D-GUARD: Thwarting Denial-of-Service Attacks via Hardware Monitoring of Information Flow using Language Semantics in Embedded Systems

Garret Cunningham, Harsha Chenji, David Juedes, and Avinash Karanth
School of Electrical Engineering and Computer Science, Ohio University, Athens, OH 45701
Email: {gc974517, chenji, juedes, karanth}@ohio.edu

***Abstract*—As low-level embedded systems are vulnerable to attacks that exploit flaws in either hardware or software, it is essential to enforce secure policies to protect the system from malicious instructions that significantly alter program behavior. To improve efficiency of implementation, high-level secure policy languages have been defined such that the policies can be directly synthesized into hardware monitors. However, the language semantics define policies that are static throughout the program execution which limits the flexibility. Moreover, secure policies target processor pipelines and not the network-on-chip (NoC) connecting several processor where denial-of-service attacks could originate.**

In this paper, we enable dynamically reconfigurable security policies through a high-level language called D-GUARDthat target both processor pipeline and NoC architecture in mutlicore embedded systems. Alongside static policies, D-GUARD's semantics support policies that dynamically change behavior in response to program conditions at runtime. In addition, we also propose policies to thwart denial-of-service attacks by rate limiting the packet flow into the network using the same dynamic policies expressed by D-GUARD. We describe a Verilog compiler to support realizing policies as hardware monitors for both processor pipelines and network interfaces. D-GUARD is developed using the Coq proof assistant, enabling the formal verification of policy correctness and other properties. This approach takes advantage of the abstractions and expressiveness of a higher-level language while minimizing the overhead that comes with other general-purpose approaches implemented purely in hardware, as well as offering the groundwork for a formally verified tool chain.

I. Introduction

Securing the safety of low-level manycore embedded systems remains a critical research topic, inspiring diverse approaches [2]–[4], [7], [8] to circumvent the lack of features and abstraction layers that provide security in other systems. At runtime, malicious programs may exploit flaws in software or hardware behavior to gain control. Hardware monitors attempt to detect and prevent such attacks at the point of execution. That is, they monitor data propagated through the pipeline, using tags, and determine whether preventative measures should be enacted based on implemented security policies. For example, a monitor may detect illegal memory accesses and halt execution of the program on detection.

Hardware monitors provide a compelling solution for securing manycore embedded systems, since they often do not rely on higher-level abstraction layers, such as operating

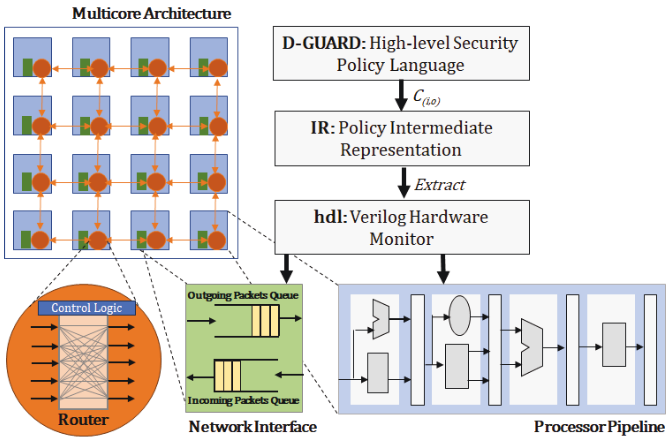


Figure 1: Overview of D-GUARD policy implemented for multicore architecture. The secure policies are applied to the multicore pipelines and network interfaces.

system. However, they often come with a trade-off of performance versus versatility. Architectures like FlexiTaint [7] and MemTracker [8] are highly successful at preventing domain-specific attacks with low overhead, but lack the flexibility to provide a broader coverage of potential policies. More general architectures, like PUMP [4] and PHMon [2], allow such flexibility via programmability. However, as a consequence, the hardware incurs a much more substantial overhead, such as an additional pipeline stage or a complete co-processor, as in the case of Nile [3].

Updating any given secure policy may require a significant tweaking of the architecture, especially in the case of non-programmable or static monitors limiting their scalability. Exploiting high-level language semantics to produce low-level security solutions has demonstrated promise through designs like GARUDA [5]. However, thus far, their semantics lack the expressiveness to design policies suitable to address attacks on highly interconnected systems, such as network-on-chip (NoC) platforms. Prior work has addressed denial-of-service attacks in NoCs in hardware by considering throttling and other rate control mechanisms [1].

In this paper, we attempt to close the abstraction gap further by developing a high-level programming language for designing hardware monitors, named D-GUARDthat targets both processor pipelines and NoC. We reduce overhead using a compiler toolchain from D-GUARD to Verilog, allowing policies to be placed directly into hardware. Moreover, D-GUARD is implemented in the Coq proof assistant. This allows D-GUARD to take advantage of Coq as a carrier language for computations and allows a pathway to formal verification for policies. As shown in 1, the high-level language D-GUARD is used to design policies that ensure safe execution of instructions and packet flows at the network level simultaneously in embedded manycore architectures.

D-GUARD's semantics are based on the bit-stream processing language Ziria [6]. We view monitors as streams that continuously monitor input and produce some output based on policy specifications. This formulation allows us to design polices that can dynamically reconfigure their behavior in response to conditions at runtime. Additionally, our general approach to policy design allows the application of policies outside of an execution pipeline. Given the influence from the network-oriented language Ziria, we demonstrate that D-GUARD is able to design flexible policies that monitor network traffic and program execution simultaneously. We implement a series of policies aimed at preventing denial-of-service (DoS) attacks, while simultaneously ensuring that ongoing computations do not violate certain security guarantees. Our experiments indicated that our selected D-GUARD policies introduce minimal overhead while successfully catching policy violations in both the network interface and execution pipelines.

II. D-GUARD: Low-Level Monitors in High-Level Software

A. The Design of D-GUARD

The high-level D-GUARD programming language is based on the intuition that hardware monitors can act as bitstream processors. That is, one can construct hardware monitors that continuously transform input bit sequences (such as metadata tags, instructions, or packet data) into output bit sequences based on its internal logic. D-GUARD's semantics takes cues from the bitstream processing language Ziria [6]. Hence, we denote D-GUARD policies as `Stream`.

We distinguish two varieties in `Stream`: *transformers* (`T`) and *computers* (`C` r). Transformers are the foundational stream: they act as a black box that converts input of type i into output of type o. Such streams are static: once synthesized, they cannot reconfigure and change their behavior. Hence, we implement computers, which act as transformers with the additional option to return a value of type r before halting execution. Computers allow policies to express conditions at which to reconfigure behavior during runtime. This is primarily represented in D-GUARD by the *staged stream* syntax, $x \leftarrow s_1; s_2$, where s_1 is a stream computer whose return value is saved in x before executing the stream s_2. Stream staging ensures that execution is mutually exclusive: the stream s_2 will not be active until s_1 halts and returns a value.

```
Streams s ≜ upd(λx.e)  (*Update stream.*)
        | done(λx.e)  (*Return result.*)
        | ite e then s1 else s2  (*Branching.*)
        | x ← s1;s2  (*Stream staging.*)
        | s1 ≫ s2  (*Stream composition.*)
        | loop(λx.s)  (*Iterated streams.*)
```

Listing 1: Selected D-GUARD syntax.

Listing 1 demonstrates the full suite of commands in D-GUARD. The quintessential stream is the update stream, `upd`$(\lambda x.\, e)$. Given an input value v, the update stream converts it into an output value according to the expression e: that is, $o = e[x := v]$. The `done`$(\lambda x.\, e)$ stream is an extension of the update stream. This acts as the "return" command for D-GUARD, in which given an input value v, the done stream provides $e[x := v]$ as a return value, and its output as a stream is v unchanged.

We implement control flow in D-GUARD via conditional branching, looping, sequential execution, and staging. The `ite` e `then` s_1 `else` s_2 stream implements conditional branching. The predicate expression e is used to determine whether stream s_1 or s_2 is executed. The `loop`$(\lambda x.\, s)$ stream implements a looping construct, where s is a stream computer. In essence, this stream acts as the stream computer s with an internal variable x to save the return value when s reaches a halting condition. The stream s is then run again, indefinitely. The `loop` command lifts stream computers to transformers. This can be analogized with recursion, or with stream staging of the form $x \leftarrow s; s$ ad inifinitum.

To allow the composition of streams, we implement sequential execution and staging operations. The $s_1 \gg s_2$ stream represents sequential execution. That is, the stream s_1 converts input of type i into output of type m, which is fed directly as input to s_2, who converts it into output of type o. Either s_1 or s_2 may be a transformer or computer, with their composition considered a computer if either component is one, and considered a transformer otherwise. The $x \leftarrow s_1; s_2$ stream represents staging. Unlike the $\gg$ operator, staging enforces mutual exclusivity in stream execution, and further requires s_1 to be a computer. A staged stream acts as the stream s_1 until a halting condition is met, at which point the return value r is stored in x and the stream reconfigures to $s_2[x := r]$.

To facilitate implementation as hardware monitors, we developed a toolchain that compiles high-level D-GUARD policies into synthesizable Verilog code. High-level policies are compiled into an HDL-esque intermediate representation (IR), then extracted into equivalent Verilog code, which can be synthesized and placed into hardware such as an execution pipeline.

Our IR is a simple imperative programming language with variable assignment (`SAssign`) and modularization (`SModule`). Most streams in D-GUARD compile directly to commands in the IR, with loops (`loop`$(\lambda x.\, s)$) and stream staging ($x \leftarrow s_1; s_2$) being less trivial. Loops implement the

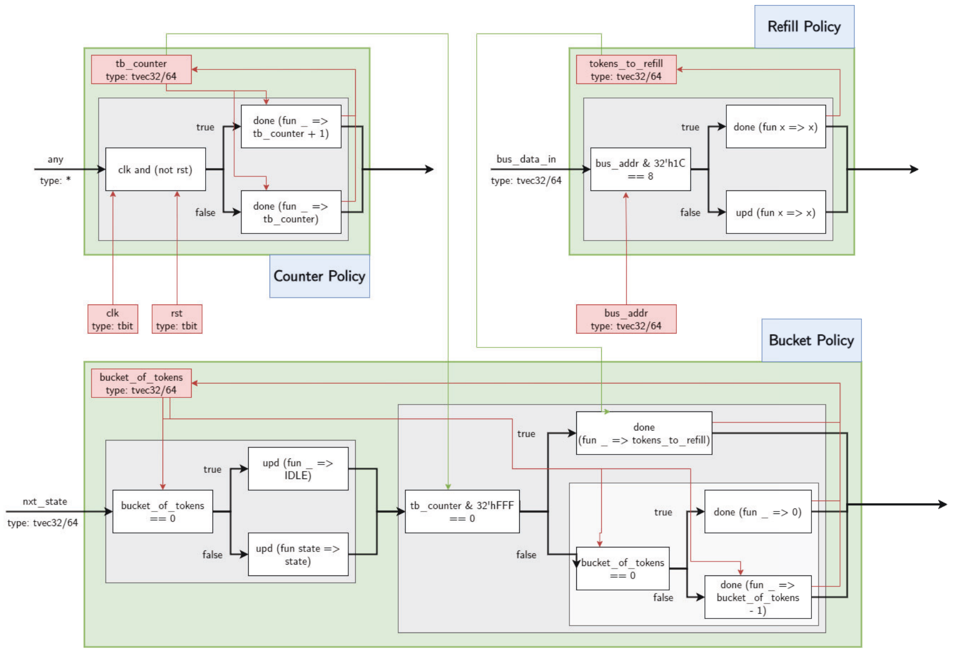


Figure 2: DoS prevention policy dataflow diagram.

	Policy	Behavior	Location
Pipeline	`leak`	Allow only writes to a fixed memory location, preventing all reads.	ID / EX
	`sjsfi`	Combination of `secjmp` (Fig. 3) and SFI policy that forces addresses into a fixed, safe range.	ID / EX
	`shadow`	On function calls, push return addresses onto a 32-deep stack. On return, check the proposed address against the stack, triggering a violation in the case of mismatch.	ID / EX / MEM
	`taint`	Taint memory addresses as write-only. If a read instruction accesses a tainted memory address, a violation is triggered.	ID / EX / MEM
Network	`bucket`	Maintain a bucket of tokens and halt packet transmission if the bucket reaches 0. Decrement tokens from the bucket as packets are processed, then determine refills based on data from `counter` and `refill`.	Network Interface (NI)
	`counter`	Count clock cycles if reset bit is not active.	CPU Clock
	`refill`	Read bus data and save incoming values specifying the number of tokens to refill the bucket with.	Wishbone Bus

Table I: Overview of policies.

```
Stmt c ≜  SAssign x e    (*Assignment.*)
       | SModule m    (*Module creation.*)
       | SSeq s1 s2    (*Sequential execution.*)
       | SITE e s1 s2    (*Branching.*)
```

Listing 2: Selected IR syntax.

loop body s with a variable x that saves the return values of the body. In this sense, we consider a loop as a stream that loops a computer indefinitely. Stream staging splits the contents of s_1 and s_2 into two modules, `SModule` s_1 and `SModule` s_2, respectively. A variable x is assigned the return value of s_1 before executing s_2.

Because the design of our IR is modeled after HDLs, extraction to Verilog is relatively straightforward. All IR commands translate directly to commands and operations in Verilog, with `SModule` introducing extra wires to propagate information about halting and return values. A simple policy is shown in Figure 3, with its respective representations in D-GUARD, the IR, and extracted Verilog.

B. *Case-Study: Denial-of-Service Prevention Policy*

D-GUARD's bitstream-based semantics allow it to express any policy where analyzing a continuous stream of data makes sense. As a substantial case study, we use D-GUARD to write policies for the execution pipeline, tracking information such as memory addresses, and for network traffic, tracking the rates of transferred packets. We detail the latter here, with the complete list of implemented policies listed in Table I. Figure 2 demonstrates the individual policies as dataflow diagram.

```
(* Check immediate jump address range. *)
Definition sec_addr ≜
  BFieldRange ImmAddr (Int.repr 10) (val 0).

Definition secjmp : stream T tvec32 tvec32 ≜
  ite jump
    then (ite sec_addr
      then (upd (λ e ⇒ e))
      (* Convert to nop. *)
      else (upd (λ _ ⇒ 32'h15000000)))
    else (upd (λ e ⇒ e)).
```

(a) `secjmp` in D-GUARD. `jump` checks if the incoming instruction is a jump instruction. `sec_addr` checks if the immediate address of the jump is within a "safe" range.

```
SITE jump
  (SITE sec_addr
    (SAssign o i)
    (SAssign o 32'h15000000))
  (SAssign o i)
```

(b) `secjmp` compiled to our IR.

```
module secjmp(
  input clk,
  input [31:0] i,
  output [31:0] o,
);
  always @(posedge clk) begin
    if (jump) begin
      if (sec_addr) begin
        o ← i;
      end else begin
        o ← 32'h15000000;
      end
    end else begin
      o ← i;
    end
  end
endmodule
```

(c) `secjmp` extracted to Verilog.

Figure 3: Secure jump policy (`secjmp`) in D-GUARD, compiled IR, and extracted Verilog. Prevents jump instructions to "unsafe" addresses.

We implement a token-bucket policy to prevent excessive influx of packets to a network. In D-GUARD, we express this as three policies: `bucket`, `counter`, and `refill`. The primary `bucket` policy:

```
Definition bucket
    (tb_counter tokens_to_refill : tvec32)
    : stream T tbit tbit ≜
  loop (λ bucket_to_tokens : tvec32 ⇒
    ite (bucket_of_tokens == 0)
      then (upd (λ _ ⇒ IDLE))
      else (upd (λ x ⇒ x))
    ≫ ite ((tb_counter & 32'h00000FFF) == 0)
        then (done (λ _ ⇒ tokens_to_refill))
        else (ite (bucket_of_tokens == 0)
          then (done (λ _ ⇒ 0))
          else (done
            (λ _ ⇒ bucket_of_tokens - 1))))
```

observes the current state of the device (e.g., `ACTIVE` or `IDLE`), and decrements the number of tokens in the variable `bucket_of_tokens` for each packet that is successfully transmitted. When the bucket is empty, the state is forcefully changed to `IDLE`. Otherwise, the state is output unchanged. The remainder of the policy determines when to refill the bucket based on a certain window of clock cycles.

The `counter` policy tracks clock cycles and a reset, incrementing the counter variable `tb_counter` every clock cycle, so long as the reset bit is not active:

```
Definition counter (clk rst : tbit)
    : stream T tbit tbit ≜
  loop (λ tb_counter : tvec32 ⇒
    ite (clk and (not rst))
      then (done (λ _ ⇒ tb_counter + 1))
      else (done (λ _ ⇒ tb_counter)))
```

Finally, the `refill` policy saves the number of tokens to refill the bucket with. As a stream, it reads data coming into the bus. The policy determines if the data is intended as a quantity of tokens for refilling the bucket, and if so saves the amount to the variable `tokens_to_refill`. As output, the data passes through the monitor unchanged:

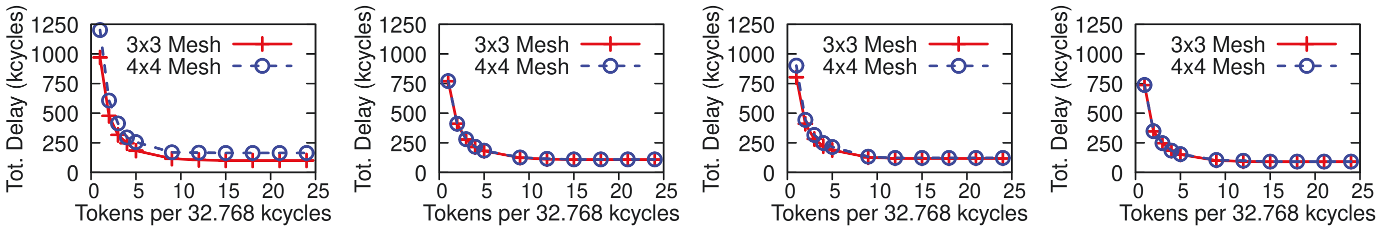
```
Definition refill (bus_addr : tvec32)
    : stream T tvec32 tvec32 ≜
  loop (λ tokens_to_refill ⇒
    ite ((bus_addr & 32'h0000001C) == 8)
      then (done (λ x ⇒ x))
      else (upd (λ x ⇒ x)))
```

Both policies `counter` and `refill` handle external logic for the main policy `bucket`, transmitting data as variables to the policy as a high-level function.

III. Implementation and Performance Evaluation

A. Architecture and Implementation

We use OptimSoc [9] as the evaluation platform, simulating a NoC. The integrated security policies are simultaneously running in each CPU. The NoC consists of 3×3 or 4×4 tiles in a mesh topology. A tile consists of two CPU cores with their own instruction and data caches, usable general purpose memory (32 MB, shared between the CPUs), as well as a network interface (NI) to use the NoC. The NI, reachable by addressing a predefined memory address from either CPU



(a) Traffic Pattern 0 (b) Traffic Pattern 1 (c) Traffic Pattern 2 (d) Traffic Pattern 3

Figure 4: Flow completion delay (with 100% delivery ratio) for four different traffic patterns on 3×3 and 4×4 mesh network-on-chip. X-axes denote the number of tokens refilled every 32,768 CPU cycles. See text for details.

core, is connected using a Wishbone bus to the memory as well as both CPUs. There are four cardinal directions at the NI, which is standard for a 2-D mesh. Flit buffers exist for ingress and egress on all directions, including a local direction towards the CPU.

The token-bucket scheduler is implemented in the outgoing (i.e., towards the NoC) packet buffer of this local direction network interface that is present in each tile. We have modified the NI's System Verilog source code to implement a simple counter that represents the number of available tokens, as discussed in Section II-B. A packet is transmitted by placing it in the egress flit buffer (towards the NoC), but only if the token counter is strictly positive. Upon transmitting, the counter is decremented; it is refilled to a preset value (configurable in software, with default value 20) every N clock cycles (N = 32,768). If a token is not available or if the egress buffers are full, backpressure is applied to the CPU core by not acknowledging any writes on the memory bus, thus stalling the CPU pipeline until the memory write completes (i.e., until the packet is transmitted).

An accompanying compiler and test/debug infrastructure for OptimSoC enables bare metal applications to run on each CPU core on each tile during simulation. Cores can access the NoC through an MPI-like API that is implemented by OptimSoC. We use this facility to create traffic patterns in the network, to measure the effect of the token bucket scheduler on end-to-end performance metrics. Note that no packets are dropped due to the perfect backpressure mechanism; the packet delivery ratio is 100% but the latency increases if congestion is present, as a tradeoff. The reason for using two CPU cores in each tile can now be understood as follows: each core is dedicated to sending or receiving packets to or from the NoC, respectively. The packet transmission delay on a NoC link (a few cycles) is much lower than the interrupt servicing and context switching time on the CPU core (hundreds/thousands of cycles). Thus, it is very hard to saturate all the network buffers and links if a core has to send and receive. By separating send and receive functionality, we can load the NoC to a much higher traffic level.

B. Evaluation

The existence of a token-bucket scheduler (TBS) on each tile should affect the time taken to send a fixed number of packets. For example, if the TBS is configured to allow only one packet every N cycles, then the time taken to send two packets should be 2N cycles - but if the TBS is refilled with 100 tokens, then the delay should only be N cycles. This reasoning applies to a malicious core too; it is backpressured into stalling its pipeline by the TBS on its tile, should it try to send too many packets in a fixed time

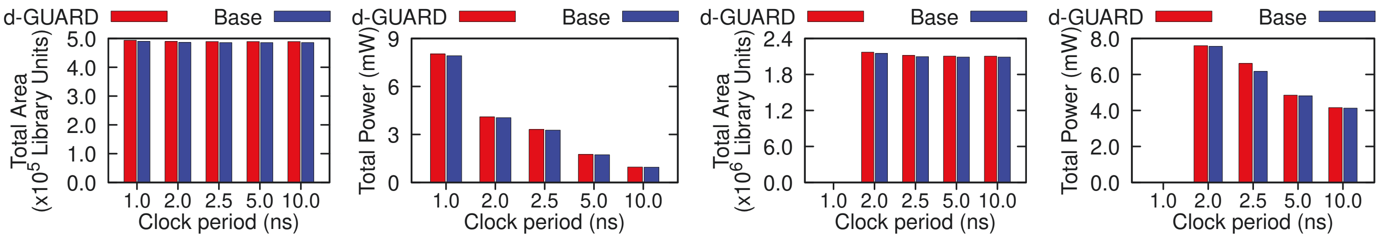
In our evaluation scenario, we measure the total delay as the time taken for each tile in the NoC to send a predetermined number of packets to a predetermined destination tile. We create four different traffic patterns each of which define a destination for a given tile, on both 3×3 and 4×4 topologies as follows:

1) Traffic Pattern 0: the destination for every tile is the tile in the top left (i.e., one of the corners in a mesh)
2) Traffic Pattern 1: tile i sends all its traffic to tile $(i+1)$
3) Traffic Pattern 2: tile i chooses a random tile
4) Traffic Pattern 3: only the tile below the top left sends all its traffic to the tile above it; all other tiles are dormant.

The results are shown in Figure 4 for 20 packets per flow, for each of the above patterns and evaluated on 3×3 and 4×4 NoCs. We vary the number of tokens refilled every N cycles over a range [1,24]. As expected, we observe that the total delay reduces as more tokens become available, which means that more packets can be transmitted per cycle. Traffic Pattern 0, a *concentrator* topology, incurs the highest delay as there is heavy congestion in the corners of the mesh. In contrast, Pattern 1 (Figure 4b) requires most of the tiles to talk to their immediate neighbors, which evenly distributes most of the traffic, relieving congestion and reducing the total delay. Congestion is exacerbated as the total number of tiles in the topology increases (Figure 4a), but only for concentrated traffic patterns.

Pattern 2 (Figure 4c) randomizes the destination, which in general results in uneven load on the NoC. Therefore, we observe that delay for 4×4 is higher than the delay for a 3×3 mesh. Pattern 3 (Figure 4d) is a special test case where only one link in the NoC is active (independent of the size of the NoC). Thus, we expect the delay to be identical for all topologies, which we observe to be true. In conclusion, we have empirically observed the effect of the TBS on the total delay.

In order to compare the hardware overhead of implementing



(a) Synopsys 14nm (b) Synopsys 14nm (c) Nangate 45nm (d) Nangate 45nm

Figure 5: Comparison of total area and total power consumption of a single NoC tile (2 CPU cores and 1 NoC interface) with and without D-GUARD policies with Synopsys 14nm (a, b) and Nangate 45nm (c, d).

the scheduler, we synthesized a single NoC tile (D-GUARD in Figure 5) using Synopsys Design Compiler R-2020.09-SP5-5 at two different synthetic techology nodes of 14nm (Synopsys) and 45nm (Nangate FreePDK). The `mflowgen` framework is used to automate the compilation and EDA workflow. For comparison we have also synthesized an unmodified NoC tile without any of the policies mentioned in this paper ("Base" in Figure 5).

We note again that each NoC tile consists of two OpenRISC 1000 CPU cores (with a modified pipeline for D-GUARDand no modifications for Base) and a NoC interface (with just one token bucket scheduler for D-GUARD and no modifications for Base. The results are shown in Fig. 5 and demonstrate a negligible overhead for D-GUARD. The clock period (speed) was varied over a range from 1 ns (1 GHz) to 10 ns (100 MHz). In both 14nm and 45nm nodes, the total area decreases very slightly as clock speed decreases. A clock period of 1 ns is too low for the 45 nm node; the EDA optimizer was unable to synthesize the circuit without incurring timing violations. However this is possible for 14nm as the dimensions of the cell decreases significantly, allowing for lower propagation delay. Overall, we note the almost negligible overhead of D-GUARD compared to a base implementation without any additional functionality. We can conclude that the hardware overhead of D-GUARD is negligible compared to bulky components of a CPU core such as the instruction and data caches, the packet/flit buffers of the NoC interface, as well as the general purpose memory on each tile.

IV. Future Work & Conclusion

As the design of embedded systems continues to become more complex with multiple cores, they open themselves to new security threats and vulnerabilities that have traditionally targeted desktops and workstations. Low-power embedded systems without the isolation abstractions supported by operating systems are especially vulnerable to exploits targeted at the low-level hardware. We are hopeful that the application of high-level languages for designing low-level security solutions will help close the abstraction gap while maintaining the low overhead necessary to remain practical. Given the high interconnectivity of these systems, flexible and highly general approaches are needed. With D-GUARD, we demonstrate the possibility of interconnected and parallel policies on a NoC, opening the way to more applications targeting other attacks on similar systems. Additionally, since D-GUARD is developed using the Coq proof assistant, policies can be formally verified. This extends to the compilation from high-level programs to low-level Verilog, promising a formally verifiable path from source code to integrated hardware.

V. Acknowledgment

This research was partially supported by NSF grants CCF-1936794, CCF-1901192, CCF-2311544 and CCF-2324645.

References

[1] S. Charles, Y. Lyu, and P. Mishra, "Real-time detection and localization of distributed dos attacks in noc-based socs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4510–4523, 2020.

[2] L. Delshadtehrani, S. Canakci, B. Zhou, S. Eldridge, A. Joshi, and M. Egele, "{PHMon}: A programmable hardware monitor and its security use cases," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 807–824.

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

[4] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight Jr, B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 487–502.

[5] S. Sefton, T. Siddiqui, N. S. Amour, G. Stewart, and A. K. Kodi, "Garuda: Designing energy-efficient hardware monitors from high-level policies for secure information flow," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2509–2518, 2018.

[6] G. Stewart, M. Gowda, G. Mainland, B. Radunovic, D. Vytiniotis, and C. L. Agullo, "Ziria: A dsl for wireless systems programming," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 415–428. [Online]. Available: https://doi.org/10.1145/2694344.2694368

[7] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flexitaint: A programmable accelerator for dynamic taint propagation," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 2008, pp. 173–184.

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

[9] S. Wallentowitz, A. Lankes, A. Zaib, T. Wild, and A. Herkersdorf, "A framework for open tiled manycore system-on-chip," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 535–538.