DAGguise: Mitigating Memory Timing Side Channels

Peter W. Deutsch[*]
MIT
Cambridge, MA, USA
pwd@mit.edu

Yuheng Yang[*]
MIT
Cambridge, MA, USA
yuhengy@mit.edu

Thomas Bourgeat
MIT
Cambridge, MA, USA
bthom@mit.edu

Jules Drean
MIT
Cambridge, MA, USA
drean@mit.edu

Joel S. Emer
MIT/NVIDIA
Cambridge, MA, USA
jsemer@mit.edu

Mengjia Yan
MIT
Cambridge, MA, USA
mengjiay@mit.edu

ABSTRACT

This paper studies the mitigation of memory timing side channels, where attackers utilize contention within DRAM controllers to infer a victim's secrets. Already practical, this class of channels poses an important challenge to secure computing in shared memory environments.

Existing state-of-the-art memory timing side channel mitigations have several key performance and security limitations. Prior schemes require onerous static bandwidth partitioning, extensive profiling phases, or simply fail to protect against attacks which exploit fine-grained timing and bank information.

We present DAGguise, a defense mechanism which fully protects against memory timing side channels while allowing for dynamic traffic contention in order to achieve good performance. DAGguise utilizes a novel abstract memory access representation, the Directed Acyclic Request Graph (*r*DAG for short), to model memory access patterns which experience contention. DAGguise shapes a victim's memory access patterns according to a publicly known *r*DAG obtained through a lightweight profiling stage, completely eliminating information leakage.

We formally verify the security of DAGguise, proving that it maintains strong security guarantees. Moreover, by allowing dynamic traffic contention, DAGguise achieves a 12% overall system speedup relative to Fixed Service, which is the state-of-the-art mitigation mechanism, with up to a 20% relative speedup for co-located applications which do not require protection. We further claim that the principles of DAGguise can be generalized to protect against other types of scheduler-based timing side channels, such as those targeting on-chip networks, or functional units in SMT cores.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

[*]Both authors contributed equally to this research.

KEYWORDS

Security, Timing side channels, Directed acyclic graphs, Memory traffic shaping

ACM Reference Format:
Peter W. Deutsch, Yuheng Yang, Thomas Bourgeat, Jules Drean, Joel S. Emer, and Mengjia Yan. 2022. DAGguise: Mitigating Memory Timing Side Channels. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3503222.3507747

1 INTRODUCTION

Side channel attacks, a class of attacks that exploits microarchitectural vulnerabilities to breach system security, have become a serious security threat in recent years. An attacker can use such vulnerabilities to steal secrets from a victim by monitoring the side effects of the victim's actions on various microarchitectural structures [27], including caches [7, 18, 19, 32], branch predictors [1, 8], on-chip networks [30], and memory controllers [29].

In this paper, we focus on studying memory timing side channels which exploit shared memory controllers, a broad attack surface. These attacks are practical and highly effective, as memory controllers are typically shared by all cores on a machine. For instance, Wang et al. [29] have demonstrated that contention on memory buses can be used to extract RSA keys. Pessl et al. [22] have further shown that row-buffer contention can be used to monitor keystrokes and recover user passwords.

In general, side channel attacks can be viewed via a communication model where there is a *transmitter* (the victim) that modulates a *channel*, with that modulation being detected by a *receiver* (the attacker) [5, 14]. When the channel is a cache, the receiver is generally *active*, i.e., it modulates the channel *itself* in order to detect a transmission. In cache-based channels, this involves preconditioning the channel prior to transmission to detect the modulation, e.g., using Prime+Probe [18, 20]. In a memory timing side channel the channel is a memory controller, and the modulation by the transmitter is memory requests based on secret values that make the memory controller busy. In this case, the active receiver must *concurrently* modulate the channel (opposed to preconditioning it) by emitting memory requests to try to contend with the transmitter's requests. Due to memory queuing and scheduling delays, the latency of the receiver's requests can be affected by the transmitter's traffic patterns. Therefore, the receiver can use the timing information of its *own* memory requests to infer the transmitter's secret.

1.1 Mitigation Challenges

Prior work has struggled to efficiently mitigate these attacks, broadly exploring two directions. The first approach has been to completely block interference between memory requests emitted by different applications using partitioning techniques such as Temporal Partitioning (*TP*) [29] and Fixed Service (*FS*) [25]. While secure, such approaches incur high performance overheads as they statically partition memory bandwidth across applications and allocated bandwidth can often go under-utilized.

The second explored approach has been to shape the transmitter's requests into a predefined pattern that is independent from the secret, such as demonstrated in Camouflage [36]. Camouflage leverages offline profiling to obtain the *distribution* of timing distances between consecutive memory requests, and then shapes request patterns on-the-fly so that the distance between consecutive requests follows this predefined distribution. Although Camouflage can achieve better performance than Temporal Partitioning [29] and Fixed Service [25], it does not offer the same level of security. As described by its authors [36], Camouflage was designed to hide coarse-grained timing information and only provides security when the attacker's timer resolution is low. Camouflage is also severely limited in its flexibility, requiring prior knowledge of co-running applications' bandwidth requirements during profiling to determine an optimal shaping distribution.

We observe that no prior work can simultaneously meet the following criteria for an *optimal* secure memory controller:

(1) *Security:* Completely blocking information leakage to an attacker that uses the latency of its own memory requests to infer a secret.
(2) *Limited Performance Overhead:* Allowing a dynamic allocation of bandwidth across applications.
(3) *Low Profiling Costs:* Requiring, at most, a simplistic profiling step which does not need prior knowledge of co-located applications.

1.2 Our Proposal: *r*DAGs and DAGguise

We introduce *DAGguise*, an effective defense mechanism that simultaneously satisfies the three requirements above. To accomplish this, DAGguise shapes requests according to a novel memory request representation, the *Directed Acyclic Request Graph* (*rDAG* for short).

The Directed Acyclic Request Graph Representation. In an *r*DAG, each vertex represents a memory request which experiences an unknown amount of contention in the memory controller. An edge between two vertices indicates the existence of a timing dependency between the two requests, i.e., the destination request can only be emitted by the core after the source request completes. If there exists no path between two vertices, that implies that the corresponding requests can be emitted in parallel.

As a representation of memory request patterns, *r*DAGs have two appealing properties: *generality* and *versatility*. *r*DAGs are *general* enough to fully describe any fine-grained request pattern, describing the distances between consecutive requests, timing dependencies between requests, and the requests' memory-level parallelism. Moreover, rather than being a constant representation of memory request timing, *r*DAGs are *versatile*, being able to accommodate unknown latencies within the memory controller. Specifically, when a request in an *r*DAG is delayed due to memory contention, its dependent requests are also delayed. We fully describe the structure and properties of *r*DAGs in Section 4.1.

Shaping Memory Requests Using *r*DAGs. The key idea of DAGguise is to shape memory requests into a pre-defined pattern described using an *r*DAG, which we call a defense *r*DAG. Specifically, DAGguise introduces a request shaper between the transmitter and the memory controller. The request shaper works as a proxy agent of the transmitter and disguises the transmitter's request patterns. The shaper buffers requests from the transmitter and emits requests following the timing dependencies described by the defense *r*DAG by either delaying some requests or emitting fake requests. At the cycle when the defense *r*DAG prescribes the need to emit a request, the shaper checks whether any request from the transmitter has been buffered. If such a request exists, that request is sent, otherwise a fake request is generated to maintain conformity with the defense *r*DAG and preserve security.

To achieve better performance, DAGguise is assisted with an offline profiling phase which aims to generate a defense *r*DAG that can match the bandwidth requirements of the program to be protected. The profiling is lightweight and is performed on the transmitter in isolation, without requiring the need to account for any co-running applications.

DAGguise satisfies our three secure memory controller requirements, thanks to the generality and versatility of *r*DAGs. First, DAGguise securely hides memory request patterns, as the memory requests emitted by the shaper are fully dependent on the defense *r*DAG, and completely independent from the transmitter's original request patterns. Since the defense *r*DAG is not dependent on any secrets, even if the receiver can fully reconstruct the defense *r*DAG, it cannot glean any information from the transmitter.

Second, DAGguise can achieve better performance than TP [29] and FS [25]. Rather than statically allocating bandwidth between applications, DAGguise allows the memory controller to *dynamically* adjust the bandwidth allocation between the shaped requests and co-running applications. For instance, when a co-running application emits an increased number of requests, the requests from the defense *r*DAG suffer from more contention. As these requests and the subsequent requests which are dependent on them are delayed in turn, the shaper's bandwidth utilization naturally reduces accordingly.

Lastly, DAGguise's offline profiling cost is low. Since *r*DAGs are versatile, we only need to independently profile the transmitter to derive a defense *r*DAG which achieves good performance, requiring a far smaller profiling cost compared to Camouflage [36].

We use Rosette [28] to formally verify the security properties of DAGguise, ensuring that the shaped access patterns of a transmitter are *indistinguishable* to any receiver. We evaluate the performance overhead of DAGguise using gem5 [4] and run SPEC benchmarks alongside two security-sensitive applications, DocDist [11] and DNA sequencing [24]. Our results show that DAGguise introduces considerably less performance overhead compared to FS [25]. DAGguise incurs a 10% system slowdown on a two-core system, improving system-wide performance by 6% compared to Fixed Service [25].

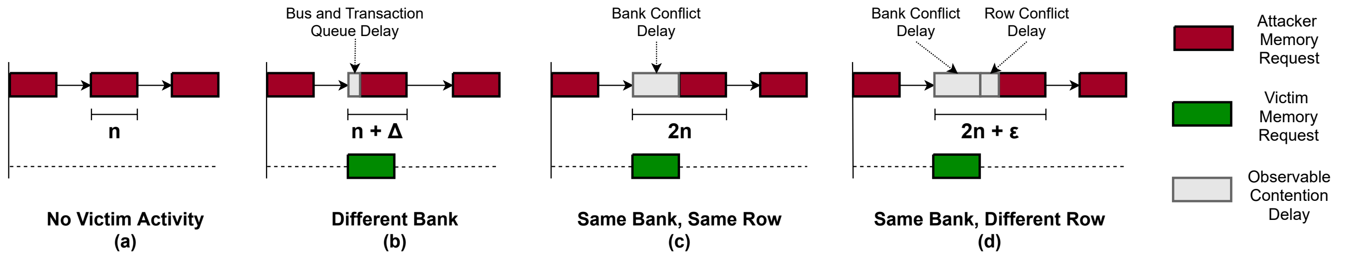

Figure 1: An example of memory timing side channels which exploit different types of memory contention. An attacker can discern a victim's detailed memory request patterns based on the latency of its own requests.

We show that DAGguise also scales well compared to Fixed Service, achieving a 12% performance speedup compared to FS on an eight-core machine. Furthermore, DAGguise is area efficient, requiring only 0.037mm^2 of area to instantiate eight parallel shaper instances.

Note that, while we focus on utilizing DAGguise to mitigate memory timing side channels, the key insights of DAGguise are generalizable. *r*DAGs are a general representation of request patterns for various microarchitectural structures. The principles of DAGguise can be applied to mitigate other types of timing side channels involving schedulers and queues, such as instruction port contention in SMT cores [2].

Contributions. This paper makes the following contributions:

- The introduction of the Directed Acyclic Request Graph (*r*DAG) representation, a generalizable and versatile way to describe memory access patterns.
- The design of an effective and performant defense mechanism, DAGguise, demonstrating that it is possible to exceed the performance of state-of-the-art defenses, i.e., fine-grained static temporal traffic partitioning [25], while preserving the same security guarantees.
- A formal analysis of the security properties of DAGguise using Rosette [28], a solver-aided programming framework.
- A detailed performance and area evaluation of DAGguise, demonstrating a 12% speedup over Fixed Service [25] with an area footprint of only 0.037mm^2.

2 BACKGROUND

2.1 Memory Basics

In modern computing systems, processors access the main memory system via one or more memory controllers (MCs). A memory controller manages a memory *channel*, organized hierarchically into *ranks* and *banks* [13]. Each memory channel supports multiple ranks, where a rank is a collection of DRAM chips that work in parallel to handle a memory request, e.g., to fill a cache line. Each rank is partitioned into multiple banks. Banks and ranks help support multiple outstanding requests, thus enabling a high degree of parallelism in the memory system. Each bank contains a row-buffer, which caches the data of the most recent request. Under an open-row policy, temporally adjacent accesses to the same DRAM row can hit in the row-buffer. Conversely, a closed-row policy forbids hits in the row-buffer.

Memory requests can interfere with each other's timing at several points within the memory controller [13]. Upon arrival from the last-level cache, memory requests are first buffered in a *transaction queue*. Then, each memory request is converted to a sequence of DRAM commands, which are placed into a *command queue* based on their addresses. These command queues are arranged such that there is one queue per bank or per rank of memory. Finally, depending on the DRAM command scheduling policy, commands are scheduled to the DRAM devices based on resource availability and timing constraints. Command scheduling can vary in complexity, ranging from a basic First Come First Served (FCFS) policy, to policies that optimize for row-buffer hits or bus direction switches (i.e. by grouping reads and writes together) [13].

2.2 Memory Timing Side Channels

Shared memory controllers expose a large attack surface for timing side channel attacks. Compared to pipeline structures and private caches, memory controllers are shared by all processes and virtual machines running on the same chip. Several attacks have already shown the viability of memory timing side channels [22, 29].

A memory timing side channel involves a victim (transmitter) program and an attacker (receiver) program communicating via contention within a memory controller. The transmitter in the victim's security domain emits a sequence of memory requests based on some secret values. The receiver in the attacker's security domain aims to obtain the secret by monitoring the transmitter's memory access pattern. While the attacker cannot *directly* observe the victim's access pattern, it can emit a sequence of memory requests and observe how the victim's accesses interfere with its own as they contend with each other in the shared memory controller.

An Attack Example. Figure 1 provides an example of how an attacker can discern a victim's detailed request patterns based on the latency of its own requests. In this example, the attacker always follows the same request pattern, emitting a new request a constant amount of time after the previous request completes. The attacker's requests are always mapped to the same bank and the same row. We consider a simplified memory where each request takes *n* cycles to service and the DRAM uses an open-row policy.

In Figure 1(a), when the victim does not emit any requests, none of the attacker's requests are delayed. In Figure 1(b), when the victim emits a request targeting a different bank from the attacker's requests, one of the attacker's requests is delayed for Δ cycles due to contention in the transaction queue and the shared memory bus. In

Figure 1(c), when the victim emits a request to the same bank and the same row as the attacker's requests, one of the attacker's requests is delayed until the victim's request completes. As a result, the attacker observes a request latency of $2n$ cycles. In Figure 1(d), when the victim's request targets the same bank but a different row compared to the attacker's requests, one of the attacker's requests will suffer an additional ϵ cycle penalty, i.e., the time for the memory controller to close the current row and open a new row. As demonstrated, an attacker can use its own latency to effectively discern a victim's request patterns, including the number of memory requests, the timing of these requests, as well as their bank and row address information.

2.3 Threat Model

We assume the attacker and the victim are in different security domains and the attacker cannot directly access the victim's secret data. The attacker and the victim run on the same machine accessing DRAM via one or more shared memory controllers. The attacker can be either a user-level application or privileged system software, where the latter case applies to a system with support for enclaves, such as Intel SGX [12], Sanctum [6], or Keystone [17].

The attacker performs a memory timing side channel attack to glean secrets from the victim's domain. As described in Section 2.2, the attacker actively generates requests to interfere with the victim's memory requests and aims to infer the victim's memory request patterns based on its own response latencies. We do not consider physical attacks, that is, where the attacker physically accesses and probes the DRAM bus to *directly* observe the timing, addresses, or even data of memory requests. Such attacks require the attacker to physically possess the attacked device. Similar to other existing defense mechanisms [25, 29, 36], we do not block information leakage due to early termination time. Termination time leakage is intrinsically a program-level issue, and cannot be effectively addressed at the microarchitectural level.

We consider a defense mechanism to be secure if no attacker can distinguish a transmitter's memory request patterns. The memory latencies observed by the attacker should be independent from the victim's actual memory activity. A formal definition of the *indistinguishability* property is provided in Section 5.

3 MOTIVATION

3.1 Limitations of Existing Approaches

There exists two directions in mitigating memory timing side channels: partitioning and traffic shaping. We observe that existing mitigation mechanisms [25, 29, 36] along these two directions suffer from several key limitations. In this section, we examine two state-of-the-art defense mechanisms, Fixed Service [25] and Camouflage [36].

Fixed Service. Fixed Service (FS) [25] achieves static and fine-grained temporal partitioning by introducing a deterministic schedule for memory requests. Every request is assigned to a certain "slot". Within each slot, a request sequentially passes through the request queues, the command bus, the bank, and the data bus. The slots are pipelined, with a fixed stride inserted between consecutive slots to ensure that each in-flight request uses different resources at any point of time. Therefore, no collisions can occur in any of the shared microarchitectural resources.

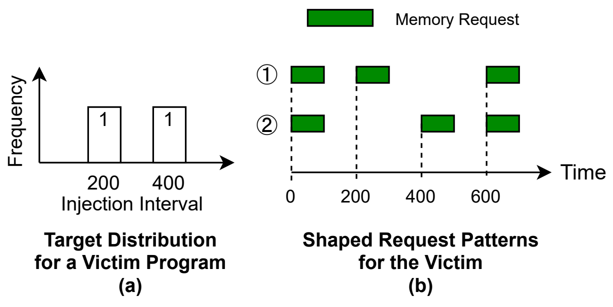
Memory Request
Frequency
1
1
200
400
Injection Interval
Target Distribution for a Victim Program
(a)
①
②
Time
0
200
400
600
Shaped Request Patterns for the Victim
(b)

Figure 2: A demonstration of how Camouflage cannot hide fine-grained request patterns.

The memory controller assigns slots to different security domains using a *round-robin, no-skip* arbitration policy. If a security domain does not have a pending request for its slot, the slot is wasted. This strict partitioning approach completely isolates the memory access patterns of security domains from one another and achieves a strong non-interference property.

Fixed Service's strict partitioning can often significantly degrade bandwidth utilization, incurring a high performance overhead. For instance, if there are N security domains, one bandwidth-intensive domain and the remaining idling, the memory-intensive domain will only be able to utilize $1/N$ of the total bandwidth, with the remaining bandwidth being wasted.

Camouflage. Camouflage [36] is a memory traffic shaping mechanism. It shapes the timing of memory requests to follow a predetermined distribution which is independent from any victim secrets. Camouflage relies on offline profiling to obtain the distribution that approximately matches the bandwidth utilization of the victim application. Shaping to a distribution is accomplished by selectively delaying existing memory requests, and issuing fake requests when necessary.

Unfortunately, Camouflage does not offer strong security guarantees. Specifically, Camouflage is unable to hide fine-grained memory access patterns, including the *ordering* of memory requests and bank contention. We use an example in Figure 2 to illustrate how distribution-based traffic shaping is insufficient to block information leakage. Assume Camouflage aims to shape the injection time of consecutive victim requests to the target distribution in Figure 2(a), i.e., one 200-cycle interval and one 400-cycle interval. Under Camouflage, the output of the shaper is not necessarily deterministic. Given different victim request inputs, the shaper can generate two different request sequences as shown in Figure 2(b). Both request sequences ① and ② conform to the distribution in Figure 2(a), but they differ in the *ordering* of the injection intervals. Sequence ① has the 200-cycle interval first and then the 400-cycle interval next, while sequence ② swaps the order of the two intervals. An attacker can use memory timing side channels (Section 2.2) to easily distinguish the two sequences. Moreover, the distribution used by Camouflage does not consider any bank information. Thus,

Table 1: Design goals of DAGguise and comparison with existing defense mechanisms.

	Fixed Service [25]	Camouflage [36]	DAGguise (this paper)
Security	✓	x	✓
Performance Overhead	High	Low	Medium
Profiling Cost	–	High	Low

Camouflage is further vulnerable to attacks which exploit bank contention.

Another limitation of Camouflage is that its offline profiling process is both expensive and oftentimes infeasible. The timing distribution of the victim is inherently dependent on co-running applications, as memory contention can slow down the victim program and significantly affect the injection intervals of its memory requests. Therefore, to obtain good performance, the target timing distributions used by Camouflage must not only be tailored to the program being protected, but also to the applications expected to run alongside the victim, significantly increasing the offline profiling cost. Moreover, such a profiling method is completely infeasible if the co-running applications also need protection and the application owners do not want to share any memory bandwidth usage information.

3.2 Design Goals

We propose DAGguise to achieve the three design goals as shown in Table 1, comparing it with the existing defense mechanisms, Fixed Service [25] and Camouflage [36].

First, our security goal is to block information leakage via fine-grained memory access patterns, including the number of memory requests, the timings of requests, and bank/row information. Second, the defense mechanism should incur a low performance overhead. Different from FS [25], which uses static partitioning which leads to significant bandwidth under-utilization, DAGguise can flexibly allocate memory bandwidth among different applications based on each application's actual bandwidth requirements. Finally, we aim to address the substantive profiling issue in Camouflage [36]. DAGguise uses a feasible and lightweight profiling method which only needs to profile the victim application *alone*, without needing knowledge about the bandwidth requirements of potentially co-located programs.

4 DAGGUISE: *r*DAG REQUEST SHAPER

We propose DAGguise, an effective defense mechanism to mitigate memory timing side channels. The core idea of DAGguise is to shape memory requests into a pre-determined pattern described using a novel graph representation, which we call a *Directed Acyclic Request Graph* or *rDAG* for short. An *r*DAG is general enough to describe any detailed memory request pattern, including those ignored by Camouflage [36], such as the ordering of requests and their bank information. Moreover, *r*DAGs are versatile and can react to contention within the memory controller, helping to adjust memory bandwidth allocation automatically to achieve better performance.

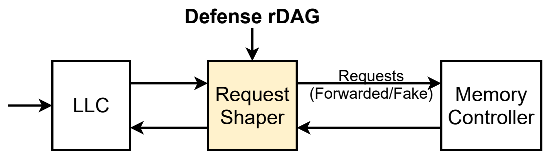

Figure 3: DAGguise overview.

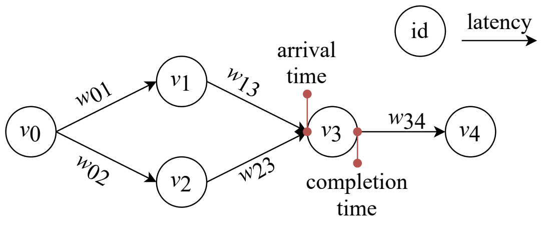

Figure 4: An *r*DAG example.

At a high level, DAGguise introduces a request shaper that works as a proxy agent for the transmitter and emits requests following the timing dependencies prescribed by an *r*DAG, which we call a defense *r*DAG. An overview of DAGguise is shown in Figure 3. The shaping operation is achieved by delaying existing requests and emitting fake requests. Note that any secret-independent defense *r*DAG can be used to effectively block information leakage. To achieve better performance, the defense *r*DAG should match the bandwidth utilization of the victim application. We profile the victim application alone to construct a defense *r*DAG by configuring parameters in an *r*DAG template.

In this section, we first introduce the *r*DAG representation in Section 4.1. We then describe the DAGguise scheme through an illustrative example in Section 4.2, demonstrating both the security properties of DAGguise and the versatility of the *r*DAG representation. We then provide further details of the DAGguise architecture, describing the offline profiling methodology in Section 4.3 and the online shaping mechanism in Section 4.4.

4.1 Directed Acyclic Request Graph (*r*DAG)

We introduce *Directed Acyclic Request Graphs* (*rDAGs*) to describe memory request patterns. An *r*DAG is a weighted direct acyclic graph that encodes the detailed timing dependencies between memory requests. An example of an *r*DAG is shown in Figure 4. Each vertex represents a memory request. Each edge, connecting two vertices, represents a timing dependency between the two requests. A timing dependency indicates that the destination request can only be emitted after the memory controller finishes serving the source request, e.g., request v_1 must be emitted after the response for request v_0 leaves the memory controller. If there does not exist a path between two vertices, it means the two corresponding requests can be emitted in parallel, such as request v_1 and v_2.

To consider possible memory contention, an *r*DAG encodes detailed timing information as follows. First, an *r*DAG encodes two time points for each memory request, its *arrival time* and its *completion time*. The arrival time is the time point when a request arrives at the memory controller and enters the transaction queue;

the completion time is the time point when a request has been fully consumed by the memory controller and the response for the request leaves the memory controller. We often conventionally represent the *r*DAG with implicit time flowing from left to right. Hence, it is convenient to associate the arrival time with the left side of a vertex, and the completion time with the right side of a vertex, as shown in Figure 4. Note that, due to contention, it can take a *variable amount of time* between when the request arrives at the memory controller and when the memory controller finishes serving the request.

Each vertex is associated with a bank ID and a tag to indicate whether it is a read or write request. This information is included since the memory controller's scheduling policy takes these properties into account in deciding when to serve the request.

Each edge is associated with a weight that measures the latency between the completion time of the source request and the arrival time of the destination request. For example, the weight of the edge connecting vertices v_0 and v_1 is w_{01}, so we have $t_{arrival}(v_1) = t_{completion}(v_0) + w_{01}$.

***r*DAG Properties.** *r*DAGs have two appealing properties: *generality* and *versatility*. First, an *r*DAG is general enough to represent any fine-grained request pattern, describing the injection intervals between requests (used by Camouflage [36]), timing dependencies between requests, and memory level parallelism. Moreover, an *r*DAG can also describe complex and irregular request patterns generated by real applications.

Second, rather than being a constant representation of memory request patterns, *r*DAGs are versatile, meaning that an *r*DAG can accommodate for unknown memory latencies. Specifically, when a request in an *r*DAG is delayed due to memory contention, its dependent requests will also be delayed. For example, in Figure 4, if the request v_3 suffers from bank contention, the completion time of the vertex v_3 will be delayed. As a result, the arrival time of the dependent request v_4 is also delayed. This versatility property allows an *r*DAG to flexibly represent different memory request injection times.

Original *r*DAGs vs. Defense *r*DAGs. A victim's *unshaped* memory request pattern can also be described using an *r*DAG, which we call the *original rDAG*. This original *r*DAG varies with the secret value used by the victim application.

The *r*DAG representation conveniently visualizes the lifetime of memory requests. Specifically, in an original *r*DAG, the time represented by a vertex (i.e. between the vertex's arrival and completion times) corresponds to the time that the request spends within the memory controller, where it may experience contention. The latency indicated by the edge weight represents inter-request timing relationships, corresponding to the time to traverse through the cache hierarchy and perform dependent computations in a core.

A defense *r*DAG is used to describe the memory request patterns that should be emitted by the DAGguise request shaper, which is placed between the LLC and the memory controller (as shown in Figure 3). The request shaper takes a secret-independent defense *r*DAG as input, and shapes the victim's memory requests according to this *r*DAG. Effectively, the shaper *encapsulates* the victim's original *r*DAG inside the defense *r*DAG.

Note that we *do not* need to obtain the original *r*DAG for our defense mechanism to work. We also note that the defense *r*DAG also does not need to closely resemble the original *r*DAG to achieve good performance. We show how to directly obtain a defense *r*DAG via statistical profiling in Section 4.3.

4.2 An Illustrative Example

We now describe the DAGguise scheme through an illustrative example, shown in Figure 5, demonstrating both the security properties of DAGguise and the versatility of the *r*DAG representation.

Security Properties of DAGguise. In this example, a victim application emits different memory access patterns based a boolean secret value. Figure 5(a) shows the victim's original request patterns using *r*DAGs. When the victim's secret is 0, the application emits one request at a time with a 100-cycle interval between the completion time of each request and the arrival time of its subsequent request. When the secret is 1, the application emits requests slower with a 200-cycle interval between consecutive requests. In both cases, we assume a fixed DRAM latency of 100 cycles. DAGguise works by shaping the two memory request patterns into the same pattern described by the defense *r*DAG in Figure 5(a), making the interval between consecutive requests 150 cycles.

Figure 5(b) demonstrates how each of the victim's request patterns are shaped in accordance with the defense *r*DAG. The first line for each secret represents the victim's original request pattern, corresponding to the victim's original *r*DAG in Figure 5(a). The shaper delays the victim's requests, as shown on the second line for each secret, to match the timing pattern prescribed by the defense *r*DAG. The final request pattern output by the shaper is shown on the third line.

When the secret is 0, the shaper delays each of the victim's requests by 50 cycles to increase the timing interval between requests to 150 cycles (as required by the defense *r*DAG). When the secret is 1, the shaper needs to both delay the victim's requests and issue fake requests. Since the victim issues requests with 200-cycle intervals, and the defense *r*DAG emits requests faster with a timing interval of 150 cycles, the shaper generates a *fake request* when the victim has no outstanding request pending. For example, the second and the fourth requests output by the shaper (on the third line) are fake requests. The victim's actual requests are further delayed to become the third and fifth requests output by the shaper.

Since the requests generated by the shaper have identical timing intervals (shown on the third line) regardless of the victim's secret, and because an attacker cannot differentiate between contention caused by fake and real requests, the shaping scheme employed by DAGguise guarantees that an attacker cannot distinguish between the victim's original request traces. A formal security verification of this property is discussed in Section 5.

Adaptivity Properties of DAGguise. Figures 5(c) and (d) further demonstrate how the versatility property of *r*DAGs helps DAGguise achieve good performance. We continue to use the same victim application and defense *r*DAG as the previous example. In these figures, the victim application protected by DAGguise shares a memory controller with an unprotected co-running application. Figure 5(c) describes the unprotected application with two phases of differing request intervals, modeling a real-world application

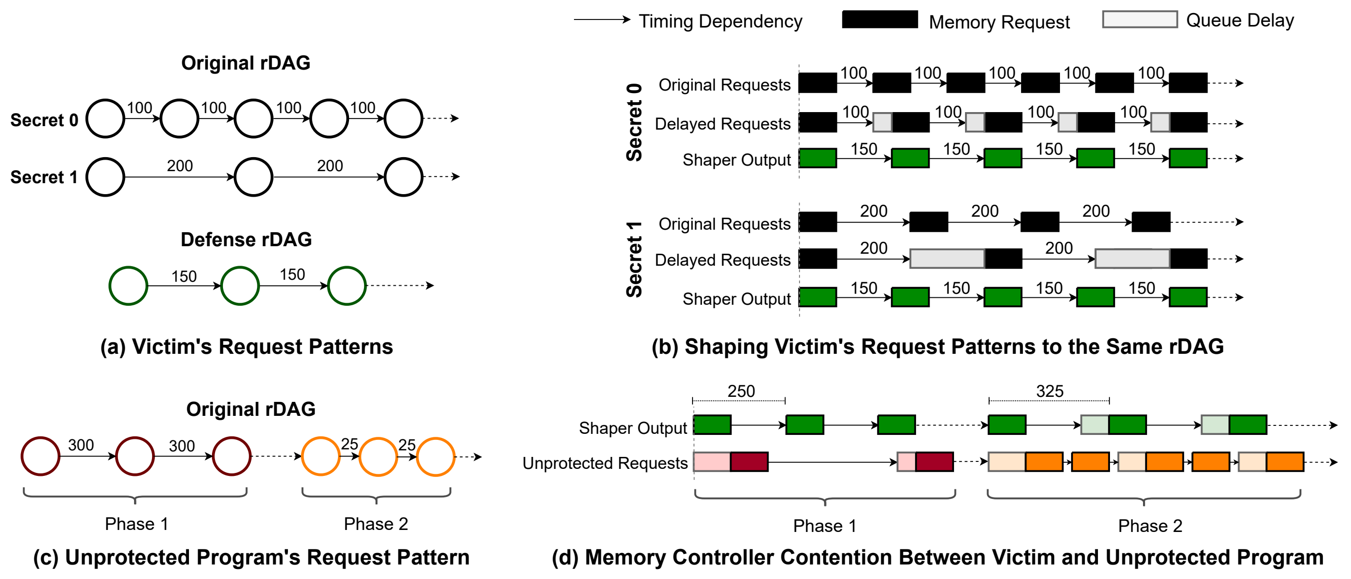
Timing Dependency
Memory Request
Queue Delay
Original rDAG
Secret 0
Secret 1
Defense rDAG
(a) Victim's Request Patterns
Original Requests
Delayed Requests
Shaper Output
(b) Shaping Victim's Request Patterns to the Same rDAG
(c) Unprotected Program's Request Pattern
Phase 1
Phase 2
Unprotected Requests
(d) Memory Controller Contention Between Victim and Unprotected Program

Figure 5: A running example to demonstrate the security and adaptivity properties of DAGguise.

with varied memory behaviors. Figure 5(d) details the effects of contention between the unprotected application's memory requests and the victim's shaped memory requests.

In phase 1, the unprotected application emits memory requests at a slow interval of 300 cycles, and there is not much contention at the memory controller. As a result, the shaper is able to maintain the victim's ideal injection interval of 250 cycles, i.e., 100 cycles for the memory access latency, and 150 cycles for the timing dependency (i.e., the weighted edge between vertices in the defense *r*DAG).

In phase 2, the unprotected application generates requests at a rapid interval of 25 cycles, causing a large amount of contention at the memory controller and delaying many of the shaper's requests. To maintain the timing dependencies in the defense *r*DAG, the shaper emits the next request 150 cycles after the response of the previous request returns. As a result, due to contention, the injection intervals of the victim's requests in phase 2 are increased from the original 250 cycles to 325 cycles. By slowing down the shaper's emission rate, the scheduler is able to allocate more bandwidth to the unprotected application and achieve better overall memory utilization.

Thanks to the versatility property of *r*DAGs, DAGguise is able to adapt to memory controller contention and adjust its own emission rate, allowing the memory controller to achieve better memory utilization while still maintaining security. Note that, while the example focuses on the case of running a protected application with an unprotected application, DAGguise also works effectively for the case of running multiple protected applications together. In this case, multiple defense *r*DAGs can interact with each other in a similar way as in Figure 5(d), as a "denser" defense *r*DAG can obtain more bandwidth from the memory controller.

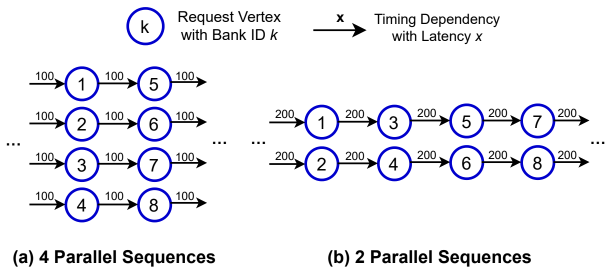
Request Vertex with Bank ID k
Timing Dependency with Latency x
(a) 4 Parallel Sequences
(b) 2 Parallel Sequences

Figure 6: Example *r*DAGs used in DAGguise, derived from *r*DAG templates.

4.3 Offline Profiling Method

The goal of the offline profiling phase is to find a suitable defense *r*DAG to be used by the memory request shaper. It is important to note that shaping requests to any secret-independent defense *r*DAG will ensure security. The offline profiling step is thus used to optimize for system-wide performance.

DAGguise uses a lightweight two-step profiling method: 1) obtaining an *r*DAG search space by configuring parameters in an *r*DAG template, and 2) profiling the victim application *alone* using different candidate *r*DAGs to select a final defense *r*DAG.

Generating an *r*DAG Search Space. Rather than searching the entire space of possible *r*DAGs, we generate an *r*DAG search space by deriving candidate *r*DAGs from an *r*DAG template. The search space can be generated by varying configurable parameters in the *r*DAG template, including the number of parallel sequences, the edge weights, and the write ratio (the frequency of write requests). Note that the template determines the complexity of the *r*DAGs. We intentionally choose templates that follow a regular and repetitive pattern, aiming to simplify the defense *r*DAGs and reduce the

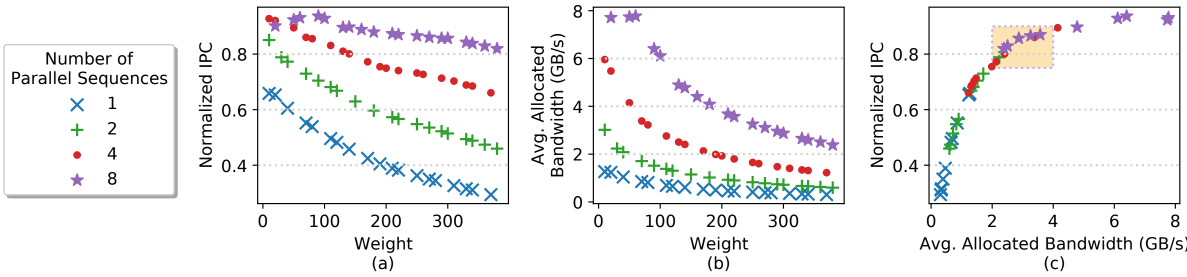

Figure 7: Selecting a defense *r*DAG for DocDist based on sensitivity to allocated bandwidth.

hardware overhead of storing and processing the defense *r*DAGs during the online shaping phase (Section 4.4).

Figure 6 shows two examples of *r*DAGs used in DAGguise, as derived from an *r*DAG template. Figure 6(a) demonstrates an *r*DAG with four parallel sequences (where each sequence contains requests that alternate between two different banks) with uniform edge weights of 100 DRAM cycles. Figure 6(b) demonstrates an *r*DAG derived from the same template when reducing the number of parallel sequences to 2 and increasing the edge weights to 200 DRAM cycles.

Selecting a Defense *r*DAG. To select the final *r*DAG from the search space, we test these candidate *r*DAGs on the protected program by feeding each candidate *r*DAG to DAGguise and measuring the impact of DAGguise on the protected program's performance. Intuitively, if we choose a candidate *r*DAG with smaller edge weights and more parallel sequences, the defense *r*DAG becomes denser and thus can request more bandwidth from the memory controller, reducing the amount of bandwidth remaining for co-running applications. In other words, the density of the defense *r*DAG determines the allocated bandwidth to the protected application. To optimize for system-wide performance, we derive the final defense *r*DAG based on the victim program's sensitivity to allocated bandwidth. This presents a trade-off between the protected program's own IPC, and the proportion of memory bandwidth consumed by that program (which is made unavailable to co-running applications).

Figure 7 shows an example of selecting a defense *r*DAG for DocDist, a security sensitive application (see Section 6.1 for details about the experimental setup and DocDist). In this example, a candidate *r*DAG can have 1, 2, 4, or 8 parallel sequences, and a uniform edge weight varying from 0 to 400 DRAM cycles. We run the victim alone, recording the victim's IPC (Figure 7(a)) and memory bandwidth utilization (Figure 7(b)) for each candidate defense *r*DAG. Figure 7(c) combines the IPC and bandwidth results of (a) and (b), demonstrating how the victim program's IPC changes according to the bandwidth allocated to it.

From Figure 7(a) and (b), we observe that as the edge weight decreases and the number of parallel sequences increases, the normalized IPC of the protected program increases, and the allocated bandwidth also increases. From Figure 7(c), we observe that the IPC of DocDist increases quickly when increasing the allocated bandwidth from 0 to 3 GB/s, with a diminishing return after the allocated bandwidth exceeds 4 GB/s. A cost-effective selection of defense *r*DAGs should lie within the highlighted region where the allocated bandwidth is around 2-4 GB/s. Thus, for our evaluation in Section 6, we select a defense *r*DAG for DocDist identical to Figure 6(a), comprising of 4 parallel sequences and a uniform edge weight of 100 DRAM cycles. As DocDist is a streaming application which performs very few writes, we set the write ratio (the proportion of vertices in the defense *r*DAG marked as writes) to be small (i.e. $\frac{1}{1000}$). For applications with more varied access patterns, further profiling can be performed to derive an appropriate write ratio which maximizes IPC and minimizes allocated bandwidth.

Note that the actual system-wide performance can vary as co-running applications can have varying bandwidth demands. We heavily rely on the versatility property of *r*DAGs to dynamically adjust the bandwidth utilization for the protected application when running with other applications.

Low Profiling Cost. The profiling cost required by DAGguise is low for two reasons. First, as the *r*DAG representation is versatile, we only need to profile the victim application *alone*, without requiring any information about co-located applications. This makes our approach much more practical than the approach used by Camouflage [36]. Second, we generate candidate *r*DAGs from *r*DAG templates, significantly reducing the search space of defense *r*DAGs.

Multithreaded Applications. So far, we have considered how to profile a single-threaded application to generate an optimal defense *r*DAG. For programs with multiple threads belonging to the same security domain, it is possible to utilize a single defense *r*DAG shared across all threads of a program, or multiple defense *r*DAGs with each one being exclusively used by one thread. These two approaches have different trade-offs in profiling cost and system-wide performance. Using a single defense *r*DAG for all threads allows for different threads to share vertices in the defense *r*DAG, reducing the number of fake requests issued and thus increasing system-wide performance. However, this approach may increase the offline profiling cost, since using a single defense *r*DAG may require re-profiling the application for each possible number of concurrent threads. Using one *r*DAG per thread reduces the overall offline profiling cost, in exchange for an increased online performance overhead.

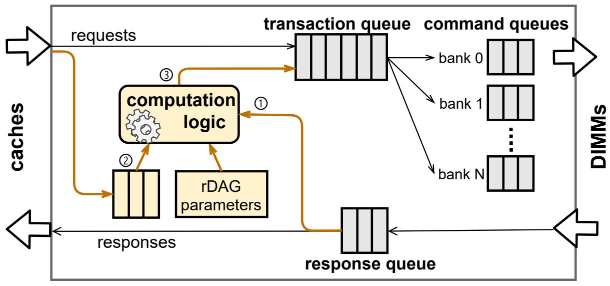

Figure 8: DAGguise memory controller architecture.

4.4 Online Shaping Mechanism

DAGguise uses a request shaper to disguise the transmitter's request pattern. Specifically, the shaper works as a proxy agent for the transmitter and emits requests following the defense *r*DAG by either delaying some of the transmitter's requests or emitting fake requests. Figure 8 shows the memory controller with DAGguise's hardware highlighted.

The baseline memory controller has a transaction queue and multiple command queues which are arranged so that there is one queue per bank. We have discussed how the memory controller manages these queues in Section 2.1.

DAGguise's hardware needs to include the following three components for *each* security domain that needs protection: a private transaction queue, *r*DAG parameter registers, and the shaper logic. To make our mechanism work, every memory request is tagged with a security domain ID. If a memory request is emitted by a domain that is under protection, the request will be inserted into the corresponding domain's private queue, while other requests are directly inserted into the global transaction queue.

We describe the shaper's operations using a single-bank configuration as an example. The shaper logic keeps track of the responses and decides when to emit the next request. When the shaper receives a response for its own security domain (①), the shaper logic computes whether the next request is ready to be emitted. If the next request is ready to be emitted, the shaper checks the private transaction queue to see whether there exists a pending request (②) matching the type of request to be issued (i.e. read or write). In the case that such a matching pending request exists in the private queue, the request will be transferred to the global transaction queue; otherwise, a fake request will be generated and inserted into the transaction queue (③).

In a multi-bank scenario, the access pattern to different memory banks could leak information. Thus, DAGguise needs to ensure that the bank access pattern is the same no matter what secret value is used. Recall that each vertex in an *r*DAG is associated with a bank ID (Section 4.1). Each step in Figure 8 takes the bank information into account. For example, when the shaper is ready to emit the next request, in addition to checking whether a pending request exists in the private queue, it needs to search the private queue to look for a request with the same bank ID. Similarly, when generating fake requests, the shaper needs to generate the request with the bank ID as prescribed by the defense *r*DAG.

As discussed in Section 2.2, row-buffer hits and misses can also leak information [22]. To hide row-buffer access patterns when using DAGguise, the memory controller must use a closed-row policy, ensuring that DRAM rows are closed immediately after every read or write. It also is possible to encode row-buffer activity in the defense *r*DAG to avoid the overhead of using a closed-row policy. Each vertex could additionally specify whether the memory access is a row-hit, and *r*DAGs with varying row-buffer hit ratios could be explored during the profiling stage. Such a scheme would save costs related to always closing a row, but when a vertex is marked as a row-hit in the *r*DAG and the program actually accesses a different (closed) row, DAGguise would need to emit a fake request to maintain security (negatively impacting performance). We leave further exploration of this direction as future work.

***r*DAG Computation Logic.** The *r*DAG computation logic is responsible for tracking the execution status of the defense *r*DAG and determining whether a request needs to be emitted by the shaper on a given cycle (and the bank ID/write status of that request if one is required). The complexity of this logic is fully determined by the defense *r*DAG. Recall that the defense *r*DAG is derived from an *r*DAG template following a regular and repetitive pattern (Section 4.3). Consequently, the corresponding computation logic is fairly simple.

For example, to track the status of the *r*DAG template in Figure 6, this logic only needs to track the following states for each bank: a bit to indicate whether the shaper is waiting for a response, a bit to indicate whether the next request is a read or write, and a counter to track the remaining cycles until the next request is required. We evaluate the area overhead of this computation logic for multiple parallel security domains, in addition to the required supplementary private queue storage in Section 6.4.

Fake Requests. When the shaper needs to emit a request but there does not exist a matching pending request, the shaper must insert a fake request into the global transaction queue. The fake request accesses a random address in the targeted bank. Issuing fake requests, however, can incur high energy consumption. Prior work [25] has introduced multiple approaches to address these energy concerns. One possible approach is to "suppress" fake requests. Rather than issuing these requests to the DIMMs, we can update the timing parameters and DRAM states as if the request was actually performed, as the data of these fake requests is irrelevant. An alternative approach is to use the fake requests to do useful work, e.g., issuing prefetching requests. For simplicity, we use the suppression approach in this paper.

Shaper Management. The DAGguise hardware structures, particularly the rDAG parameter registers for each security domain, need to be securely managed by privileged software that is part of the system's trusted computing base (TCB). Such software could be a security monitor [6] or microcode [12] in a system with support for enclaves, or the operating system or hypervisor when protecting user-level applications/virtual machines. Specifically, the privileged software is responsible for initializing and clearing the *r*DAG parameter registers when requested. During context switches, the privileged software is required to save and restore the *r*DAG registers, private queue state, and computation logic states.

5 SECURITY VERIFICATION

In this section, we start by describing a formally modeled system of DAGguise, followed by an explicitly defined indistinguishability property. We then provide details on the verification process. We perform our verification using k-induction [26] (a common technique for the verification of transition systems) in Rosette [28], a solver-aided programming language that integrates SMT solvers.

5.1 System Modeling

We model the DAGguise system as a state machine, consisting of an rDAG request shaper and a memory controller. We denote the simulation state of the DAGguise system at the beginning of cycle i as S_i, which includes the states of the shaper, the memory controller, and the buffers between them. We use S_{reset} to denote the state after a reset operation.

The inputs to the state machine are two memory request traces from a transmitter and a receiver, denoted as Req_{Tx} and Req_{Rx} respectively. The transmitter's request trace is passed to the request shaper, while the receiver's request trace is directly passed to the memory controller. The outputs of the state machine are two memory response traces to the transmitter and the receiver, which are denoted as Resp_{Tx} and Resp_{Rx}. A request/response trace is a vector, with the element at index i describing whether the trace contains a request/response at cycle i and the bank ID of the request/response. For example, $\text{Req}_{Tx}[i] = (\text{valid}_i, \text{bankID}_i)$.

Given a state S_i at cycle i and request traces for j cycles, Req_{Tx} and Req_{Rx}, we use the notation $S_i \xrightarrow[\text{Req}_{Tx},\text{Req}_{Rx}]{\text{Resp}_{Tx},\text{Resp}_{Rx}} S_{i+j}$ to denote simulation of the system for j cycles from state S_i to state S_{i+j} that outputs the response traces Resp_{Tx} to the transmitter and Resp_{Rx} to the receiver.[1] The transition function is determined by the configuration of the DAGguise system, including the chosen defense rDAG and the scheduling policy used by the memory controller.

For demonstration purposes, in our Rosette implementation we model a simplified memory controller that uses a FCFS scheduling policy and a constant memory latency of 2 cycles. The modeled request shaper uses a defense rDAG with a sequence of strictly dependent requests. It is feasible to extend our tool to verify whether the security property holds for different rDAGs and complex memory controllers.

5.2 Security Property

Recall that in Section 2.3, we consider a system as secure if an adversary (the receiver) cannot distinguish between different request traces of the victim (the transmitter) based on its own response latencies. By *indistinguishability* of request traces, we mean that the receiver's response trace Resp_{Rx} is independent from the transmitter's request trace Req_{Tx}.

We formally define $\text{P}(S_0, n)$, meaning the system achieves indistinguishability when running the system from the state S_0 for n cycles.

$$\begin{aligned}
\text{P}(S_0, n) := \forall\ &\text{Req}_{Tx}, \text{Req}'_{Tx},\ \forall\ \text{Req}_{Rx} \\
&\textbf{if } S_0 \xrightarrow[\text{Req}_{Tx},\text{Req}_{Rx}]{\text{Resp}_{Tx},\text{Resp}_{Rx}} S_n \text{ and } S_0 \xrightarrow[\text{Req}'_{Tx},\text{Req}_{Rx}]{\text{Resp}'_{Tx},\text{Resp}'_{Rx}} S'_n \\
&\textbf{then } \text{Resp}_{Rx} = \text{Resp}'_{Rx}
\end{aligned}$$

We verify that $\text{P}(S_0, n)$ holds for an arbitrary n when setting $S_0 = S_{\text{reset}}$. Note that in practice, a request may depend on previous responses, and so may depend on previous requests. This is not a problem as we prove the property for all possible sequences of requests, independently of how they came to be.

5.3 Verifying the Security Property using K-Induction

We use k-induction [26] to verify the security property above. Our verification involves the following two high-level steps:

1) *Base step*: Perform bounded model checking to verify that the security property $\text{P}(S_{\text{reset}}, k)$ holds for a small integer k;
2) *Induction step*: Simulate the system from two arbitrary starting states S and S', taking two arbitrary request traces Req_{Tx} and Req'_{Tx} for k+1 cycles. Assuming the receiver cannot distinguish between the two cases in the first k cycles, check whether the receiver can distinguish them in the $(k$+1$)$-th cycle.

Base Step. To perform the bounded model checking of $\text{P}(S_{\text{reset}}, k)$, we model arbitrary inputs to the system by defining three symbolic vectors to represent Req_{Tx}, Req'_{Tx}, and Req_{Rx}. We then simulate the system symbolically for k cycles and obtain the response traces for the receiver, Resp_{Rx} and Resp'_{Rx}. We implement the symbolic simulation process in Rosette and call the SMT solver to search for a binding of symbolic vectors to concrete values that violates the assertion $\text{Resp}_{Rx} = \text{Resp}'_{Rx}$.

Induction Step. The induction step is equivalent to searching for a violation of the following assertion.

$$\begin{aligned}
&\forall\ Req_{Tx}, Req'_{Tx},\ \forall\ Req_{Rx},\ \forall\ S, S' \\
&\textbf{\textit{if}}\ S \xrightarrow[Req_{Tx},Req_{Rx}]{Resp_{Tx},Resp_{Rx}} S_{k+1} \textit{ and } S' \xrightarrow[Req'_{Tx},Req_{Rx}]{Resp'_{Tx},Resp'_{Rx}} S'_{k+1} \\
&\quad \textit{and } Resp_{Rx}[0\!:\!k) = Resp'_{Rx}[0\!:\!k) \\
&\textbf{\textit{then}}\ Resp_{Rx}[k] = Resp'_{Rx}[k]
\end{aligned}$$

Similarly, we use symbolic vectors to represent the request traces and the starting states, S and S'. Again we call the SMT solver to search for a binding of symbolic vectors to concrete values that satisfies the assumption $\text{Resp}_{Rx}[0\!:\!k) = \text{Resp}'_{Rx}[0\!:\!k)$ and violates the assertion $\text{Resp}_{Rx}[k] = \text{Resp}'_{Rx}[k]$.

We follow standard methodology by incrementing the value of k until the induction step succeeds. The minimal k is related to the system's configuration, proportional to the number of cycles needed for a request to traverse the whole system. The time complexity of the verification process increases significantly with the value of k. For our specific implementation, the induction step works with $k = 6$, thanks to the simplified configuration of our model. While the verification is performed on a simplified model, the verification

[1]This standard notation of state machine transition places the input below the arrow and the output above the arrow.

Table 2: Baseline architecture configurations.

Parameter	Value
Multicore	2 and 8 out-of-order cores at 2.4GHz
Core	8-issue, out-of-order, 192-entry ROB
Private L1 I-Cache/D-Cache	32KB each, 64B line, 8-way 4-cycle round-trip (RT) latency
Private L2 Cache	256kB, 64B line, 16-way, 13-cycle RT latency
Shared L3 Cache	1MB per core, 64B line, 16-way 42-cycle RT latency
DRAM Configuration	4GB (2-core) and 8GB (8-core) 1 Channel, 1 Rank/Channel, 8 Banks/Rank Frequency: 1600Mbps
DRAM Timing Parameters	t_{RC} = 39, t_{RCD} = 11, t_{RAS} = 28, t_{FAW} = 24, t_{WR} = 12, t_{RP} = 11, t_{RTRS} = 2, t_{CAS} = 11, t_{RTP} = 6, t_{BURST} = 4, t_{CCD} = 4, t_{WTR} = 6, t_{RRD} = 5, t_{REFI} = 7.8μs, t_{RFC} = 260ns

process itself is sound, with potential to extend to more complex configurations.

6 EVALUATION

6.1 Experimental Setup

To evaluate the performance overhead of DAGguise, we use gem5 [4], a cycle-accurate simulator. We use DRAMSim2 [23] to model the memory controller and DIMMs. Table 2 shows the details of the simulated architectures.

Experiment Configurations. We compare DAGguise and FS-BTA, a state-of-the-art protection scheme, against an insecure baseline. *FS-BTA* is short for Fixed Service Bank Triple Alternation, a performance optimized variant of Fixed Service [25]. FS-BTA aggressively pipelines requests such that parallel bank accesses can occur under narrow circumstances, while still maintaining non-interference.

The insecure baseline uses an open-row policy, while FS-BTA and DAGguise utilize a closed-row policy to mitigate row-buffer attacks. Both schemes can mitigate the memory timing side channels described in Section 2.2. Note that we do not provide a performance comparison to Camouflage [36] as it does not fully hide bank contention, while most of the performance penalties associated with FS-BTA and DAGguise stem from protecting bank access patterns.

Benchmarks. To evaluate the impact of DAGguise on overall system performance, we co-locate victim programs with benchmark applications on separate cores. The sample set of fifteen co-running benchmark applications are selected from the SPEC2017rate suite [16]. For each SPEC application, we utilize the SimPoint methodology [21] to run up to 10 representative intervals of 50 million instructions each to accurately reflect the application's performance [35]. The caches are populated prior to interval data collection using 1 million warm-up instructions, and all simulations are run using gem5's system call emulation mode.

We use two victim programs: Document Distance (*DocDist*) and DNA sequence matching (*DNA*), which process unstructured data and can leak information via memory accesses [34].

DocDist [11] compares documents for similarity, computing the distance between a private input document and a public reference

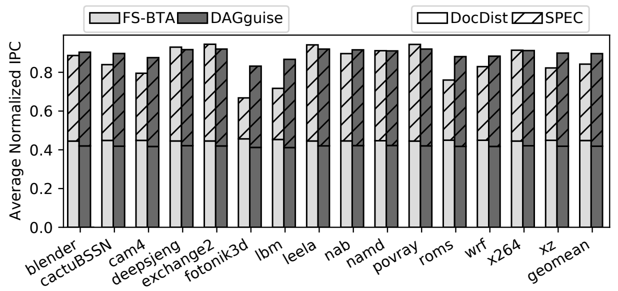

Figure 9: Average Normalized IPC running DocDist with one SPEC application on a two-core system.

document. DocDist precomputes a feature vector counting the frequency of each word in the reference document. Upon receiving an input document, it first computes a feature vector for that document, then computes the euclidean distance between the input and the reference feature vectors. The access pattern to the feature vectors can leak information.

DNA sequence matching [24] takes a private DNA sequence as input and aligns it with a public DNA sequence. Specifically, the public DNA sequence is divided into substrings and stored in a hash table. To do the alignment, the hash table is searched for common substrings with the private DNA sequence. The access pattern to the hash table can leak information.

We perform system-wide performance evaluations across two system environments. In Section 6.2 we evaluate a two-core system running one protected application and one SPEC benchmark. We extend our analysis in Section 6.3 to evaluate DAGguise's scalability on an eight-core system running four protected domains alongside four co-running SPEC benchmarks.

6.2 Performance Overhead

To measure the impact that DAGguise has on overall system performance within a two-core system, we measure the IPCs of each application (DocDist protected by DAGguise, and one SPEC benchmark), and then normalize each IPC to its baseline performance under the insecure configuration (under the same co-location). We then take the *average* of these values to arrive at an average normalized IPC, representing the overall performance of the system, shown in Figure 9.

We observe that in a two-core environment DAGguise has a 10% system slowdown compared to the insecure baseline, while enjoying a modest performance increase over FS-BTA, achieving a relative 6% performance increase.

As a general trend, we note that DAGguise is particularly good at maintaining the performance of co-running applications at the expense of the protected program's performance. In most observed cases, the SPEC program performs better using DAGguise than FS-BTA (20% better, on average), while DocDist does worse (7% worse, on average), resulting in an overall system speedup. For some non-memory-bound benchmarks (such as leela) we observe an overall *decrease* in performance, as the additional bandwidth made available to unprotected applications by DAGguise is not used by the benchmark, while the protected program still pays

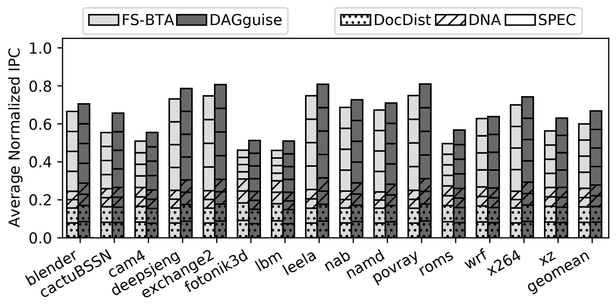

Figure 10: Average Normalized IPC of two DocDist, two DNA, and four SPEC processes on an eight-core system.

costs for the shaper. We hypothesize that additional performance for *protected* programs may be achieved by expanding the *r*DAG search space to consider more complex defense *r*DAGs.

6.3 Scalability

In order to demonstrate our system's scalability compared to FS-BTA, we expand our simulation to encompass multiple co-running victim programs alongside unprotected applications. On an eight-core system we use four DAGguise shapers to protect four programs, two copies of DocDist and two of DNA, co-located with four identical unprotected copies of SPEC benchmarks. Under FS-BTA, each individual victim receives $\frac{1}{8}$ of the total number of slots, while the remaining $\frac{4}{8}$ slots are shared amongst the SPEC applications.

The average normalized IPC results for our eight-core experiment are shown in Figure 10. DAGguise encounters a 34% system-wide slowdown compared to the insecure baseline, with an improved 12% average system-wide performance gain relative to FS-BTA. In a heavily provisioned system protected by DAGguise we observe that most applications, not just unprotected ones, achieve a relative speed-up compared to their performance under FS-BTA. This suggests that DAGguise is indeed versatile to complex bandwidth patterns, allowing for better bandwidth utilization compared to FS-BTA.

6.4 Area Overhead

To evaluate the area overhead of DAGguise, we compute the combined area of the computation logic (described in Section 4.4) and the private transaction queues.

We implement the computation logic in RTL, synthesizing it using the YoSys suite [31] and the 45nm FreePDK45 cell library [15]. We evaluate an eight shaper configuration (allowing for eight independent security domains), each supporting eight banks, 16-bit *r*DAG weights, and eight private queue entries. The implementation supports template *r*DAGs like in Figure 6, allowing 1, 2, 4, and 8-parallel patterns.

To evaluate the SRAM area overhead of the private transaction queues, we use Cacti [3]. Each private queue is sized to match the expected maximum number of parallel memory accesses of a protected program. Each queue entry contains a request's 64-bit address and, if the request is a write, 64B of data. Even if the queue is full, we do not leak any information, as each queue is private to a security domain.

Table 3: Area overhead of DAGguise for 8 protected domains.

Component	Resources	Area (mm^2)
Computation Logic	13424 Gates	0.02022
Private Queue (8×8 entries)	4608 B (72B×64) SRAM	0.01705
Total	–	0.03727

The area of the computation logic and private transaction queues is reported in Table 3, with an eight shaper configuration ultimately requiring a footprint of only $0.037mm^2$.

7 GENERALIZING DAGGUISE TO MITIGATE OTHER TYPES OF SIDE CHANNELS

While this paper focuses on mitigating memory timing side channels, its key insights (to shape request patterns using an *r*DAG) are even more general. DAGguise can be applied to address a broader range of side channels, such as those that exploit contention in SMT cores [2], on-chip networks [30], cache banks [33], etc. Similar to main memory, these resources are all associated with schedulers which decide the order in which requests are served, such as a pipeline scheduler deciding which instruction will use a functional unit, and a Network-on-Chip scheduler deciding which packet will use a network link. The scheduler introduces extra latency to some requests due to contention, which can leak information.

The principles of DAGguise can be used to mitigate these *scheduler-based* channels. For example, consider using DAGguise to block leakage via functional unit contention in SMT cores. We can profile the transmitter program and construct a defense *r*DAG, where each vertex represents an instruction's request to use a specific type of functional unit. We then place a request shaper between the decode and the dispatch stages. The shaper emits requests following the defense *r*DAG by delaying instructions and emitting fake instructions. It is promising to improve performance by pairing defense *r*DAGs with complementary functional unit bandwidth requirements on the same core. We leave this as future work.

8 RELATED WORK

We have discussed most related work in Section 3. A few other pertinent works are as follows.

Temporal partitioning (TP) [29] divides time into fixed-length periods during which only requests from a given security domain are scheduled, rather than interleaving requests, as in Fixed Service [25]. TP still guarantees non-interference but performs worse than FS, suffering from a static bandwidth allocation.

In addition to FS-BTA, which we compare against in Section 6, *Fixed Service* [25] has other variants which perform spatial partitioning at the bank, rank, or channel-level. While these variants can improve performance, they severely limit the number of simultaneous programs and the allowable memory usage of each. DAGguise has no such spatial partitioning requirements.

Ascend [9] and its follow-up paper [10] examine a different threat model and consider a passive (non-interfering) attacker probing memory buses to observe request patterns, and tries to obfuscate them using traffic shaping. They rely on a fixed request rate that is changed periodically, resulting in bounded amounts of leakage.

9 CONCLUSION

We introduced DAGguise, an effective defense mechanism against memory timing side channels. DAGguise utilizes Directed Acyclic Request Graphs (*r*DAGs), a novel memory request pattern representation, to shape memory access patterns into secret-independent ones. DAGguise is able to attain formally verified security guarantees while allowing for dynamic traffic contention to achieve good performance, only requiring a lightweight and feasible offline profiling process. DAGguise's insights can further be applied to other scheduler-based side channels which exploit contention in other microarchitectural structures, such as SMT cores and on-chip networks.

ACKNOWLEDGMENTS

This work was funded in part by the NSF under grant CNS-2046359, and by the Air Force Office of Scientific Research (AFOSR) under grant FA9550-20-1-0402. We thank the anonymous reviewers and our shepherd Onur Mutlu for all of their valuable feedback. We also thank Zirui (Neil) Zhao for his help in generating the SPEC application checkpoints used in the evaluation.

A ARTIFACT APPENDIX

A.1 Abstract

Our artifact comprises of two distinct parts: a unified gem5 and DRAMSim2 model (for performance evaluation), and a Rosette model (for security verification).

The unified gem5 and DRAMSim2 model is to profile defense *r*DAGs and evaluate the performance of *DAGguise*. To profile defense *r*DAGs, we include the sample victim programs (*DocDist* and *DNA*) as described in the paper, in addition to an *r*DAG generation tool. For performance comparison, our model also evaluates an insecure baseline and another secure scheme (*FS-BTA*).

The Rosette model is to verify the security property of *DAGguise* using k-induction as described in Section 5 of the paper.

A.2 Artifact Check-List (Meta-Information)

gem5 Simulator:

- **Program:** SPEC 2017, DocDist, and DNA (mrsFAST).
- **Compilation:** SPEC 2017 was compiled with clang-3.9, and gem5 was compiled with gcc-5.4.0.
- **Data set:** SPEC 2017 benchmark inputs are *reference* size.
- **Run-time environment:** Linux (with Docker support).
- **Hardware:** 48-core machine recommended (but not required).
- **Run-time state:** We utilize the SimPoint methodology to run up to 10 representative intervals of 50 million instructions each to accurately reflect the application's performance.
- **Metrics:** Results are reported as normalized CPI and average DRAM memory bandwidth.
- **Output:** Plots are generated using the provided scripts.
- **How much disk space required (approximately)?:** The simulation framework occupy approx 2GB, and the SPEC benchmarks (+ checkpoints) occupy approx 40GB.
- **How much time is needed to prepare workflow (approximately)?:** 1 Hour.
- **How much time is needed to complete experiments (approximately)?:** 1 Day.
- **Publicly available?:** Yes - https://github.com/CSAIL-Arch-Sec/DAGguise
- **Code licenses (if publicly available)?:** Berkeley Style (gem5), BSD (DRAMSim2, mrsFAST), Proprietary (SPEC 2017).
- **Workflow framework used?:** HTCondor is used to launch batch jobs.
- **Archived (provide DOI)?:** 10.5281/zenodo.5748606

Rosette Implementation:

- **Run-time environment:** Linux (with Docker support), Racket interpreter, Rosette library.
- **How much disk space required (approximately)?:** 1GB.
- **How much time is needed to complete experiments (approximately)?:** 1 Hour.
- **Publicly available?:** Yes - https://github.com/CSAIL-Arch-Sec/DAGguise-verification
- **Code licenses (if publicly available)?:** MIT License.
- **Archived (provide DOI)?:** 10.5281/zenodo.5748606

B GEM5 ARTIFACT

B.1 Description

B.1.1 How to Access. Our unified simulator model can be found at https://github.com/CSAIL-Arch-Sec/DAGguise.

B.1.2 Software Dependencies. The gem5 simulator infrastructure was tested on Ubuntu 18.04. We require the same dependencies as a standard gem5 installation (https://www.gem5.org/documentation/general_docs/building). For plotting, we require some additional Python libraries, which can be installed by running:

```
pip3 install -r eval_scripts/requirements.txt
```

For convenience, we additionally include a complete Dockerfile which captures all software dependencies required to build and use our infrastructure. However, as recreating our results requires running roughly 500 simulation instances, we highly recommend running HTCondor (without using Docker) to manage these jobs.

B.1.3 Data Sets. SPEC 2017 benchmark inputs are *reference* size.

B.2 Installation

Clone the DAGguise repository with *--recurse-submodules* to also get the DRAMSim2 implementation.

After cloning the repository, set the following environment variables:

```
export GEM5_ROOT=/path/to/gem5/
export SPEC_ROOT=/path/to/SPEC/
```

Then, following the experimental workflow below will automatically build/execute the simulation as required.

B.3 Experiment Workflow

B.3.1 Overview. We include an example workflow to reproduce Figures 7 and 9 in `eval_scripts/`.

Building Simulator and Preparing Checkpoints:

- `run_once.sh` - Patches the user-provided SPEC2017 gem5 checkpoints to be compatible with the provided version of gem5 (this only should be run once).
- `build_gem5.sh` - Builds gem5 using the standard SCons workflow.
- `generate_sample_checkpoints.sh` - Builds DocDist (the sample victim program) and generates a standalone checkpoint.

Generating Figure 7:

- `run_sensitivity_condor.sh` - Runs the defense *r*DAG parameter sweep (i.e. the offline profiling step, described in Section 4.3).
- `plot_fig7.sh` - Plots the results shown in Figure 7.

Generating Figure 9:

- `merge_checkpoint.sh` - Merges the single CPU checkpoints of DocDist and each SPEC SimPoint checkpoint into a new combined checkpoint (used for 2 CPU simulation).
- `generate_dag.sh` - Generates a sample defense *r*DAG.
- `run_simu.sh` - Executes the merged SPEC/DocDist checkpoint(s) under the simulator framework.
- `plot_fig9.sh` - Plots the results shown in Figure 9.

Some scripts are split into `_single` and `_condor` variants. The `_single` scripts are used to execute a single SimPoint/DocDist execution pair, while `_condor` scripts are used for batch execution.

B.3.2 Running in Docker. While we recommend running outside of a Docker environment due to the relatively large size of the SPEC benchmark suite, we do include a Dockerfile if desired. To build the docker environment, run:

```
docker build -t dagguise . -f docker/Dockerfile
```

from the root directory. Then, you can ssh into the docker environment by running:

```
docker run –rm -it –entrypoint bash dagguise
```

B.3.3 Running with HTCondor. If the system has HTCondor installed (highly recommended), running the `_condor` variant scripts will handle job submission on your behalf. To check the status of the jobs, run `condor_q`.

B.4 Evaluation and Expected Results

Following the aforementioned workflow should generate Figures 7 and 9 which match those in the paper.

B.5 Experiment Customization

To change the *r*DAG under examination when reproducing Figure 9, modify the flags provided within the `generate_dag.sh` script. More information about the supported flags can be found in the DAG generation tool source code (`dag_generator/dag_generator.py`).

The sample victim programs (found in `sample_programs/`) can also be tweaked if desired.

C ROSETTE ARTIFACT

C.1 Description

C.1.1 How to Access. Our Rosette model can be found at https://github.com/CSAIL-Arch-Sec/DAGguise-verification.

C.1.2 Software Dependencies. The security verification tool was tested on Ubuntu 18.04. We include a complete Dockerfile which captures all software requirements to build and use our verification tool, including the Racket environment and Rosette library.

C.2 Installation

After cloning the repository, enter the repository folder and build the docker environment by running:

```
docker-compose up -d
```

C.3 Experiment Workflow

Log into the docker container and enter the repository folder with:

```
docker-compose exec dagguise-verification bash
cd /DAGguise-verification
```

Run the security verification tool with:

```
raco test src/checkSecu.rkt
or
raco test ++arg -cycle ++arg 5 src/checkSecu.rkt
```

The command without extra arguments runs k-induction with the default value of $K = 6$. The user can test with different K values, such as $K = 5$ in the above example.

C.4 Evaluation and Expected Results

The security verification tool implements k-induction and verifies the base and induction steps. With a properly-chosen K, such as $K = 6$, the program should output:

```
**** Base Step Finished ****
(unsat)
...
**** Induction Step Finished ****
(unsat)
```

meaning the security property is not violated and no counter example is found. We find 6 is the minimal value of K to prove the security property of our DAGguise model.

With an improperly-chosen K, such as $K = 5$, the program should output a counter example. This means the verification is unsuccessful, and the user should try a larger K value.

D METHODOLOGY

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. 2007. Predicting Secret Keys via Branch Prediction. In *Cryptographers' Track at the RSA Conference (CT-RSA)*. Springer.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. https://doi.org/10.1109/SP.2019.00066

[3] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-chip Memories. *ACM Transactions on Architecture and Code Optimization* (2017). https://doi.org/10.1145/3085572

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* (2011). https://doi.org/10.1145/2024716.2024718

[5] Thomas Bourgeat, Jules Drean, Yuheng Yang, Lillian Tsai, Joel Emer, and Mengjia Yan. 2020. CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches. In *Proceedings of the 53th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. https://doi.org/10.1109/MICRO50266.2020.00092

[6] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security)*. USENIX Association.

[7] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2020. A Benchmark Suite for Evaluating Caches' Vulnerability to Timing Attacks. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. https://doi.org/10.1145/3373376.3378510

[8] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM. https://doi.org/10.1145/3173162.3173204

[9] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. 2012. A Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*. ACM. https://doi.org/10.1145/2382536.2382540

[10] Christopher W Fletchery, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. 2014. Suppressing the Oblivious RAM Timing Channel while Making Information Leakage and Program Efficiency Trade-offs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. https://doi.org/10.1109/HPCA.2014.6835932

[11] Hazem Gamal. 2020. Document Distance. https://github.com/Hazem-Gamall/document-distance.

[12] Intel. 2013. Intel Software Guard Extensions Programming Reference. https://software.intel.com/en-us/sgx/sdk.

[13] Bruce Jacob, David Wang, and Spencer Ng. 2010. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann.

[14] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. https://doi.org/10.1109/MICRO.2018.00083

[15] Jesper Knudsen. 2008. Nangate 45nm open cell library. *CDNLive, EMEA* (2008).

[16] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. 2020. The SPEC CPU Benchmark Suite. In *Systems Benchmarking*. Springer.

[17] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM. https://doi.org/10.1145/3342195.3387532

[18] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE. https://doi.org/10.1109/SP.2015.43

[19] Michael Neve and Jean-Pierre Seifert. 2006. Advances on Access-driven Cache Attacks on AES. In *Selected Areas in Cryptography*. Springer.

[20] Colin Percival. 2005. Cache Missing for Fun and Profit. http://www.daemonology.net/papers/htt.pdf.

[21] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using Simpoint for Accurate and Efficient Simulation. *ACM SIGMETRICS Performance Evaluation Review* (2003). https://doi.org/10.1145/781027.781076

[22] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium (USENIX Security)*. USENIX Association.

[23] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Computer Architecture Letters* (2011). https://doi.org/10.1109/L-CA.2011.4

[24] sfu compbio. 2020. DNA Sequence Matching. https://github.com/sfu-compbio/mrsfast.

[25] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. 2015. Avoiding Information Leakage in the Memory Controller with Fixed Service Policies. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*. ACM. https://doi.org/10.1145/2830772.2830795

[26] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. Checking Safety Properties Using Induction and a SAT-solver. In *Formal Methods in Computer-Aided Design*. Springer.

[27] Jakub Szefer. 2016. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *Journal of Hardware and Systems Security* (2016).

[28] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM. https://doi.org/10.1145/2509578.2509586

[29] Yao Wang, Andrew Ferraiuolo, and G. Edward Suh. 2014. Timing Channel Protection for a Shared Memory Controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. https://doi.org/10.1109/HPCA.2014.6835934

[30] Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. 2013. SurfNoC: A Low Latency and Provably Non-interfering Approach to Secure Networks-on-chip. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. ACM. https://doi.org/10.1145/2485922.2485972

[31] Clifford Wolf. 2016. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/.

[32] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *23rd USENIX Security Symposium (USENIX Security)*. USENIX Association.

[33] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. *Journal of Cryptographic Engineering* (2017). https://doi.org/10.1007/s13389-017-0152-y

[34] Xiangyao Yu, Christopher W Fletcher, Ling Ren, Marten van Dijk, and Srinivas Devadas. 2013. Generalized External Interaction with Tamper-Resistant Hardware with Bounded Information Leakage. In *Proceedings of the 2013 ACM Workshop on Cloud Computing Security Workshop*. ACM. https://doi.org/10.1145/2517488.2517498

[35] Zirui Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher W. Fletcher, Adam Fletcher, Darko Marinov, and Josep Torrellas. 2020. Speculation Invariance (InvarSpec): Faster Safe Execution Through Program Analysis. In *Proceedings of the 53th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. https://doi.org/10.1109/MICRO50266.2020.00094

[36] Yanqi Zhou, Sameer Wagh, Prateek Mittal, and David Wentzlaff. 2017. Camouflage: Memory Traffic Shaping to Mitigate Timing Attacks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. https://doi.org/10.1109/HPCA.2017.36