeen, S. Gueron, H. Haitao, S. Johnson, R. Leslie-Hurd, H. Patil, C. Rozas, and H. Shafi, "Intel software guard extensions (Intel SGX)," in *Tutorial at International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2015.

[5] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[6] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, "SecureME: a hardware-software approach to full system security," in *International Conference on Supercomputing*. ACM, 2011, pp. 108–119.

[7] V. Costan and S. Devadas, "Intel SGX explained," Cryptology ePrint Archive, Report 2016/086, 2016., Tech. Rep.

[8] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *International Symposium on Computer Architecture (ISCA)*. ACM, 2007, pp. 482–493.

[9] R. Elbaz, D. Champagne, R. Lee, L. Torres, G. Sassatelli, and P. Guillemin, "TEC-Tree: A low cost, parallelizable tree for efficient defense against memory replay attacks," in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2007, pp. 289–302.

[10] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-x: A flexible architecture for hardware-managed isolated execution," in *International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2014, pp. 190–202.

[11] B. Gassend, G. E. Suh, D. Clarke, M. Van Dijk, and S. Devadas, "Caches and hash trees for efficient memory integrity verification," in *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2003, pp. 295–306.

[12] S. Gueron, "A memory encryption engine suitable for general purpose processors," *International Association for Cryptologic Research (IACR)*, 2016.

[13] S. Gulley and V. Gopal, "Haswell cryptographic performance," *Intel Corporation*, 2013.

[14] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "Theoretical fundamentals of gate level information flow tracking," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1128–1140, 2011.

[15] E. Keller, J. Szefer, J. Rexford, and R. B. Lee, "NoHype: virtualized cloud infrastructure without the virtualization," in *International Symposium on Computer Architecture (ISCA)*. ACM, 2010, pp. 350–361.

[16] C. Kerschbaumer, E. Hennigan, P. Larsen, S. Brunthaler, and M. Franz, "Crowdflow: Efficient information flow security," in *Information Security*. Springer, 2015, pp. 321–337.

[17] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2011, pp. 109–120.

[18] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, pp. 168–177, 2000.

[19] D. Lie, C. A. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *Symposium on Operating Systems Principles (SOSP)*. ACM, 2003, pp. 178–192.

[20] R. C. Merkle, "Protocols for public key cryptosystems." in *Symposium on Security and Privacy (SP)*, 1980, pp. 122–134.

[21] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein, "seL4: from general purpose to a proof of information flow enforcement," in *Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2013, pp. 415–429.

[22] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in I2C and USB," in *Design Automation Conference (DAC)*. IEEE Computer Society, 2011, pp. 254–259.

[23] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A full system simulator for x86 CPUs," in *Design Automation Conference (DAC)*. IEEE Computer Society, 2011, pp. 1050–1055.

[24] D. Perez-Botero, J. Szefer, and R. B. Lee, "Characterizing hypervisor vulnerabilities in cloud computing servers," in *International Workshop on Security in Cloud Computing (SCC)*. ACM, 2013, pp. 3–10.

[25] B. Rogers, S. Chhabra, M. Prvulovic, and Y. Solihin, "Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly," in *International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2007, pp. 183–196.

[26] B. Rogers, M. Prvulovic, and Y. Solihin, "Efficient data protection for distributed shared memory multiprocessors," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, 2006, pp. 84–94.

[27] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-level integrity and confidentiality protection for distributed shared memory multiprocessors," in *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2008, pp. 161–172.

[28] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *Computer Architecture Letters*, pp. 16–19, 2011.

[29] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler, "Late-binding: enabling unordered load-store queues," in *International Symposium on Computer Architecture (ISCA)*. ACM, 2007, pp. 347–357.

[30] T. Sha, M. M. Martin, and A. Roth, "Scalable store-load forwarding via store queue index prediction," in *International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2005, pp. 159–170.

[31] ——, "Nosq: Store-load communication without a store queue," in *International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2006, pp. 285–296.

[32] W. Shi and H.-H. S. Lee, "Authentication control point and its implications for secure processor design," in *International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2006, pp. 103–112.

[33] W. Shi, H.-H. S. Lee, M. Ghosh, and C. Lu, "Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, 2004, pp. 123–134.

[34] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva, "High efficiency counter mode security architecture via prediction and precomputation," in *International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2005, pp. 14–24.

[35] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 107–119, 2004.

[36] S. Subramaniam and G. H. Loh, "Fire-and-forget: Load/store scheduling with no store queue at all," in *International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2006, pp. 273–284.

[37] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2003, p. 339.

[38] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "AEGIS: architecture for tamper-evident and tamper-resistant processing," in *International Conference on Supercomputing (ICS)*. ACM, 2003, pp. 160–171.

[39] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 437–450, 2012.

[40] M. Tiwari, X. Li, H. M. Wassel, F. T. Chong, and T. Sherwood, "Execution leases: A hardware-supported mechanism for enforcing strong non-interference," in *International Symposium on Microarchitecture (MICRO)*. ACM, 2009, pp. 493–504.

[41] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security," in *International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 2011, pp. 189–199.

[42] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2009, pp. 109–120.

[43] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2008, pp. 173–184.

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

[45] N. Weste, D. Harris, and A. Banerjee, "CMOS VLSI design," *A circuits and systems perspective*, pp. 59–60, 2005.

[46] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *International Symposium on Computer Architecture (ISCA)*. ACM, 1995, pp. 24–36.

[47] Y. Xia, Y. Liu, and H. Chen, "Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks," in *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2013, pp. 246–257.

[48] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin, "Improving cost, performance, and security of memory encryption and authentication," *International Symposium on Computer Architecture (ISCA)*, pp. 179–190, 2006.

[49] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in *International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, 2003, pp. 351–360.

[50] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta, "SENSS: Security enhancement to symmetric shared memory multiprocessors," in *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2005, pp. 352–362.