DAGGER: Exploiting Language Semantics for Program Security in Embedded Systems

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

***Abstract*—Without the isolation abstractions of operating systems, low-level embedded systems are especially vulnerable to attacks that exploit flaws in either software or hardware to gain control of program behavior. Runtime monitors at the hardware level have shown promise towards by identifying malicious instructions and enforcing programmer-defined policy at runtime. However, the efficiency of monitors comes at the cost of ease of implementation, as policies for ensuring the safe execution of software must be defined at the hardware level. To bridge the abstraction gap, high-level security policy languages have been defined with the ability to be synthesized into hardware monitors, but are limited by semantics that only define policies whose behavior remains static throughout a program's execution, which limits the practical use case.**

In this paper, we enable dynamically reconfigurable security policies through a high-level language named DAGGER. Alongside static policies, DAGGER's semantics support policies that dynamically change behavior in response to expert-defined conditions at runtime. Additionally, we introduce a Verilog compiler to support realizing policies as hardware monitors. DAGGERis 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

The ubiquitous connectivity of embedded devices has led to a rapid increase in their usage across a wide range of applications, from Internet of Things (IoT) to System-on-Chips (SoC) platforms. Embedded system security is compromised when malicious code exploits pre-existing software flaws to subvert execution in order to gain control of program behavior. Low-power embedded systems without the isolation abstractions supported by operating systems are especially vulnerable to software which may exploit low-level access to the hardware in order to violate correctness or security guarantees.

Even when the software is trusted, an attacker can insert malicious code at runtime to compromise system security. This can include code injection attacks, return-oriented programming, buffer overflows, and control-flow hijacking, among others [9, etc.]. Since enforcing security policies entirely in software can be expensive, several recent systems implemented aspects of policies in hardware [2]–[6], [8], [12], [14]. PUMP [6], for example, cached policies in hardware while defining the tag-checking and propagation rules in software to minimize impact on performance and power. Runtime program monitors provide fine-grained control over security policies enforcing control-flow integrity (CFI) and information-flow control (IFC) by granting experts the flexibility to create robust solutions against expected attacks, usually on a per-instruction basis.

Interest in creating hardware-level solutions for ensuring runtime security has been on the rise, seeing the creation of several methods for preventing memory-based and control-flow hijacking attacks [4]. Hardware monitors represent a subset of these designs that enforce security policies by monitoring events in the pipeline during program execution. Such approaches typically introduce significant overhead either by inserting additional pipeline stages, as in the case of PUMP [6], or by taking a hybrid approach that uses software solutions, such as the reliance on the operating system in PHMon [4].

While hardware monitors are effective in the variety of attacks they can prevent, they are not ideal for systems that run with limited hardware resources or otherwise lack the abstractions of an operating system, such as low-level embedded systems. GARUDA [10] attempted to resolve this through a language for defining policies that are synthesizable to hardware and can be inserted into any pipeline stage. This offered a middle ground between the flexibility of programmer-specified policies and the efficiency of monitors that minimize area and power overhead. GARUDA, however, is limited by semantics that cannot support dynamically reconfigurable policy in response to the data being monitored. This implies that GARUDA is unable to prevent attacks like Spectre variants through techniques like Speculative Taint Tracking [18], which require tracking information about speculative execution and tainted instructions to decide the appropriate policy.

In this paper, we propose a high-level policy language called **DAGGER** for defining runtime security policies that can be synthesized as hardware monitors. The advantage of this new language is the ability to dynamically change policies based on program's runtime behavior. To translate high-level policies into low-level hardware monitors, we introduce a compiler from DAGGER to synthesizable Verilog modules. The primary abstraction of DAGGER's semantics is to consider hardware monitors as *streams* that transform data, where we adopt ideas from the bitstream processing language Ziria [11] to define the semantics. We introduce the semantics of of DAGGER's data streams, expanding on what streams mean in the context of enforcing security policies through hardware

monitors and adapting theory from Ziria. We elaborate on our compiler from DAGGER to Verilog, including a custom HDL-like intermediate representation (IR). Using OptimSoC [17], an open-source implementation of OpenRISC1000 architecture, we implement four policies (data leak, secure jump and software fault isolation, shadow stack, and taint tracking) on a five-stage MIPS pipeline. We observed marginal overhead ($<$ 1% overhead) in power and area when implemented in 14nm and 45nm technologies and no performance overhead when implemented on BEEBS benchmark suite.

II. Dynamic Policies in DAGGER

In this section we discuss the threat model, followed by an exposition on the design and implementation of DAGGER.

A. Threat Model and Practical Concerns

Our threat model assumes the presence of a passive adversary who can load and run malicious software on a system (e.g., remotely). They can provide malicious input (e.g., a file) to trusted and/or open source software (e.g., a web server) in order to exploit a known vulnerability. Most importantly, they do not have physical access to the target system - they can only exfiltrate the results of the computation using the network, but not through any physical side channels. A practical example of such an adversary is a Heartbleed attack [1], [7], where cryptographic keys can be exfiltrated through the network after an out-of-bounds memory read, following a malicious input provided to an open source web server.

The dynamics of policy design and an active adversary are outside the scope of this paper. It is possible for an adversary to infer or probe the current set of policies by providing certain inputs, which can then be refined to "defeat" the incumbent policy set. We assume in this paper that the system designer is responsible for correctly specifying the policies. DAGGER is only concerned with the *translation* of specified policies into a synthesizable hardware description language.

B. The Design of DAGGER

To support dynamically reconfigurable policies in DAGGER, we extend ideas from the bitstream processing language Ziria [11]. Hardware monitors can be thought of as bitstream processors in the execution pipeline, taking in input data (such as instructions or tags) and producing corresponding output on a per-cycle or per-instruction basis. DAGGER Streams act as bitstream processors that continuously map inputs of type i to outputs of type o. The language distinguishes between two types of streams: *transformers* and *computers*. Stream transformers behave as the archetypal streams, mapping inputs to outputs with no halting conditions. Therefore, transformers are static in their behavior. They cannot be reconfigured to adapt to a running program's behavior: once the behavior of a transformer is defined, it cannot be changed without resynthesizing.

We incorporate the notion of stream computers for this purpose. Stream computers act as transformers until a halting condition is met, at which point a return value is given.

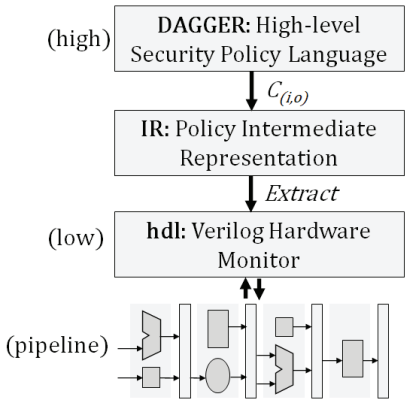


Fig. 1: Full DAGGER toolchain, from high-level policies to HDL compilation and insertion into a pipeline.

```
Streams s ≜ upd(λx. e)   (∗Update stream.∗)
            done(λx. e)   (∗Send result to next stage.∗)
            ite e then s1 else s2   (∗Branching.∗)
            x ← s1; s2   (∗Stream staging.∗)
            s1 ≫ s2   (∗Stream composition.∗)
            loop(λx.s)   (∗Iterated streams.∗)
```

Listing 1: Selected DAGGER syntax.

The central abstraction for DAGGER is the *staged stream* $x \leftarrow s_1; s_2$, which dynamically reconfigures control by executing a computer s_1 until a return value r is given, at which point the entire stream reconfigures to $s_2[x := r]$, which may be itself a computer or a transformer. The streams s_1 and s_2 run as mutually exclusive stages, only handing over control once a programmer-specified condition is met. In the resulting Verilog, each stage corresponds to a module that can be dynamically activated during execution of the security policy.

DAGGER follows the system outlined in Figure 1. It defines a high-level language for security policies named Streams, whose syntax is shown in Listing 1. This is used to write software implementations of security policies such as enforcing software fault isolation (SFI) or speculative taint tracking. DAGGER's Streams can be compiled to a flexible intermediate representation that models hardware descriptor language (HDL) such as Verilog. This allows DAGGER to be extracted to Verilog modules which can be inserted into processor pipeline. In this section, we focus on the high-level language of DAGGER, with discussion of the IR and extracted Verilog modules following in Section III.

Figure 2 visually demonstrates the semantics of DAGGER via diagrams of streams. The archetypal stream transformer is upd($\lambda x.\, e$). This stream maps inputs v to outputs $o = e[x := v]$ for arbitrary expressions e. In comparison, the archetypal stream computer is done($\lambda x.\, e$), which acts as a "return" statement for the language. Similar to upd, it maps input values v to outputs. However, the main output of done is

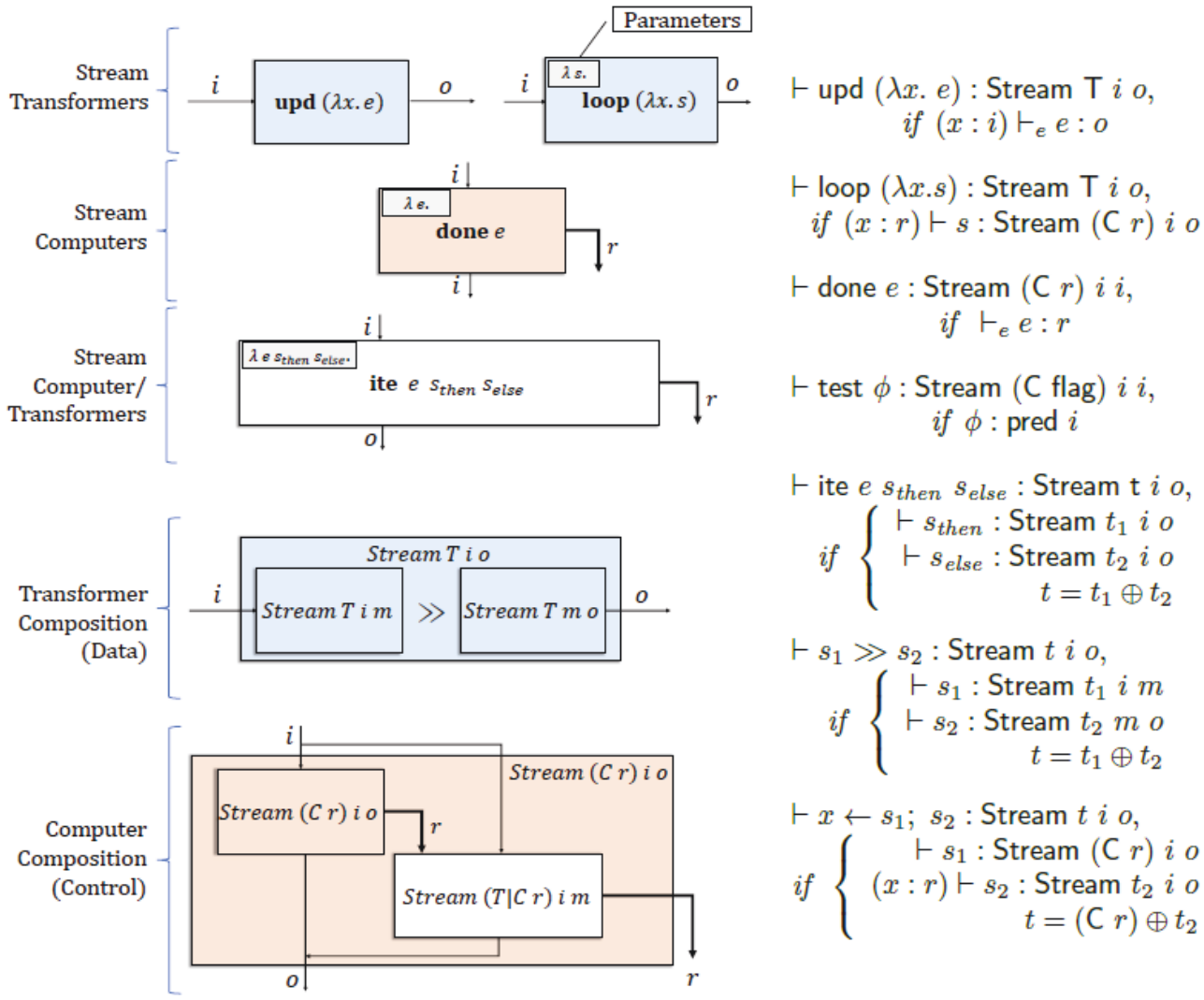


Fig. 2: Dataflow and type inference rules for DAGGER streams.

of a distinct return type r and bound to a variable rather than the stream's output channel. As a stream, done does an identity mapping of inputs to the stream output i.e. $o = v$. Variables are bound using the staged stream $x \leftarrow s_1; s_2$, where the return value of s_1 is bound to x and control is given to s_2, which may depend on x. We also introduce the loop$(\lambda x.\, s)$ construct to lift computers to transformers. This is done by running iterations of the stream computer s and reassigning values returned by s to x. Loops can also be considered as an abstraction of recursion, being equivalently stated using staged streams as $x \leftarrow s; s$.

DAGGER also includes semantics for control flow options, including conditional branching and sequential execution of streams. The ite e then s_1 else s_2 command functions as a conditional branch, where e is an expression that evaluates to a boolean predicate and determines whether s_1 or s_2 is executed. Both s_1 and s_2 can be transformers or computers, wherein the ite stream inherits its type as the product of s_1 and s_2's. Stream composition $s_1 \gg s_2$ denotes sequential execution of streams. This is different from staged streams in that the output of s_1 is directly fed as input to s_2 for immediate execution, rather than the philosophy of mutually exclusive execution used in staging. Similar to conditional branches, s_1 and s_2 can be transformers or computers, with the composition inheriting its type as their product.

We implement the semantics of DAGGER's Streams and IR languages in the Coq proof assistant. This has the advantage of both allowing computations to be handled by the carrier language (Coq) and allowing for the toolchain to be formally verifiable. Thus, policies written in DAGGER can be certified for correctness, assuring trust that policies meet desired security specifications. DAGGER can express a broad variety of security policies for memory safety and control-flow integrity. As an example, consider the following implementation of speculative taint tracking (STT) [19] in DAGGER:

Definition stt : stream T tvec64 tvec64 ≜

```
loop (λ vp : exp tvec64 ⇒
  taint_track  vp  ≫
  ite (zero)
    then (done (λ _ ⇒ vp))
    else (upd (λ e ⇒ e))).
```

```
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.

$$
\begin{aligned}
&C_{(i,o,r)}[\text{upd}(\lambda x.\, f)] \triangleq \text{SAssign } o\ (f\ i)\\
&C_{(i,o,r)}[\text{done}(\lambda x.\, f)] \triangleq \text{SSeq}\\
&\qquad (\text{SAssign } o\ i)\\
&\qquad (\text{SAssign } r\ (f\ i))\\
&C_{(i,o,r)}[\text{ite } e \text{ then } s_1 \text{ else } s_2] \triangleq\\
&\quad \textbf{let } c_1 \triangleq C_{(i,o,r)}[s_1] \textbf{ in}\\
&\quad \textbf{let } c_2 \triangleq C_{(i,o,r)}[s_2] \textbf{ in}\\
&\quad \text{SITE } e\ c_1\ c_2\\
&C_{(i,o,r)}[x \leftarrow s_1; s_2] \triangleq\\
&\quad \textbf{let } c_1 \triangleq C_{(i,o,x)}[s_1] \textbf{ in}\\
&\quad \textbf{let } c_2 \triangleq \lambda x.\, C_{(i,o,r)}[s_2] \textbf{ in}\\
&\quad \text{SSeq (SModule } c_1\text{) (SModule } c_2\text{)}\\
&C_{(i,o,r)}[s_1 \gg s_2] \triangleq\\
&\quad \textbf{let } m \triangleq \text{fresh\_channel() } \textbf{in}\\
&\quad \text{SSeq } C_{(i,m,r)}[s_1]\ C_{(m,o,r)}[s_2]\\
&C_{(i,o,r)}[\text{loop}(\lambda x.\, s)] \triangleq \lambda x.\, C_{(i,o,x)}[s]
\end{aligned}
$$

Listing 3: DAGGER to IR compiler rules.

Adapting a conservative version of the original method [19], the stream stt takes as instructions from the reorder buffer (ROB) as input and produces instructions that are either unchanged or zeroed away for speculative execution. In the former case, the visibility point is manually updated to maintain the same value to prevent speculative execution from advancing beyond the current unsafe instruction. The taint_track stream judges the safety of speculative instructions and masks unsafe instructions to prevent execution. This is done by checking if the instruction is an access instruction and is behind a tainted instruction in the ROB, thus propagating taint information. If deemed unsafe, the instruction is masked to all zeroes. The ite branch checks if this has occurred. If not, then the instruction given as output is identical to the original and executed. If so, then the visibility point is manually kept constant to stall speculative execution until the instruction in question becomes safe. This example shows the ability of DAGGER to dynamically reconfigure policies to prevent unsafe instructions from execution.

III. Compilation to Hardware Monitors

To address the security concerns of technologies like low-power embedded systems that lack software-level security methods, we designed a compiler from the high-level DAGGER language to synthesizable Verilog modules that can be placed directly into a pipeline. Our compiler first compiles DAGGER streams into an intermediate representation (IR), which is extracted to Verilog code. We discuss the semantics of the IR and the compilation tool chain in this section.

Our IR is a simple imperative language that supports variable assignment (SAssign) and modularization of code (SModule) for the purpose of staging. Listing 2 shows the full suite of commands, including sequential execution (SSeq) and conditional branching (SITE). The DAGGER to IR compiler rules are given in Listing 3. While most statements in DAGGER have direct analogues in the IR, the cases for stream staging ($x \leftarrow s_1; s_2$) and loops (loop($\lambda x.\, s$)) are less trivial.

For stream staging, we treat s_1 and s_2 as separate modules. This extracts in a straightforward way to Verilog modules containing the code for s_1 and s_2 respectively, helping to reduce area overhead and allowing the policy designer to optimize the placement of streams within the pipeline. For loops, we compile the code of the loop body s and create a local variable to save the return value from each iteration of the loop. The usage of the term "loop" begins to lose meaning from here on, since it refers to the action of looping a stream computer indefinitely. As hardware, the loop command acts as a standard monitor that implements the loop's body, with the addition of an internal register to save some value for future runs.

We use the IR as a guide for building Verilog modules. This is done using Coq's support for extraction to Haskell. The extracted program prints Verilog code matching the behavior of the original DAGGER stream. Since the IR is designed as an imperative, HDL-like language, marginal effort is needed to convert it into Verilog with the exception of supporting modules. An example of a simple SFI policy at each step of compilation is given in Figure 3. This policy enforces that store instructions write to a sandbox in memory by masking the highest order byte and manually setting it to be 0xA2.

To support the modularization of stream staging, we determine if a module is the active policy, and therefore if it should execute its logic. Two flags are needed to determine if a policy is active; one to see if it has control and one to see if it has been terminated. The module is allowed to run if and only if the control flag is on and the termination flag is off. The termination flag for one stream is used as the control flag for the stream that follows. That is, for the command $x \leftarrow s_1; s_2$, the termination flag in s_1 is used as the control flag for s_2. This propagates control flow information between modules. Likewise, data lines for return values are connected between modules. This allows s_2 to use the value saved into x by s_1.

IV. Implementation and Performance Evaluation

In this section, we describe the implementation & integration of the DAGGER policies in an OpenRISC1000 processor. We then synthesize this processor using Electronic Design Automation (EDA) tools at different technology nodes and measure the power and area of the design.

OptimSoC [17] is an open source implementation of a tiled, manycore system-on-chip based on the OpenRISC1000 architecture. The processor itself is based on the `mor1kx`

```
Definition mask e ≜ (val 32'h00FFFFFF) && e.
Definition force_range e ≜ (val 32'hA2000000) || e.

Definition sfi : stream T tvec32 tvec32 ≜
  ite store
    then ((upd mask) ≫ (upd force_range))
    else (upd (λ e ⇒ e)).
```

(a) sfi in DAGGER. store checks the instruction for store opcodes.

```
SITE store
  (SSeq
    (SAssign internal0 (32' h00FFFFFF & i))
    (SAssign o (32'hA2000000 | internal0 )))
  (SAssign o i)
```

(b) sfi compiled to our IR.

```
module sfi(
    input clk,
    input [31:0] i,
    output [31:0] o,
);
    reg [31:0] internal0;
    wire [31:0] internal0_w;
    assign internal0_w = internal0;

    always @(posedge clk) begin
        if (store) begin
            internal0 = (32'h00FFFFFF & i);
            o = (32'hA2000000 | internal0);
        end else begin
            o = i;
        end
    end
endmodule
```

(c) sfi extracted to Verilog.

Fig. 3: SFI policy (sfi) in DAGGER, compiled IR, and extracted Verilog.

implementation, with code to interface with Network-on-Chip (NoC) routers, memory and other peripherals. The code base is mainly written in SystemVerilog, allowing users to simulate its behavior using tools such as Verilator, or synthesize the design using EDA tools. An accompanying compiler and test/debug infrastructure enable bare metal applications to "run" on this processor during simulation.

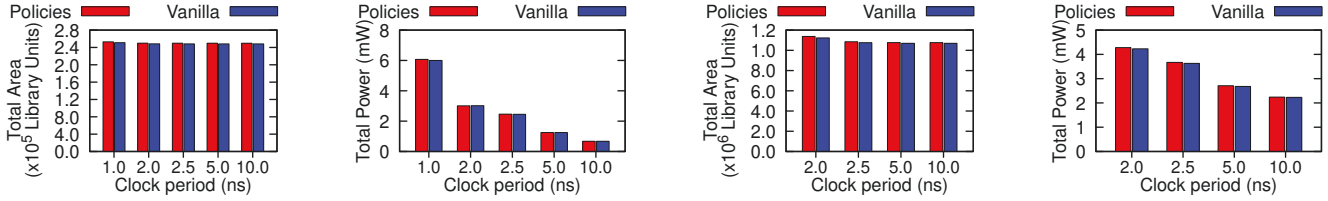
Due to the complexity of the code base, it becomes hard to directly integrate the Verilog code extracted from Coq into the OptimSoC code base. We have therefore used the extracted code as a starting point for integration, and implemented the following policies - data leak, secure jump and software fault isolation (SJSFI), shadow stack, and taint tracking. In all cases, if the policy is violated, an exception is raised and the processor stops execution of code.

1) Data Leak: The memory location `0x60000` can only be written to but any read will trigger a violation.
2) SJSFI: All writes to the memory address range `0x80000-0x8FFF` will be redirected to `0x70000-0x7FFF`.
3) Shadow: A 32-deep stack is maintained in the hardware, onto which return addresses are pushed whenever a function is called. When it returns, the return address is checked against the value stored on the stack. A mismatch will trigger a violation.
4) Taint Tracking: Memory addresses `0x61230-0x61238` are tainted as write-only. These addresses are stored in a 32-deep associative array implemented in hardware. Any attempt to read from any memory location will cause the address to be looked up in this table. If the address matches and is marked as tainted, a policy violation is triggered.

For performance evaluation, we synthesized an OptimSoC-based design after integrating these four policies. To isolate the overhead of the four policies, we also synthesized OptimSoC in the same configuration but without the policies. More specifically, the baseline design (*Vanilla* in Fig. 4) consists of a single core of the `mor1kx` CPU which implements the 32-bit OpenRISC 1000 ISA. This CPU is instantiated on a single tile with 32 MB of RAM, resulting in a single core system that can run code compiled by a `gcc`-based toolchain. The policies are implemented at various places in the pipeline and then synthesized (*Policies* in Fig. 4). The `mflowgen` framework is used to compile these Verilog designs through a Synopsys Design Compiler R-2020.09-SP5-5 EDA workflow. For the standard cell library, we use both the Synopsys 14nm educational library as well as the FreePDK-based Nangate 45nm library.

The results are shown in Fig. 4. They demonstrate an almost consistent 1% overhead for the policies across both cell libraries, for both area and power measurements. The clock period (speed) was varied over a range from 1ns (1GHz) to 10ns (100 MHz). In both 14nm and 45nm nodes, the total area remains almost constant for 400 MHz to 100 MHz speeds, but we see a slight increase for 500 MHz on the 45nm node. This suggests that more probing is required at speeds between 400 MHz and 1 GHz, but we've chosen to keep the speeds constant across different nodes for easy comparison. Power consumption (note the different Y axes ranges) again shows expected behavior, with higher speeds consuming exponentially more power. Still, the larger trend holds: the combined *Policies* implementation still sees a low overhead of about 1%.

Two contrasting factors are responsible for this low overhead. Currently, the policies are implemented as being static in nature, requiring memory addresses and ranges to be hard-coded into the Verilog implementation. There are no special instructions to configure these policies at run time. As a result, very few registers and combinational logic circuitry are needed to implement mechanisms such as detecting whether



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

Fig. 4: Comparison of total area and total power consumption of a single core single tiled OptimSoC-based design with (*Policies*) and without (*Vanilla*) the policies, at different clock periods at different technology nodes: (a) total area for Synopsys 14nm, (b) total power for Synopsys 14nm, (c) total area for Nangate 45nm, (d) total power for Nangate 45nm.

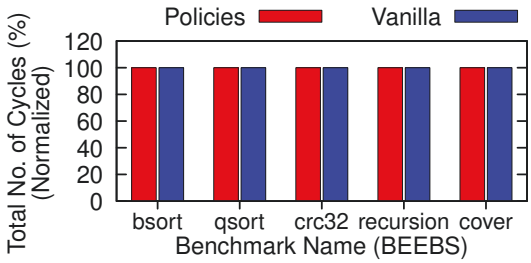


Fig. 5: Measuring the total number of CPU cycles required to fully execute a benchmark program: bubblesort (bsort), quick sort (qsort), 32-bit cyclic redundancy check (crc32), a program with self- and mutual recursion (recursion) and a loop containing many switch cases (cover). In each case, the number of cycles incurred by the *Policies* implementation was normalized to the one incurred by *Vanilla*.

a given memory address falls within a certain range (i.e., matches certain bits). Exception raising and handling circuitry is already part of the *Vanilla* implementation since it's required to support operating systems; policy errors only raise a new type of exception with a new error code, which is again cheaper to implement. Second, there are several other parts of the CPU such as the memory (cache) and clock gating circuitry that dwarf the policy-related circuitry. Therefore, due to the comprehensive nature of the *Vanilla* implementation, the *Policies* implementation is able to leverage existing mechanisms; minimal circuitry is needed to hook into certain parts of the pipeline, perform simple computations, and raise exceptions if necessary.

The OptimSoC platform allows benchmarking of the designed CPU by running compiled bare metal applications without any application/binary interfaces needed for operating systems. The `verilator` framework is used to translate the HDL source code into C++, which is then compiled on the host to yield a design simulator. Then, the bare metal application is given as input to this, allowing us to measure the number of cycles required to execute as well as any input/output of the program (through terminal emulation). We've ported the BEEBS benchmark suite for embedded systems to the OptimSoC platform, and chose five different benchmarks at random. The source code for the programs were compiled and identical binaries were fed to the verilated versions of the *Policies* and *Vanilla* implementations.

The number of CPU cycles were measured in each case (Fig. 5). To easily compare across benchmarks, the values for each benchmark were normalized to the Vanilla implementation. As we can see in the figure, the number of cycles incurred are identical for both implementations, for each of the five benchmarks. This can be easily explained, as the policies are embedded into the processing pipeline and do not incur an additional cycle. For example, memory addresses are computed by adding an immediate (base) to the contents of a register (offset). This computed address is intercepted at the output of the ALU just before it is sent to the memory unit. The policy implementation checks whether the access is legal and raises an exception accordingly. This incurs power/area costs but not timing costs as it leverages the existing memory access infrastructure.

V. Related Work

Designs for hardware monitors that prioritize efficiency and limited overhead come the cost of being domain specific. DIFT [13], FlexiTaint [15], MemTracker [16], and Speculative Taint Tracking (STT) [18] all achieve low performance overhead with their proposed architectures. However, each is constricted to a certain domain of security policies that it can enforce, such as IFC in the case of DIFT, CFI in the case of STT, or memory safety in the case of MemTracker. In order to protect against a broader domain of attacks, multiple solutions have to be implemented simultaneously, potentially compounding overhead costs.

In contrast, designs that prioritize flexibility, like PUMP [6], PHMon [4], and Nile [5] can represent a rich class of policies. PHMon, for example, demonstrated capability to prevent a range of attacks on memory and control-flow vulnerabilities, as well as demonstrated ability to conduct hardware-accelerated fuzzing and debugging. The flexibility of PHMon stems from its general design and programmability. This is accomplished with support from a custom OS kernel to interface with the architecture, making a hybrid solution. Likewise, PUMP demonstrated ability to enforce memory and control-flow safety policies through a general, reprogrammable design without the need for an operating system. Instead, the PUMP monitor is added as a new stage in the pipeline.

However, embedded systems may be designed without an operating system to interface with or otherwise lack the additional computing capacity to accommodate an extra stage in the processor's pipeline or an entire co-processor for tag checking.

The GARUDA [10] project attempted to consolidate these two issues by allowing the flexibility of a high-level language for designing policies and the efficiency of hardware monitors by compiling to synthesizable Verilog modules that contain only the logic necessary to execute policies within a pipeline. This reduces the overhead of more general designs like PUMP or Nile, which introduced entire stages or additional processors for the sake of flexibile policy support. However, GARUDA's synthesized monitors are static in design. That is, monitors have to be resynthesized and replaced in the pipeline to change behavior. Thus, policies that require adjusting behavior in reaction to events during program execution are not supported. DAGGER addresses this by introducing a modular design to the high-level policy language and a compiler that generates hardware monitors with the ability to reconfigure in response to programmer-defined conditions at runtime. The result is a flexible, highly expressive language to representing security specifications while limiting the overhead of introducing general architectures that incur unnecessary overhead.

VI. Future Work & Conclusion

Enforcing safety guarantees on modern computing platforms comes with a non-trivial performance cost. Software-level solutions offer immense flexibility by allowing security experts to explicitly counter anticipated attacks. Hardware monitors attempt to strike a middle-ground between the flexibility of software and the efficiency of hardware. DAGGER offers the flexibility of implementing dynamic policies that can be compiled to verilog. Further, policies implemented in DAGGER can be verified in CoQ assistant proof, further validating the design and implementation of the policies. Overall, DAGGER expands on prior work of designing language semantics by designing low-overhead dynamically reconfigurable policies.

VII. Acknowledgment

This research was partially supported by NSF grants CCF-1936794, CCF-1703013, and CCF-1901192. We sincerely thank the anonymous reviewers for their excellent feedback.

References

[1] "CVE-2014-0160." Available from MITRE, CVE-ID CVE-2014-0160., Dec. 3 2013. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160

[2] J. Coburn, S. Ravi, A. Raghunathan, and S. Chakradhar, "Seca: Security-enhanced communication architecture," in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 78–89. [Online]. Available: https://doi.org/10.1145/1086297.1086308

[3] A. P. Deb Nath, S. Ray, A. Basak, and S. Bhunia, "System-on-chip security architecture and cad framework for hardware patch," in *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2018, pp. 733–738.

[4] 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.

[5] 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.

[6] 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.

[7] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, ser. IMC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 475–488. [Online]. Available: https://doi.org/10.1145/2663716.2663755

[8] F. Restuccia, A. Meza, and R. Kastner, "Aker: A design and verification framework for safe and secure soc access control," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE Press, 2021, p. 1–9. [Online]. Available: https://doi.org/10.1109/ICCAD51958.2021.9643538

[9] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, mar 2012. [Online]. Available: https://doi.org/10.1145/2133375.2133377

[10] 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.

[11] 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

[12] C. Sturton, M. Hicks, S. T. King, and J. M. Smith, "Finalfilter: Asserting security properties of a processor at runtime," *IEEE Micro*, vol. 39, no. 4, pp. 35–42, 2019.

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

[14] B. Tan, R. Elnaggar, J. M. Fung, R. Karri, and K. Chakrabarty, "Toward hardware-based ip vulnerability detection and post-deployment patching in systems-on-chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 6, pp. 1158–1171, 2021.

[15] 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.

[16] 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.

[17] 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.

[18] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 954–968.

[19] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data," in *MICRO-52*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 954–968. [Online]. Available: https://doi.org/10.1145/3352460.3358274