The Guardian Council: Parallel Programmable Hardware Security

Sam Ainsworth
University of Cambridge, UK
sam.ainsworth@cl.cam.ac.uk

Timothy M. Jones
University of Cambridge, UK
timothy.jones@cl.cam.ac.uk

Abstract

Systems security is becoming more challenging in the face of untrusted programs and system users. Safeguards against attacks currently in use, such as buffer overflows, control-flow integrity, side channels and malware, are limited. Software protection schemes, while flexible, are often too expensive, and hardware schemes, while fast, are too constrained or out-of-date to be practical.

We demonstrate the best of both worlds with the Guardian Council, a novel parallel architecture to enforce a wide range of highly customisable and diverse security policies. We leverage heterogeneity and parallelism in the design of our system to perform security enforcement for a large high-performance core on a set of small microcontroller-sized cores. These Guardian Processing Elements (GPEs) are many orders of magnitude more efficient than conventional out-of-order superscalar processors, bringing high-performance security at very low power and area overheads. Alongside these highly parallel cores we provide fixed-function logging and communication units, and a powerful programming model, as part of an architecture designed for security.

Evaluation on a range of existing hardware and software protection mechanisms, reimplemented on the Guardian Council, demonstrates the flexibility of our approach with negligible overheads, out-performing prior work in the literature. For instance, 4 GPEs can provide forward control-flow integrity with 0% overhead, while 6 GPEs can provide a full shadow stack at only 2%.

CCS Concepts. • **Security and privacy → Hardware security implementation**; • **Computer systems organization** → *Multicore architectures*; *Heterogeneous (hybrid) systems.*

Keywords. Hardware Security, Heterogeneous Multicore

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378463

ACM Reference Format:
Sam Ainsworth and Timothy M. Jones. 2020. The Guardian Council: Parallel Programmable Hardware Security. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3373376.3378463

1 Introduction

Devices increasingly run applications the user doesn't trust. Code may be running on cloud platforms that share resources with third-party workloads [17], or on personal devices that also store sensitive information [60]. Within today's heightened threat environment [6, 25, 27, 34, 39, 48, 50, 53, 56], it is increasingly necessary to detect security violations in real time to ensure device safety and trustworthiness.

Enforcing security properites in hardware is a tempting proposition [11, 23, 28, 31, 32, 36, 37, 47]. However, fixed-function hardware is limited in utility if an attacker can simply change their targets to components without protection, or design software to deliberately circumvent the defences. Additionally, when vulnerabilities are discovered in hardware implementations [34, 56], they cease to become useful. Hardware schemes are typically unable to react to new threats as they appear, and take too long to be introduced into production systems, resulting in long windows of vulnerability. Software schemes [3, 13, 27, 45, 52, 57], while flexible, often result in too high overheads to see widespread implementation: typically, security features with an overhead greater than 5% do not see widespread use [56]. Current programmable security hardware [26, 44, 54, 62] is limited in performance, energy and silicon utilisation efficiency by the high overheads of software programmable resources.

The Guardian Council presents a different paradigm, providing the fine-grained user-level control and customisation of software, while keeping the low overheads associated with pure-hardware techniques. By leveraging heterogeneous parallelism and providing a parallel programming model for checking security properties, we can support updatable hardware security at extremely low overhead. Instead of fixed-function security units or software schemes sharing compute resources with a program, we add a set of dedicated parallel programmable compute units alongside the main compute core, each several orders of magnitude smaller than the main core [55]. These Guardian Processing Elements

(GPEs) are augmented with data-forwarding channels and limited, widely applicable, fixed-function support, dedicated to security monitoring. We describe and evaluate our architecture and parallel programming model for efficient updatable security protection that can change in response to attack evolution over a device's lifetime. We utilise these to analyse filtered traffic from a processor to detect a broad variety of violations, achieving high efficiency, applicability, and programmability.

We demonstrate our architecture by offloading techniques inspired by those currently implemented in hardware or software, such as Rowhammer prevention [38], shadow stacks [3], Control-Flow Guard [57] and AddressSanitizer [2] onto our GPEs at minimal power, performance, and area overheads.

Contributions

- We propose a parallel architecture, the Guardian Council, comprised of an array of Guardian Processing Elements, to allow realtime monitoring of entire program execution while maintaining programmability.
- We describe a programming model that provides an abstraction of the parallelism and monitoring strategy, to allow this architecture to implement a wide variety of different security policies, or Guardian Kernels.
- We give Guardian Kernel examples that can be implemented on the Guardian Council for broad use cases.
- We evaluate the overheads of implementing modern security protocols for a wide variety of techniques on our programmable architecture in simulation.

2 Requirements

Software schemes [3, 13, 27, 45, 52, 57] to improve security typically have unacceptably high performance overheads. Hardware schemes [11, 23, 28, 31, 32, 36, 37, 47] are slow to react to new attacks, and can be rendered ineffective if limitations in their fixed functionality can be exploited. We want to provide a system that can avoid the shortcomings of each. This requires programmable security hardware capable of implementing a wide set of current and future techniques.

2.1 Security Detection

A range of attacks target modern systems, all of which must be prevented from causing damage, such as buffer overflows [6, 27], Rowhammer attacks on DRAM [39], side channels [22, 41] (including speculative-execution attacks [40, 43]) and fault attacks [15, 33]. We need to be able to mitigate these with custom actions per technique, such as by terminating a program when necessary on a buffer overflow, or by taking actions to avoid harm. Responses vary according to the attack but may require, for example, refreshing DRAM rows if a suspected Rowhammer has been observed, or rescheduling threads if a side-channel attack is likely.

This paper is mainly focused around schemes that detect security violations. These are particularly amenable to use of decoupled hardware that is allowed to observe the execution stream [51]. Other policies exist for providing security guarantees [23], which may also be offloaded to similar hardware, but these are beyond the scope of this paper.

2.2 Analysis Channels

Many existing schemes [28, 36, 37, 47] use a variety of information channels to infer security violations, including performance counters, loads and stores, and instructions committed by the system. Any technique that seeks to implement similar detection algorithms must provide efficient channels for this information. Such channels should avoid slowdown of user programs, while allowing monitoring of malicious software. This favours a transparent hardware approach, instead of the programs themselves sending the data by software signalling.

2.3 Heterogenous Parallelism

Given the breadth of attacks, it is necessary to check a large number of different properties at once. Each requires custom programmable behaviour, and may need to analyse significant portions of a program's execution, at low cost in performance, area, and energy. To do this it is necessary to exploit parallelism. Small cores provide many orders of magnitude more efficiency [55] than the large out-of-order superscalar cores used in modern systems for good single-threaded performance. If we can exploit these to implement our architecture, overheads can be minimised.

Still, to be able to use such an implementation strategy the requisite compute capabilities must be amenable to parallelisation. Fortunately in this instance we have two opportunities to exploit parallelism. First, we can implement multiple different techniques at the same time, to detect different security violations, and each can be run on its own parallel unit. Second, for security properties that are more intensive or observe a larger proportion of the original program stream, we can exploit parallelism within a technique. This requires an efficient programming model that allows detection of the properties we need yet is efficient to implement in hardware so we can limit the overheads. The next section takes these requirements and presents a programmable security infrastructure based around an array of tiny processing elements.

2.4 Threat Model

We assume that user code is untrusted, and that the operating system's kernel is non-malicious but not inherently trusted. This means that the operating system should be allowed to install security monitors onto our programmable system without the monitors themselves needing to be analysed dynamically, but that the operating system need not be bug-free, and can have violations of its own properties reported to it by our system.

The specific guarantees provided by a programmable technique should, where possible, be a configurable part of the software programming model: a user should be able to trade

guarantees and coverage for performance, and vice versa. Still, the fixed-function hardware we use mandates some constraints. To decouple a parallel and latency-tolerant model of execution, useful when utilising heterogeneous cores, we should allow some latency between an attack occurring and it being detected. We therefore design our system to ensure detections in a) a time-frame where only a small amount of computation can occur, and b) before context switches or system calls that could allow an attacker to escalate control to shut down the security monitor itself.

Such detection hardware should be future-proof, by featuring widespread programmability and observability. We achieve this despite the provision of fixed-function hardware to improve performance, by allowing new functionality to be emulated in software on parallel hardware where necessary. To limit design invasion into the main core, which could affect critical paths and implementation complexity, we deliberately limit observable behaviour to the committed instruction stream. This means that attacks based on the behaviour of speculative execution [40] must be picked up through mechanisms that leave a committed-instruction footprint, such as an attacker's attempts to recover leaked data. To achieve this we assume some monitors running on our fabric will have visibility of system-wide behaviour, able to monitor attackers on behalf of potential victims.

3 The Guardian Council

The system architecture of The Guardian Council is shown in figure 1. A conventional high-performance, out-of-order superscalar core is coupled to a set of small, microcontroller-sized Guardian Processing Elements (GPEs), each around a thousandth the size of the main core and traditionally used in systems-on-chips for programmable state machines [55]. Data channels (section 3.2) move information to the programmable units from the core's instruction retirement stage and hardware performance counters. This is filtered (section 3.3) based on the data the GPEs are set to be interested in, mapped to a parallel unit (section 3.4), then placed in a FIFO queue to be processed. On each GPE a Guardian Kernel runs, which is a small program that implements (part of) a detection algorithm we are interested in. If a Guardian Kernel detects a violation, an exception is raised to the main core, which can then handle it either by ending the appropriate process or taking other mitigating actions.

3.1 Guardian Processing Elements

At the heart of our architecture is a set of small, in-order, microcontroller-sized processing units, to allow high computational capability at low power, performance, and area overhead. An example is shown in figure 2. Each is limited in terms of capabilities, but as a sum they can exceed the peak performance of the main core, given parallel work.

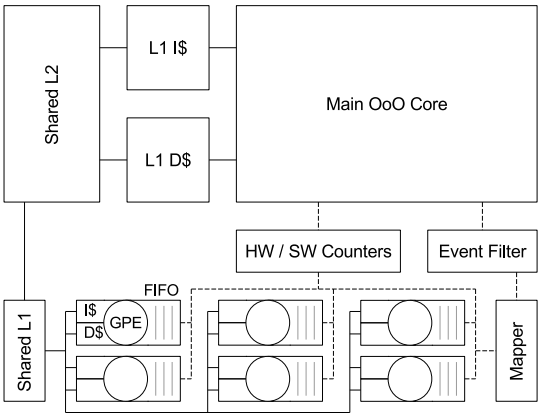


Figure 1. Architecture of the Guardian Council.

The GPEs each run analysis Guardian Kernels mandated by the operating system, or subscribed to by the user, but pre-approved by the OS as part of its trusted code base. The GPEs have a connection to the hardware counters of the main core, small private L0 instruction and data caches, along with a shared L1 cache. Data from the main core is passed to them from channels indicating instructions committed, which are then accessed by the cores from their respective FIFO queues.

The GPEs do not need to run the same instruction-set architecture as the main core, and are unlikely to need floating point hardware. Suitable target hardware would be a simplified design, such as a Cortex M0+ [55], with a two-stage pipeline and support for integer Thumb2 instructions.

3.2 Data Channels

We provide channels from main cores to the GPEs to carry execution data from programs running on the main core. These dedicated channels allow visibility of instruction execution on the core without adding explicit communication into the program's code. As in previous work on malware detectors [37, 47] and debug trace analysers [10, 14, 59], we add in a channel from the commit stage of the main core, to transport the opcode, program count, instruction commit time and operand data to the GPEs. We also transport hardware counter data [28, 30, 36, 46]. Together, these channels allow a large amount of visibility into execution. For example, loads, stores, branches, and performance counters, along with more complicated memory management instructions and system calls, can be and forwarded for analysis.

These are similar to existing trace debug mechanisms, with two key differences. As the Guardian Council works on-chip, and trace is optimised for off-chip communication, the Guardian Council typically forwards data instead of compressing and recomputing (for example, by sending instruction PCs instead of offsets). In addition, while current off-chip traces lose data when overloaded (to avoid slowdown), the Guardian Council instead ensures full coverage at the expense of slowdown, to guarantee security.

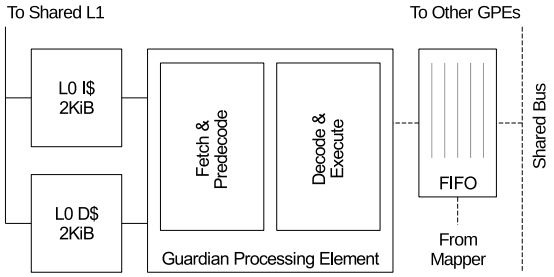


Figure 2. Microarchitecture of a single GPE.

To limit design invasion, we do not provide channels directly into the main core. We read the main core's registers at commit time, and load/store data from the load-store queue. Otherwise, the main core's microarchitecture remains the same. The Guardian Council does not, therefore, directly see the behaviour of speculative state. This has the side effect of limiting the potential of speculative state propagating to introduce inadvertent side channels via the Guardian Council itself. As the Guardian Council has a system-level view over running applications, we instead allow it to observe such behaviour indirectly, through the committed actions of attackers, in priming and/or probing caches, which are easier to spot than a victim's seemingly innocuous misspeculation.

3.3 Event Filter

The event filter is configured by the user to observe the main core. It sees all retired instructions and associated data from the commit stage of the core, but passes on only a fraction of these to the next stage. Although we could pass all retired instructions to a GPE that runs filtering software, to keep up with the main core it is beneficial to implement fixed-function logic for the task. We filter based on the opcode. The event filter is implemented as a configurable SRAM lookup table based on instruction masking, with a bit per instruction type: if the bit is set, then matching instructions are forwarded on to the mapper.

3.4 Mapper

Once irrelevant data-channel information has been filtered out, the mapper distributes the output to the appropriate GPE, both between separate Guardian Kernels running on different GPEs, and also within a particular Guardian Kernel, if it has multiple threads on different GPEs. To simplify this, Guardian Kernels are pinned to GPEs, as we discuss in section 4.3. The mapper features a separate connection from itself to each GPE, to avoid bus contention, and is implemented as a lookup table similar to the filter, along with a small amount of fixed-function routing logic.

The case of distributing to different Guardian Kernels simply involves forwarding the data to any GPE that is programmed to listen to a given instruction. However, when work must be split across multiple GPEs, scheduling opportunities become significantly more complex. We choose to implement three parallelism strategies: fixed, where particular opcodes for a Guardian Kernel always go to the same GPE; round-robin mode, where work is split evenly between Guardian Kernel threads; and a block mode, where work is sent to a single GPE until it is full, upon which another GPE is chosen, and a message sent to the first for synchronisation. The latter strategy is useful when data shows some sort of locality. We implement more complicated strategies in software on a GPE, when applicable (e.g., shadow stack, section 5.1), at the expense of parallel compute resource.

Before sending instructions to the relevant GPE(s), the mapper also masks out unimportant data. For example, if a GPE only listens to one event, the opcode may not be forwarded. If any operands are unused, they can also be discarded instead of being sent to a GPE queue. The mapper features a prefiltering stage for the most commonly observed instructions, such as loads and branches. This means that, instead of interpreting the register operands in software, target addresses are computed, or forwarded from the original execution, in hardware. This increases efficiency for the common case, while still allowing more detailed access for rarer or more complicated situations.

The mapper is conceptually similar to the event filter, in that it too reduces the events that get sent to each GPE, rather than broadcasting. Some functionality could be moved between the two, and both could be implemented purely in software. However, the mapper should receive a much smaller amount of data, as it has already been passed through the event filter, so will only be listening to a small subset of information from the main core. This means the mapper can feature more complex logic and scheduling schemes.

3.5 Hardware/Software Counters

Hardware performance counters have been shown to be important for a number of detection techniques [28, 30, 36]. We give the GPEs access to these values from the main core. However, not all counters we may need will be provided by a given system, and so we may want to use the GPEs' analysis features to generate custom counters. We add support for atomically updatable software counters in hardware. Any GPE is allowed to read from them, but writes are restricted: a GPE can only update a counter by sending the number of a counted value it has observed. This gets added to the counter atomically within the counter unit.

3.6 Per-GPE Queues

To allow decoupling between the mapping of events to a GPE and the processing of an event, each GPE contains a small FIFO queue. GPEs have their own FIFOs, which wake a GPE when not empty. If a FIFO is filled, the main core must stall commit until the FIFO has space available. This is necessary to avoid missing events that may be needed for detecting some security violations. For inter-security GPE communication, it is possible to write data to another GPE's

FIFO. This allows the GPEs to be combined to map and reduce data, or for the GPEs to act as a pipeline to perform different work on the same data, for example.

3.7 Memory System

The GPEs need a memory connection to fetch instructions. In addition, as they may keep state, they also need data memory access. Each GPE has its own small, private L0 caches for instructions and data. All GPEs share an L1 cache, which is connected in turn to an L2 cache shared with the main core. Coherence is implemented between GPEs using the shared L1 as an inclusive snoop filter to avoid full broadcast and thus limit its cost. Still, many Guardian Kernels do not need shared memory, and instead use message-passing via the GPEs' FIFOs, avoiding locks and L1 cache contention. This shared L1 is 4-way set associative, being optimised under the assumption that L0 data caches will share data that GPEs only read (AddressSanitizer, CFI), or that Guardian Kernels will stream data and mark accesses as non-temporal (favoring eviction from L1 over other data).

3.8 Multicore

Though our example (figure 1) is shown as one main core, this design is not limited to single-core setups. Each core has its own private set of GPEs, so when a main-core thread migrates to a different main core, any user-subscribed Guardian Kernels also move with it, though kernel-controlled Guardian Kernels (section 4.1) may stay in the same place, and run on the GPEs of every main core. We discuss what happens on a context switch in section 4.4.

For some attacks it is possible that by running on multiple cores the same attack can be performed with a fraction of the traffic from each, evading some forms of detector. In this scenario, the programmer can either utilise Guardian Kernels with lower (detection-specific) trigger thresholds, or communicate via data channels between GPEs on multiple main cores. At the simplest level, these data channels can be implemented via shared-memory communication with no extension. If a large amount of communication is necessary for a system, then global GPE indexing and a dedicated ring network between GPE clusters suffices for fast message-passing capability.

Multicores are not considered further in this paper because they do not affect the resulting overheads, and no evaluated Guardian Kernels would reduce in effectiveness for attacks spread across multiple cores. We include the potential attacks and solutions here for completeness: the limitations can be solved within software Guardian Kernel implementations.

3.9 Summary

The Guardian Council consists of a set of Guardian Processing Elements along with supporting logic to transfer and queue observations from a main core. Events from the main core are distributed to GPEs using filtering and mapping, to remove irrelevant data and distribute observations between GPEs. We now describe how this system can be programmed.

4 Programming Model

For a given process running on the main thread, a number of "Guardian Kernels", each intended to provide a certain security property, are run on the GPEs. Some are mandated by the operating system, and always run, and others can be subscribed to by the process itself.

4.1 User-Level vs OS-Level Detection

For some security properties, we only care that the security of our own process isn't violated. An example of such a property is control-flow integrity [3]. In such cases, we can allow the user to specify which properties of the system they want to make sure hold, and may even allow the user to directly program the security hardware.

However, in other cases, such as Rowhammer or side channel detection, it is the behaviour of other programs that a user must defend against. In such cases, all programs must be monitored, thus protection must be offered by the OS, and run for every application. These Guardian Kernels cannot be pre-empted by user-level Guardian Kernels, to avoid denial-of-service. Similarly, under virtualisation, a hypervisor's Guardian Kernels take priority over virtualised tasks.

4.2 Protecting the GPEs Themselves

Allowing the user to run arbitrary code on the GPEs raises its own set of potential security issues. A thread on a GPE, able to read and write memory and observe the behaviour of the main core, is liable to perform attacks such as side channels or Rowhammer triggering. This means that they too would need to be monitored by a Guardian Kernel of higher privilege for the system to be secure, and thus we would need a channel from the shared cache of the GPEs back into the event filter.

Rather than running arbitrary code on GPEs, we allow programs to subscribe to a set of services offered by existing OS-approved Guardian Kernels, potentially providing each with data. These would then run alongside mandatory Guardian Kernels run by the OS itself. New Guardian Kernels could be installed with OS permission onto the system, such as by issuing OS updates.

This solves the problem of policing the GPEs by moving the code running on them into the trusted computing base. As typical code running on such a device will be fairly simple, this is a reasonable implementation choice, and simplifies verification elsewhere.

4.3 GPE-Thread Virtualisation

Though Guardian Kernels can have multiple threads, in our implementation each GPE can only be used for one security

```
while (true)
  addr = get_fifo();
  // one bit per 32-bit instr word
  target = bitmap[(addr-min)>>5];
  if (!target & (1<<(((addr-min)>>2)&7)))
    raise_exception();
```

(a) Forward control-flow integrity

```
GPE k = 1..N
  while (true)
    (op,addr) = get_fifo();
    if (op == block)
      put_fifo(0,stack);
      stack.empty();
      put_fifo(0,(block,from,addr));
    else
      if (op == push)
        stack.push(addr);
      else
        if (stack.size() > 0)
          assert(addr == stack.top());
          stack.pop();
        else
          put_fifo(0,(pop,addr));

GPE 0
  while (true)
    (op,addr,from) = get_fifo();
    if (from != target)
      queue[from].push(op,addr);
    else
      if (op == block)
        target = from;
        stack.fill(queue[from]);
      else if (op == push)
        stack.push(addr);
      else
        assert(addr == stack.top());
        stack.pop();
```

(b) Shadow stack

```
while (true)
  addr, pc = get_fifo();
  // one bit per 128-bit allocation granule
  char bits = shadow[(address)>>7];
  if(!bits) continue;
  if(bits & (1<<((address >> 7)&8)))
     report_error(addr,pc);
```

(c) Sanitiser

```
while (true)
  address, time = get_fifo();
  if (time > cache_threshold)
    rand = int.next();
    if (rand > threshold)
      refresh(adjacents(address));
```

(d) Rowhammer

Code listing 1. Example Guardian Kernels running on GPEs.

thread at a time per application. This simplifies implementation, as each security thread is always running, and each FIFO is only used for one thread at a time.

Still, it is potentially limiting from a capacity and throughput point of view: if multiple kernels could virtualise on the same hardware, this would improve resource utilization. Since all Guardian Kernels in our system are trusted, there is no risk of side-channel attacks from this multiplexing. The reason that we do not currently support this is because it complicates the design of the FIFO. Since we need an available FIFO for all data-channel information we are listening to, we cannot switch out a currently active security thread. We therefore need some support for multiple occupancy in the FIFOs. This can be done by decoupling the FIFOs and GPEs from each other, such that there is one FIFO per thread, but threads can be moved between GPEs depending on scheduling requirements.

4.4 Context Switches

On a context switch, as a result of the Guardian Council, more state must be switched. Any OS-provided Guardian Kernels should be used for all programs, so do not need to switch, but any user-subscribed Guardian Kernels need separate state, and so must be switched out when the process changes. To avoid losing data in the FIFOs (which may be necessary for analyses that must track all state, and to prevent vulnerability to attacks designed to deliberately empty its FIFOs), we must ensure that FIFOs are empty on a context switch. Therefore, we pause the main CPU and allow the security threads to empty the FIFOs before switching. We also do the same before system calls, and any handling of precise exceptions or interrupts, to prevent a main thread from gaining operating-system privileges before previously executed code has been checked. Typically, FIFO elements move through the Guardian Council at a faster rate than other elements of the context switch that can be performed in parallel, removing the need for a delay. This FIFO latency is evaluated in section 6.5: emptying the Guardian Council's queues typically takes only a few nanoseconds, which is shorter than flushing the main core's pipeline.

5 Example Setups

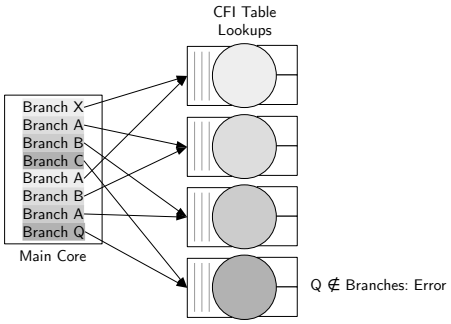
The Guardian Council is suitable for deploying a variety of defences. We present the general case of parallelism abstractions for the Guardian Council in figure 3, before giving specific examples and code using these techniques, then evaluating a subset of them in simulation. Where possible, we focus on mature techniques from the literature, to keep the design and evaluation of new techniques out of scope. We give the setup of the event filter and mapper, and pseudocode for the program(s) running on the GPEs, with details such as memory management abstracted away.

5.1 Control Flow

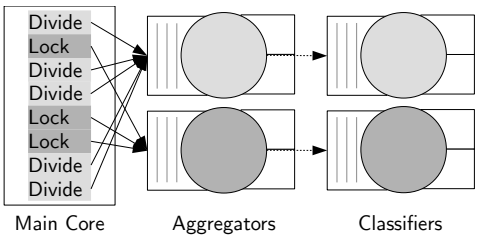
Forward Control-Flow Integrity

Our first example (listing 1(a)) is a technique similar to Control-Flow Guard [57]: all indirect branches are checked in a bit array provided by the user when subscribing to the Guardian Kernel. If the looked up bit is a zero, then an exception is raised to the main thread.

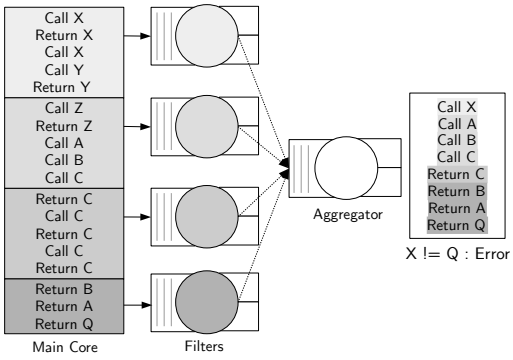
Event Filter: Indirect branches



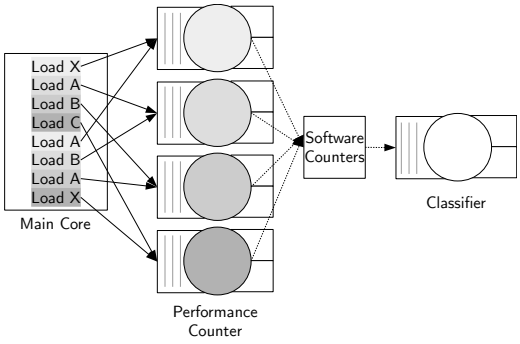
(a) *Cyclic Task Farm*, shown here running control-flow integrity, but also used for differential fault analysis prevention, reference checker, spatial safety and Rowhammer. Observations are interleaved between the GPEs, and each checked in parallel.



(b) *Parallel Pipelines*, shown here running a covert channel detection mechanism. Each type of event is sent to a single GPE, which then aggregates this information to send to a classifier GPE, in a pipeline formation.



(c) *Aggregated Bulk Task Farm*, shown here running a shadow stack mapped to GPEs. Calls and returns are processed in blocks, filtering push-pop pairs, before handling inter-block dependencies on an aggregator GPE.



(d) *Aggregated Cyclic Task Farm*, shown here running a software-defined performance counter, but is also used for counter and ensemble classifications. The parallel work is fed into the software counter unit, which is subsequently read by a single GPE running a classifier based on the data.

Figure 3. Example configurations for exploiting parallelism within Guardian Kernels.

Mapper: Indirect branches: Round robin between GPEs, masked so only the destination goes to the FIFO.
Parallelism: Cyclic Task Farm (figure 3(a)).

Fine-grained Forward Control-Flow Integrity

Our second example is inspired by Clang CFI [24], in that it is more fine-grained: instead of checking whether a destination is valid, it further checks whether the called function is the same type as the source. While normally this is implemented by instrumenting the source, to fully decouple the check, instead the source and destination PCs are observed by the Guardian Kernel, and looked up in tables of 8-bit entries to ensure a match. This increases coverage relative to our first technique at the expense of utilising more GPE resources.
Event Filter: Indirect branches
Mapper: Indirect branches: Round robin between GPEs, masked so the PC and destination go to the FIFO.
Parallelism: Cyclic Task Farm (figure 3(a)).

Shadow Stack

Here (listing 1(b)) we implement a shadow stack [1, 3], to prevent the overwriting of a return value, by a buffer overflow attack, for example. We parallelise this process using a prefiltering stage. Calls and returns are sent to a GPE, which adds these to its own shadow stack. When a GPE's queue is full, we move to another GPE. Any values that are left over from an individual GPE's stack are then sent to a coordinating GPE, which combines the results from each to verify integrity.
Event Filter: Calls, returns
Mapper: Calls, returns: send to GPE 1 to N, moving between each when a queue is full. Send opcode, current PC for calls, and return address for returns.
Parallelism: Aggregated Bulk Task Farm (figure 3(c)).

5.2 Memory Safety

Sanitiser (listing 1(c)) is a scheme for heap buffer overflow and free reuse detection, inspired by AddressSanitizer [2, 52]. Custom mallocs and frees place poison "red zones" around and within, respectively, the data's allocated region in a shadow address space, with one bit for every 128-bit allocation granule in the original address space. The memory reference checks can then be offloaded to the GPEs. Our evaluated implementation is based on dlmalloc [42], though any allocator would be suitable in practice.
Event Filter: Loads, stores
Mapper: Loads, stores: Round robin between GPEs, data masked so only the address goes to the FIFO.

Parallelism: Cyclic Task Farm (figure 3(a)).
To avoid both false positives and false negatives, the GPE queues must be allowed to empty before each malloc and free, potentially causing a stall on the main core.

5.3 Differential Fault Analysis Prevention

This technique prevents differential fault analysis attacks [49] by checking the results of ciphertext to ensure no bit flips have been induced, which could lead to the leaking of a secret key [15, 33]. Data encrypted on the main core is placed into several buffers, each of which is checked in parallel by a GPE. Any intermediate values necessary for the cipher to be repeated on each section are forwarded. If a check completes successfully, and all previous checks also complete, the buffer is allowed to be written out to IO. The use of heterogeneous cores is particularly useful in this context, in that errors will manifest differently on the main core versus the GPEs, making the system significantly harder to attack. Unlike previous work on generic parallel error detection [4, 5], GPEs can run a different architecture to the main core, as the checker code can be algorithmically generated based on the algorithm, reducing hardware complexity.
Event Filter: Explicit messaging from main core only.
Mapper: Round robin between GPEs, with messages containing pointers to the buffers to be checked.
Parallelism: Cyclic Task Farm (figure 3(a)).

5.4 Rowhammer

We emulate a scheme from Kim et al. [38, 39] to prevent Rowhammer (listing 1(d)), by randomly refreshing lines adjacent to accessed addresses. As lines being attacked by Rowhammer will be frequently activated, this will probabilistically refresh those being attacked. This can be parallelised across as many GPEs as necessary, as each observation is independent. We infer which memory accesses have gone to main memory from the CPU by timing instructions, rather than adding a direct channel to main memory.

More complicated schemes to prevent Rowhammer attacks [38] could also be implemented on our hardware. For example, state can be tracked to obtain specific details on which locations have been accessed.
Event Filter: Loads, stores, memory flushes
Mapper: Loads, stores, flushes: Round robin between GPEs, masked so the address and commit time goes to the FIFO.
Parallelism: Cyclic Task Farm (figure 3(a)).

5.5 Performance Counters

Custom Performance Counter
Here we give an example of a performance counter designed to count memory accesses within a certain range that is configured by the subscriber. To reduce traffic on the bus, it only reports new counts every 50 events.
Event Filter: Loads
Mapper: Loads: Round robin between GPEs, data masked so only the address goes to the FIFO.
Parallelism: Aggregated Cyclic Task Farm (figure 3(d)).
Performance Counter Classification
Rather than using the data channel hardware to track instructions, we can just use aggregate performance counter metrics. As with the work by Demme et al. [28] on online malware detection, we can use these as input to a feature classifier, by sampling the performance counters periodically. When anomalous software is detected, this can then be passed up to a more detailed analysis framework, either on the main core or on another GPE. As the amount of input data is relatively small, we are unlikely to need to parallelise the classifier onto multiple GPEs. We can instead use the other GPEs for other tasks, or for custom performance counters as input to the classifier. In the case where we directly use instructions as input to a classifier [47], it is likely that more performance would be required, and parallelisation would be necessary.
Ensemble Classifier
Other work features multiple classifiers, specialised to detect particular attacks, which are then combined [37]. We can run classifiers on multiple GPEs, then send the data from each to a single GPE for combination.

5.6 Side Channels

Covert Channel Detector
To detect covert channels, we give a scheme based on the work of Chen et al. [20]. This takes timing measurements of shared resources, and analyses them using a histogram-based approach. We utilise two forms of parallelism here. First, we split different resources to different GPEs. Second, we pipeline the detection: the first GPE processes data by collecting events per unit time, and the second then analyses that as a histogram to pick up the sending of ones or zeros across the channel. This is exemplified in figure 3(b), where we check for memory and divider timing channels, by detecting the frequency of bus locks, and contended divides, per unit time.
Event Filter: Bus locks, divides
Mapper: Bus locks: GPE 0; Divides: GPE 1. Filtered so that the cycle time of each instruction is sent into the FIFO.
Parallelism: Parallel Pipelines (figure 3(b)).
Flush + Reload Detector
To detect cache side-channel attacks (speculative, as used in Spectre [40] and Meltdown [43], and otherwise), we use the Guardian Council to analyse a potential attacker's use of cache flushes, for detection of eviction, and barriers, to indicate attempts to isolate instruction timing.
Event Filter: Flushes, Memory Barriers
Mapper: Flushes, Memory Barriers: Send to GPE 0.
Parallelism: Aggregated Cyclic Task Farm (figure 3(d)).
If both are used frequently, then an attempt at a cache timing attack is inferred. This will not necessarily detect all forms

Main Core	
Core	3-Wide, out-of-order, 3.2GHz
Pipeline	40-Entry ROB, 32-entry IQ, 16-entry LQ, 16-entry SQ, 128 Int / 128 FP registers, 3 Int ALUs, 2 FP ALUs, 1 Mult/Div ALU
Tournament Branch Pred.	2048-Entry local, 8192-entry global, 2048-entry chooser, 2048-entry BTB, 16-entry RAS
Memory	
L1 ICache	32KiB, 2-way, 2-cycle hit lat, 6 MSHRs
L1 DCache	32KiB, 2-way, 2-cycle hit lat, 6 MSHRs
L2 Cache	1MiB, 16-way, 12-cycle hit lat, 16 MSHRs, stride prefetcher
Memory	DDR3-1600 11-11-11-28 800MHz
Guardian Processing Elements	
Cores	In-order, 4 stage pipeline, 1GHz
FIFO	64-entry per GPE
Cache	2KiB L0 I & DCache per GPE, 16KiB shared L1

Table 1. Core and memory experimental setup.

of cache side-channel attack, as it is possible to evict data using the cache's set-associativity policy, but cache-flushing operations are significantly simpler and more efficient as a side channel, and so this technique is useful for detecting common attacks (including the Spectre and Meltdown proof of concepts [40, 43]). If more precise detection is required, then the Guardian Council could instead be reprogrammed to detect more general cache misses.

6 Evaluation

To evaluate the resources necessary for the Guardian Council, we modeled a high performance system using the gem5 simulator [16] in syscall emulation mode with the ARM instruction set and configuration given in table 1, based on that validated in previous work [35]. We evaluate six techniques from section 5, namely forward control-flow integrity, fine-grained forward control-flow integrity, shadow stacks, custom performance counters, Sanitiser, and Rowhammer prevention. Our evaluation suite is SPEC CPU2006, which we fast forward for 1 billion instructions before running for 1 billion instructions, except for those cases where fast forwarding affected the Guardian Kernel (Sanitiser, shadow stack). We used the Arm 64-bit instruction set, except from for Sanitiser, where the shadow lookup space for the heap required us to use the 32-bit Arm instruction set, due to no sparse mmap support in gem5. We present all applications that would compile and run in gem5 in both modes.

6.1 Overheads

Figure 4 shows the slowdown caused to the main core from waiting for event queue space, using increasing numbers of 1GHz GPEs to implement each Guardian Kernel.

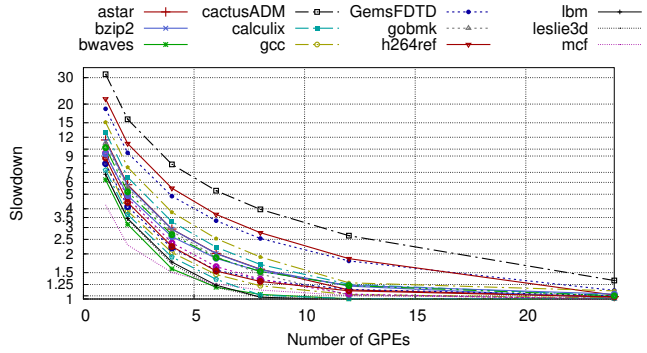
Sanitiser Figure 4(a) is the technique with highest evaluated overheads: 24 GPEs are needed for an average overhead of 4.9%. This is dominated by a few workloads with very high compute demands of the GPEs: cactusADM, h264ref, GemsFDTD and hmmer. These feature a large number of loads and stores without being particularly memory-bound, thus causing each GPE to perform a significant amount of work to verify each load and store is in-bounds and to an allocated location. Further, because the main core must stall not only when queues are full, but also on waiting for queues to empty when the allocator causes the shadow stack to change state, at low GPE counts large slowdowns are observed since work builds up for the GPEs to clear. Still, many other workloads need significantly fewer GPEs to reach sub-5% overheads.

Integrity Figure 4(b) faces much lower overheads, with four GPEs sufficient for negligible overheads on all evaluated workloads. Many perform very little indirect flow control, and so one GPE is sufficient, though others (h264ref, sjeng, omnetpp, xalancbmk, GemsFDTD) require significantly more compute on the Guardian Council.

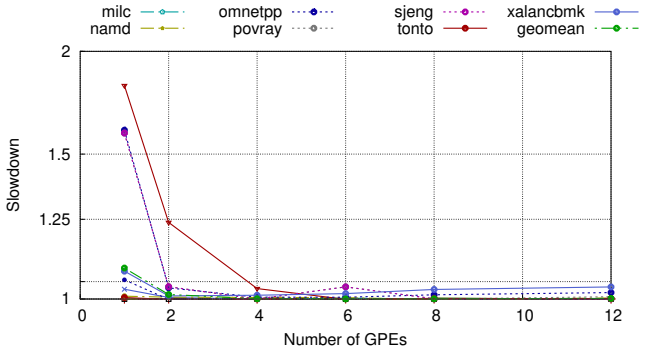
Fine-Grained Integrity Figure 4(c) responds to the same events as the coarse-grained version, and so similar benchmarks show significant overheads, while many workloads show almost none even with a single GPE. However, the increased work per indirect branch instruction causes overheads on some workloads to be higher, thus increasing the value of offload to the Guardian Council. Even with 12 GPEs, some workloads (xalancbmk and omnetpp) still show some overhead: this is not because of the main core stalling due to compute overload of the GPEs, or due to a lack of parallelism, but is instead due to high demands on the memory system, due to the larger tables that must be looked up to compare types of program count versus target, in contrast to the simple bitflag lookup of the coarse-grained scheme. Still, 1% average overhead can be reached with 6 GPEs.

Shadow stack Figure 4(d) shows that we can reduce overheads to below 5% using 6 GPEs: this is higher than for the forward-integrity techniques as all calls and returns must be instrumented, along with the more complex data structures used to maintain and combine multiple parallel shadow stacks. Four workloads (omnetpp, h264ref, xalancbmk, sjeng) asymptote above 1 even with 12 cores: this is caused by the imperfect parallelism of our shadow-stack algorithm overloading the single aggregating core (figure 3(c)) used to manage communication of each parallel shadow filter.

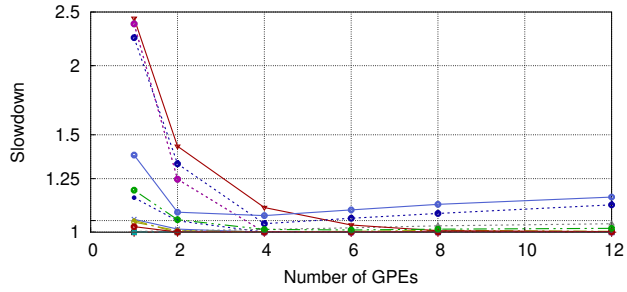
Counter As all workloads feature a significant amount of loads, we see in figure 4(e) that two or more GPEs are necessary for every evaluated workload in SPEC CPU2006. Eight GPEs are needed to lower geometric mean overhead to below 5%, and h264ref and astar still see some cost with 12 GPEs. While the analysis (load counting with hardware aggregation using the Guardian Council's software performance counters) is simpler than Sanitiser's, instrumenting all loads still requires significant GPE compute in some cases.



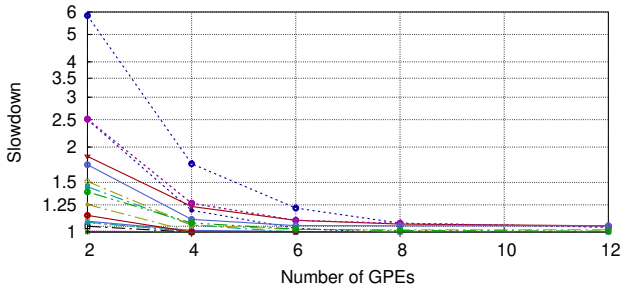
(a) Sanitiser



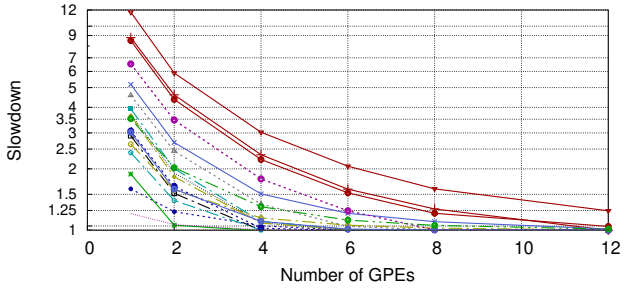
(b) Control-Flow Guard [57] style forward control-flow integrity



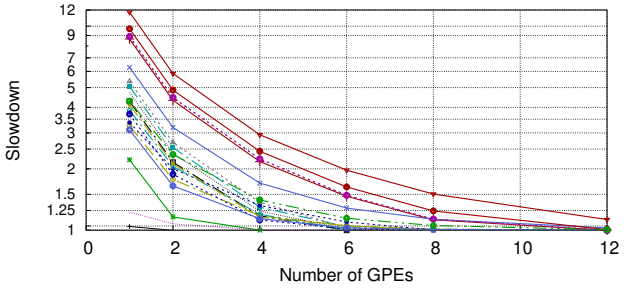
(c) Fine-grained forward control-flow integrity



(d) Shadow stack [3]



(e) Custom performance counter (loads within an address range)



(f) Rowhammer prevention [39]

Figure 4. System slowdown when using varying numbers of GPEs to implement each Guardian Kernel.

Rowhammer The pattern is broadly similar in figure 4(f) as for Counter, as most observed events require little work, with only occasional DRAM refreshes. However, as all memory events must be observed, instead of just loads, overheads are typically higher, though 8 GPEs are still sufficient for overheads below 5%.

Comparison to Software Techniques Figure 5 shows the performance of the Guardian Council versus the best performing equivalent software-only techniques we could find in the literature [18, 19, 52]. This is not a direct comparison as numbers are reported for x86 systems, and the software techniques feature complex optimisations not featured in our Guardian Council implementations. Offloading to the Guardian Council significantly reduces worst- and average-case overheads even against the best software techniques, by offloading instrumentation to dedicated hardware channels, and offloading software compute to efficient, parallel units.

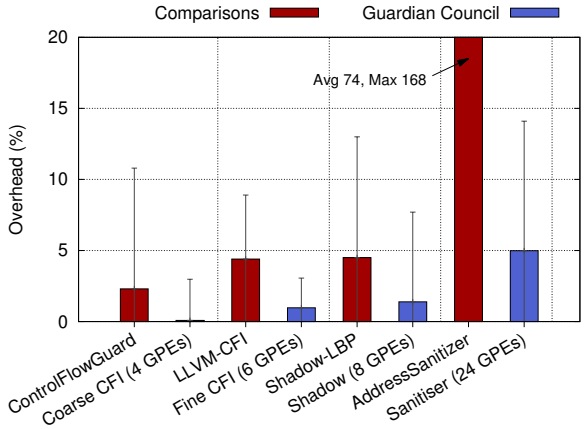
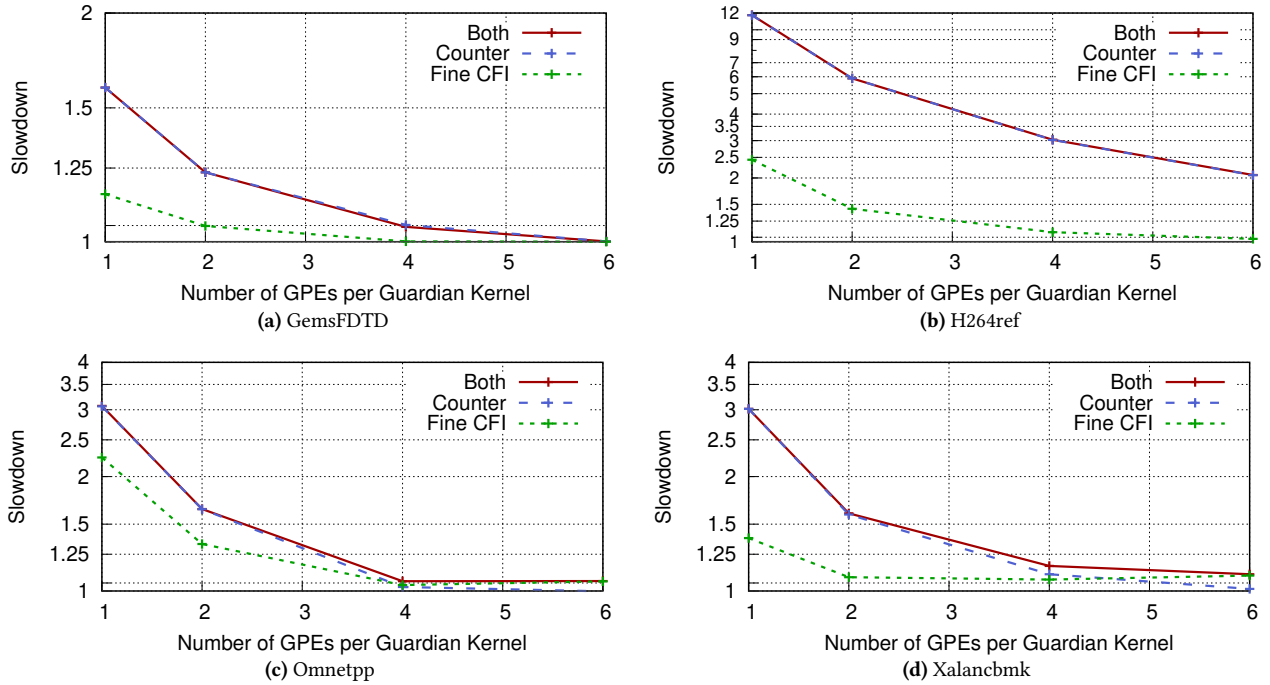


Figure 5. Average and maximum overheads for the Guardian Council and state-of-the-art software techniques.



(a) GemsFDTD **(b)** H264ref **(c)** Omnetpp **(d)** Xalancbmk

Figure 6. Slowdown for running fine-grained control flow and the load counter simultaneously, along with each individually.

6.2 Size and Power Estimates

To avoid reducing performance, some Guardian Kernels occasionally require a large number of GPEs. However, each is low performance and low overhead, with a small ISA. A comparable current core is the ARM Cortex M0+, which contains fewer than 12,000 gates [9] (approximately 50,000 transistors). Even when using 40nm power values for the M series cores and 20nm power values for A series without adjustment [7, 8], giving a highly conservative estimate, 24 GPEs at 1GHz can be included on a system within a 2.8% power consumption overhead budget. By the same data [7, 8] on comparable silicon processes we should expect these 24 GPEs to take up approximately 2.7% of the area of a Cortex A57, even if we exclude the A57's shared caches. With the added 112KiB of SRAM cache [61] that would be used for this number of GPEs, along with 32KiB of total FIFO queue space, this still results in an area overhead of only 8.4% total core area overhead relative to an unmodified main processor.

6.3 Combining Defenses

Figure 6 shows the slowdown when running two distinct Guardian Kernels simultaneously, on the subset of benchmarks that are performance intensive for both. We see that, in general, the slowdowns when running multiple defences exhibit a maximum, rather than a multiplicative, behaviour, which means that we only pay the overhead of the slowest Guardian Kernel, rather than all overheads combined. This is because, if one Guardian Kernel slows down an application's execution, then the event stream of instructions from that application will also slow down in throughput, causing lighter loads on other Guardian Kernels.

6.4 Evolution

The Guardian Council is designed to defend against both current and future attacks. This raises a question on how much security compute resource is necessary over time. As new attacks and defenses become available, resources for these will be necessary, and if those resources are already used by current defenses, then the system will slow down. For many defenses, they will only need to run some of the time: not every defense will be relevant to every process, we can switch out older defenses when more comprehensive or efficient ones become available, and different levels of coverage can be chosen based on application-specific overheads. For future generations of Guardian-Council-assisted architectures, the most widely-used Guardian Kernels can be moved into hardware, while still using the same data channels as the software GPEs. Every kernel we evaluate can be implemented simultaneously in 40 GPEs, which is less than 5% power overhead even for a comparatively small and efficient Arm Cortex A57 [7, 8]. Even this is over-provisioning: each kernel does not need its maximum allocation of GPEs simultaneously, and not all policies are necessary for all applications. Instead, we envision kernels being chosen per application based on overhead and required security. Still, for the most security conscious systems, significant over-provisioning may be necessary in terms of the number of GPEs provided in a system, to provide comprehensive lifetime coverage.

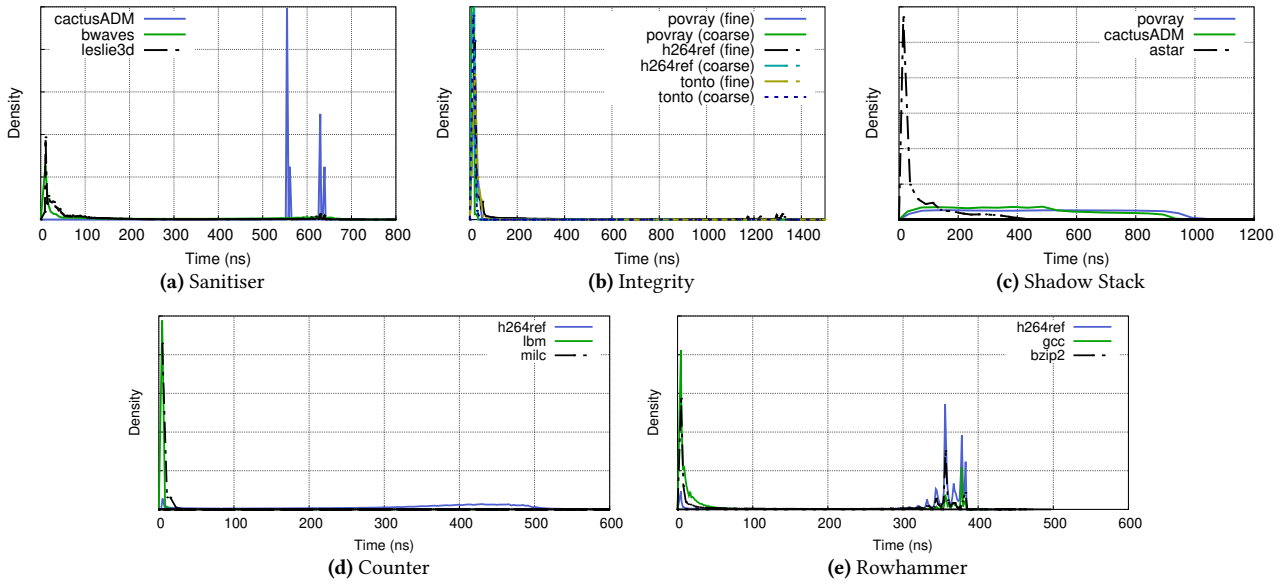


Figure 7. Delay of analysis for events when using eight GPEs to implement each Guardian Kernel.

6.5 Detection Latency

Though detection rates are controlled by choice of technique, and are typically identical to pure software implementations, the decision to decouple techniques previously implemented inline in software (such as control-flow integrity) inevitably increases detection delay. This is acceptable as long as this delay doesn't allow attackers to exploit the system as a result. A mitigation here is that Guardian Kernels can neither be terminated nor reconfigured, and main-core system calls cannot execute, until GPE queues are empty.

Since GPE queues must be cleared on context switches, this also has performance implications by, if this isn't finished on time, delaying the switch. However, we see that in most cases, the time for an event to make it through the Guardian Council, and thus the time for a queue to empty if logging is stopped ready to context switch, is in the tens of nanoseconds, far smaller than a typical context switch [58].

Delays for analysis of inputs, from being placed in the queue to being removed, when using eight GPEs, are given in figure 7. The data shown here gives, for each technique, a benchmark with high, low, and medium overheads, to show the range of results. The shadow-stack technique (figure 7(c)) exhibits the longest average delays and the most variable delays of all of the techniques. This is due to queues being added to until full, to preserve the locality of the data, as opposed to spreading equally in the other techniques. The position in the queue results in a distinctly box-shaped pattern for cactusADM and povray. However, astar features execution sections with very few function calls and thus a single GPE can keep up with the program, resulting in a spike where most observations are verified in a few nanoseconds (or cycles). Figure 7(b), for both coarse and fire-grained control flow integrity, has a lower average, with almost all observed events removed from queues and checked immediately. The exception is h264ref using fine-grained integrity, where occasionally cache misses are observed that result in higher (1200ns) overheads.

Sanitiser and Rowhammer (figure 7(e) and figure 7(a)) exhibit bimodal delays. This is because with 8 GPEs, both can still be significantly GPE-compute bound. This results in two cases: one where events are removed and checked immediately (with negligible delays of tens of nanoseconds), and one where an entire queue's worth of data must be processed first, resulting in hundreds of nanoseconds latency. This is repeated to a lesser extent with Counter (figure 7(d)), where h264ref is occasionally, but variably, GPE-compute bound, resulting in a wide bump in the hundreds of nanoseconds.

6.6 Summary

Overheads of below 5%, and significantly lower than in equivalent software techniques [2, 18, 19], can be achieved for coarse-grained flow control integrity with 2 GPEs on average, and with 4 for a fine-grained technique. Shadow stack requires 6, a custom load counter 4, and Rowhammer prevention 6. Sanitiser is significantly more expensive at the benefit of providing more comprehensive coverage, requiring 24 GPEs, but with some workloads only requiring 8. This maximal number of GPEs (24) can be achieved at around 2.8% power and 8.4% area overhead. Typically, a Guardian Kernel analyses an observed event in tens of nanoseconds (a few cycles) when enough security compute is available, and hundreds of nanoseconds when underprovisioned.

7 Related Work

A wide variety of defence techniques for system-level security analysis have been proposed and implemented. We split these up into relevant hardware and software defences.

7.1 Hardware Techniques

Hardware Logging Techniques have previously used logging hardware to improve the performance of software security mechanisms. Chen et al. [21] present log-based architectures, which move a several common tracking mechanisms into the hardware to reduce software security overheads. Shetty et al. present HeapMon [54], which combines hardware support with a helper thread to offload memory bug detection, but does not attempt to exploit the significant parallelism advantages that can be exploited by using many small cores for security. Lo et al. [44] add hardware data channels with the ability to drop a portion of their input data, to reduce software overheads. Deng et al. [29] couple a field-programmable gate array (FPGA) to a processor, to allow monitoring on configurable hardware. Compared to FPGA accelerators, the many-core Guardian Council is easier to program, has higher sequential performance, integrates with the memory system without slowing it down, and can achieve fast context switches necessary to allow different properties for different processes.

The Guardian Council takes a different approach because modern silicon scaling allows highly-complex programmable units to be very low cost, provided that parallelism is treated as a first-order design constraint. Compared with techniques using larger cores without exploiting heterogeneous parallelism [26, 44, 54, 62], The Guardian Council vastly reduces power-performance-area overheads for a given coverage.

Hardware Malware Detection Hardware malware detection has recently become a popular research topic, often as part of a hierarchy of detectors from cloud to microarchitecture, reducing the requirements at each level. These use features such as instruction streams or performance counters as input to detectors. Kazdagli et al. [36] present an evaluation methodology, and argue the need for application-specific detection using machine learning. Demme et al. [28] use performance counters as input to detection hardware. Ozsoy et al. [47] use linear classifiers and neural networks, with a channel for all instructions, as a weak classifier to direct more complicated software detection mechanisms. Both papers note that, due to the arms-race effect between malware creators and detectors, updatability of the detection scheme is required. Khasawneh et al. [37] run multiple specialised detectors at once, generated using ensemble learning.

7.2 Software Protection Mechanisms

Many security mechanisms can be implemented in pure software. Though overheads may be higher, it is possible to roll such schemes out more quickly, and backport them to existing systems. Szekeres et al. [56] present a summary of attacks and defences based around memory corruption. They argue that techniques with over 5% overhead [56] rarely see active use, which limits observable behaviour to high-level system calls, or page-level granularity protection.

Abadi et al. [3] implement control-flow integrity to detect unintended control flow. This is achieved using a combination of ID-based checking, to ensure that the location of an indirect branch is to a valid target, and a shadow stack, which checks that returns from function calls go to their original locations. Microsoft have recently introduced Control Flow Guard [50, 57], which looks up each indirect branch in a data structure to ensure it is a valid target. Return Flow Guard [1] improves upon this by adding a shadow stack. Arthur et al. [12] instead replace indirect jumps with multi-way "branch sleds" of possible targets for each location.

Rather than just preventing the negative effects of buffer overflows, it is also possible to detect attempts to access beyond boundaries. AddressSanitizer [52] uses a shadow allocation space to detect overflows. On mallocs, a "poison" zone is placed around allocations, and on a free, the same is added to all of the now-freed memory locations, preventing many cases of overflow. Software fat-pointer schemes [13, 45] give stronger guarantees at greater performance loss.

8 Conclusion

We have presented the Guardian Council, an architecture for programmable security analysis hardware. This features a data channel to feed the work of a main core into many Guardian Processing Elements, to enable widely applicable, high performance, efficient, general-purpose security detection for conventional processors.

We have used our architecture to execute a wide variety of implementations of current defences against attacks in the wild today, at very low overhead. Still, in an architecture where a large amount of low cost parallel computation is available for security, we should validly expect the techniques we can use to expand significantly. Our architecture gives programmers the basic building blocks to explore and implement a very wide range of techniques.

An updatable hardware scheme allows patches to be sent out to allow devices to self-repair, instead of product recall upon discovery of flaws. We believe that this will play a significant role in risk mitigation for future devices. This is just one design in a wide space, but it is clear that some degree of programmable security hardware will be necessary for efficient, reliable, and updatable security guarantees that can keep processors secure for their lifetime, and that effectively exploiting parallelism is necessary to achieve this at overheads that are feasible in real systems.

Acknowledgements

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant references EP/K026399/1, EP/P020011/1 and EP/M506485/1, and ARM Ltd. Additional data related to this publication is available in the repository at https://doi.org/10.17863/CAM.46514.

References

[1] http://xlab.tencent.com/en/2016/11/02/return-flow-guard/, 2016.

[2] https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm, 2017.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[4] S. Ainsworth and T. M. Jones. Parallel error detection using heterogeneous cores. In *DSN*, 2018.

[5] S. Ainsworth and T. M. Jones. Paramedic: Heterogeneous parallel error correction. In *DSN*, 2019.

[6] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(14), 1998.

[7] AnandTech. http://www.anandtech.com/show/8542/cortexm7-launches-embedded-iot-and-wearables/2, 2014.

[8] AnandTech. http://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review/6, 2015.

[9] ARM. http://www.arm.com/products/processors/cortex-m/cortex-m0plus.php.

[10] ARM. Embedded trace macrocell architecture specification. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0014q/index.html, 2011.

[11] W. Arthur, S. Madeka, R. Das, and T. Austin. Locking down insecure indirection with hardware-based control-data isolation. In *MICRO*, 2015.

[12] W. Arthur, B. Mehne, R. Das, and T. Austin. Getting in control of your control flow with control-data isolation. In *CGO*, 2015.

[13] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI*, 1994.

[14] R. Backasch, C. Hochberger, A. Weiss, M. Leucker, and R. Lasslop. Runtime verification for multicore soc with high-quality trace data. *ACM Trans. Des. Autom. Electron. Syst.*, 18(2):18:1–18:26, Apr. 2013. ISSN 1084-4309.

[15] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In *CRYPTO*, 1997.

[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), Aug. 2011.

[17] S. Brenner, D. Goltzsche, and R. Kapitza. Trapps: Secure compartments in the evil cloud. In *XDOMO*, 2017.

[18] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow integrity: Precision, security, and performance. *ACM Comput. Surv.*, 50(1), Apr. 2017.

[19] N. Burow, X. Zhang, and M. Payer. Shining light on shadow stacks. In *S+P*, 2019.

[20] J. Chen and G. Venkataramani. An algorithm for detecting contention-based covert timing channels on shared hardware. In *HASP*, 2014.

[21] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *ISCA*, 2008.

[22] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *SP*, 2010.

[23] S. Chhabra, Y. Solihin, R. Lal, and M. Hoekstra. An analysis of secure processor architectures. *Transactions on Computational Science VII*, 2010.

[24] Clang 10 documentation. Control flow integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[25] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *CCS*, 2015.

[26] M. L. Corliss, E. C. Lewis, and A. Roth. Dise: a programmable macro engine for customizing applications. In *ISCA*, 2003.

[27] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DISCEX*, volume 2, 2000.

[28] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. In *ISCA*, 2013.

[29] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *MICRO*, 2010.

[30] M. Elsabagh, D. Barbara, D. Fleck, and A. Stavrou. Detecting rop with statistical learning of program characteristics. In *CODASPY*, 2017.

[31] A. Francillon, D. Perito, and C. Castelluccia. Defending embedded systems against control flow attacks. In *SecuCode*, 2009.

[32] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *SSYM*, 2001.

[33] C. Giraud. DFA on AES. In *AES*, 2004.

[34] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller. Cache attacks on Intel SGX. In *EuroSec*, 2017.

[35] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver. Sources of error in full-system simulation. In *ISPASS*, 2014.

[36] M. Kazdagli, V. J. Reddi, and M. Tiwari. Quantifying and improving the efficiency of hardware-based mobile malware detectors. In *MICRO*, Oct 2016.

[37] K. N. Khasawneh, M. Ozsoy, C. Donovick, N. Abu-Ghazaleh, and D. Ponomarev. Ensemble learning for low-level hardware-supported malware detection. In *RAID*, 2015.

[38] D. H. Kim, P. J. Nair, and M. K. Qureshi. Architectural support for mitigating row hammering in dram memories. *IEEE Computer Architecture Letters*, 14(1), Jan 2015.

[39] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *ISCA*, 2014.

[40] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[41] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, 1996.

[42] D. Lea. A memory allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[43] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[44] D. Lo, T. Chen, M. Ismail, and G. E. Suh. Run-time monitoring with adjustable overhead using dataflow-guided filtering. In *HPCA*, 2015.

[45] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *POPL*, 2002.

[46] J. Nomani and J. Szefer. Predicting program phases and defending against side-channel attacks using hardware performance counters. In *HASP*, 2015.

[47] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev. Malware-aware processors: A framework for efficient online malware detection. In *HPCA*, Feb 2015.

[48] C. Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[49] P. Rauzy and S. Guilley. Countermeasures against high-order fault-injection attacks on crt-rsa. In *2014 Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2014.

[50] M. Schenk. Bypassing control flow guard in windows 10. https://improsec.com/blog/bypassing-control-flow-guard-in-windows-10, 2017.

[51] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1), Feb. 2000.

[52] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Address-Sanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[53] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

[54] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM J. Res. Dev.*, 50(2/3), Mar. 2006.
[55] A. L. Shimpi. ARM's Cortex M: Even smaller and lower power cpu cores. http://www.anandtech.com/show/8400/arms-cortex-m-even-smaller-and-lower-power-cpu-cores, 2014.
[56] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *SP*, 2013.
[57] J. Tang. An in-depth look at Control Flow Guard technology in Windows 10. https://documents.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf, 2015.
[58] Tsuna's blog. How long does it take to make a context switch? https://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html.
[59] P. Wagner, T. Wild, and A. Herkersdorf. Diasys: Improving soc insight through on-chip diagnosis. *Journal of Systems Architecture*, 75 (Supplement C), 2017.
[60] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical policy enforcement for Android applications. In *USENIX Security*, 2012.
[61] M. Yabuuchi, Y. Tsukamoto, M. Morimoto, M. Tanaka, and K. Nii. 20nm high-density single-port and dual-port srams with wordline-voltage-adjustment system for read/write assists. In *ISSCC*, 2014.
[62] C. B. Zilles and G. S. Sohi. A programmable co-processor for profiling. In *HPCA*, 2001.

A Artefact Appendix

A.1 Abstract

This artefact contains the simulator, a sample benchmark, guardian kernels, and scripts to reproduce and plot experiments from the ASPLOS 2020 paper by S. Ainsworth and T. M. Jones – The Guardian Council: Parallel Programmable Hardware Security, which uses a many-core support architecture to provide programmable security analysis for conventional processors. It can be used standalone for a short run, or with a copy of SPEC CPU2006 to fully re-evaluate the experiments from the original paper.

A.2 Artefact check-list (meta-information)

- **Algorithm:** Control-Flow Integrity, Sanitizer, Shadow Stack
- **Data set:** SPEC CPU2006, bitcount
- **Metrics:** Slowdown, Delay
- **Output:** Simulation metadata, Graphs
- **Experiments:** Simulation
- **How much disk space required (approximately)?:** 3GB
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 2 hours (short), 4 days (full)
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** gem5, MIT.
- **Archived (provide DOI)?:** https://doi.org/10.17863/CAM.46514

A.3 Description

A.3.1 How delivered. Our simulator, Guardian Kernels, scripts and a sample benchmark for the short evaluation are available on Github: https://github.com/SamAinsworth/reproduce-asplos2020-gc-paper.

A.3.2 Hardware Dependencies. Any recent x86-64 system running Ubuntu 16.04 or 18.04 should suffice. Other Linux or Mac operating systems may also work, perhaps with altered package dependencies, but are untested. For the full workload, a system with several cores is advised to reduce simulation time through parallel simulation runs.

A.3.3 Software dependencies. Our simulator and kernels require several package dependencies, which can be automatically installed by our scripts. For the full evaluation, a copy of SPEC CPU2006, in iso format, is required, though the short evaluation can run without.

A.4 Installation

You can install this repository as follows:

```
$ git clone https://github.com/SamAinsworth/reproduce
    ↪-asplos2020-gc-paper
```

All scripts from here onwards are assumed to be run from the scripts directory, from the root of the repository:

```
cd reproduce-asplos2020-gc-paper
cd scripts
```

To install software package dependencies, run

```
./dependencies.sh
```

Then, in the scripts folder, to compile the Guardian Council simulator and the Guardian Kernels, run

```
./build.sh
```

To compile SPEC CPU2006 for the Guardian Council (only needed for a full evaluation), first place your SPEC .iso file (other images can be used by modifying the build_spec.sh script first) in the root directory of the repository (next to the file 'PLACE_SPEC_ISO_HERE'). Then, from the scripts directory, run

```
./build_spec.sh
```

Once this has successfully completed, it will build and set up run directories for all of the SPEC benchmarks (the runs themselves will fail, as the binaries are cross compiled).

A.5 Experiment workflow

For a quick evaluation, once everything bar SPEC is built, from the scripts file run

```
./run_bitcount.sh
```

This will run a short, but representative open source workload which will cause a significant amount of compute from the Guardian Kernels, and can be completed in around 2 hours.

For the full evaluation, with the SPEC CPU2006 workloads from the paper, run

```
./run_spec.sh
```

to resimulate the Guardian Council's experiments. This will take several days, depending on the amount of RAM and number of cores on your system. By default, the script will run as many workloads in parallel as you have physical cores, as long as you have enough RAM to do so. To change this default, alter the value of 'P' inside run_spec.sh.

If any unexpected behaviour is observed, please report it to the authors.

A.6 Evaluation and expected result

To generate graphs of the data, from the scripts folder run

```
./plot_bitcount.sh
```

or for the full evaluation:

```
./plot_spec.sh
```

This will extract the data from the simulation runs' m5out/stats.txt files, and plot it using gnuplot. The plots themselves will be in the folder plots, and the data covered should look broadly similar to the plots for figures 4 and 7 from the paper. As for the small evaluation, bitcount was not included in the original paper, we have provided sample data and results in the folder sample_plots.

The raw data will be accessible in the run directories within the spec or bitcount folder, as stats*.txt and delays*.txt.

If anything is unclear, or any unexpected results occur, please report it to the authors.

A.7 Experiment customisation

Workloads New workloads can be run with each of the Guardian Kernels provided in the artefact. An example of how such a workload should be compiled is given in the Makefile of the bitcount directory. To run a workload on the gem5-guardian simulator, use the scripts in scripts/gem5_scripts, after exporting the BASE variable:

```
export BASE=*YOUR_ARTEFACT_ROOT*:
```

The *nofwd variants can be used to run full short workloads. The others are used to fast forward and sample (as is necessary for longer workloads such as SPEC).

Workloads should be for Aarch64 or Aarch32, and statically linked, to run on the simulator. For shared-memory Guardian Kernels (Sanitizer in our existing set), you must compile in the allocator: see bitcount's Makefile for more information.

Guardian Kernels You can create new Guardian Kernels to evaluate on the simulator. These are written as standard C/C++ programs, with custom instructions (typically implemented as inline ASM, but can be imported from the m5ops list - see the example kernels for more information) for FIFO queues and setup. The Filter and Mapper are programmed with secmap.ini files in the root of your simulation run directory. An example of how this is programmed is given in guardian_kernels/example_filter_map.ini. Multiple kernels can be run simultaneously by adding further lines to secmap.ini.

Simulator Much of the code for the Guardian Council is implemented in gem5-guardian/src/mem/cache/securelogentry.hh and .cc. The commit path of the O3CPU (src/cpu/o3/commit_impl.hh) can be altered to add further observation channels. If you would like further information on modifying the simulator, please contact the authors.

A.8 Notes

- We had an issue on recent Linux kernels with compiling m5threads (for the Sanitizer guardian kernel). To this end, the make command is commented out in our buildscript, and the object file for m5threads preshipped. If this causes issues, please try to rebuild m5threads (guardian_kernels/sanitizer/m5threads-master) and if that doesn't solve the issue report it to the authors.
- You may have to specify a toolset when SPEC is being built. We chose linux-suse101-AMD64.
- The gnuplot scripts will issue several warnings for the small evaluation - these can be ignored, and are caused by using the same scripts as the full evaluation.

A.9 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging