GARUDA: Designing Energy-Efficient Hardware Monitors from High-Level Policies for Secure Information Flow

Seaghan Sefton, Taiman Siddiqui, Nathaniel St. Amour, Gordon Stewart, Avinash Karanth Kodi
School of Electrical Engineering and Computer Science, Ohio University, Athens, Ohio 45701
{ss557415,ts891317,ns196414,gstewart,kodi}@ohio.edu

Abstract—Runtime monitors detect vulnerabilities in embedded systems by running alongside untrusted software in order to detect violations of security policies as they occur, ideally with minimal overhead. Prior work has demonstrated language support for largely static security policies implemented using lattices and tag-based monitors. However, compiling high-level policies to modular hardware monitors that can implement a wide variety of security policies with minimal power has not been previously proposed.

In this paper, we present a high-level security policy language, GARUDA, together with a compiler from GARUDA to Verilog, that enables the modular construction and composition of security hardware runtime monitors for a variety of security policies, including software fault isolation (SFI), secure control flow (SCF) and dynamic information flow via taint tracking. Unlike prior approaches in which the hardware monitors check all instructions, our hardware monitors are activated on-demand by the security policies which reduces the energy consumption. We perform experiments on Sniper, a full system multicore simulator, to evaluate the energy and performance tradeoffs of the security policies we have implemented so far. The policies are tested across a range of Splash-2 benchmarks.

I. INTRODUCTION

Low-power embedded systems without the isolation abstractions supported by operating systems are especially vulnerable to untrusted software, which may exploit low-level access to the hardware in order to violate correctness or security guarantees. Even when the software running on such platforms is trusted, a clever attacker can insert malicious code at runtime to compromise system security (via code injection attacks, buffer overflows, cache probing, or control-flow hijacking). Several software and hardware solutions have been proposed to overcome these system vulnerabilities, including information-flow control, the enforcement of control-flow integrity (e.g., via static or dynamic analysis), and runtime program monitoring.

Information-flow control (IFC) applies at multiple levels of the system stack – from the language level to the hardware level. Dynamic information-flow control tags input data from “unsafe” channels and propagates the tags through the rest of the computation (any data derived from unsafe data is also tagged as unsafe). Several recent efforts have demonstrated the value of propagating tags during execution to catch information-flow violations as they occur. Since enforcing such policies in software is prohibitively expensive and often ineffective, several of the recent systems implemented some policies at least partially in hardware. PUMP [1], for example, exploits temporal and spatial locality by caching the policies (rules) in hardware while defining the tag-checking and propagation rules in software, thereby minimizing the impact on performance and power. This allowed PUMP to support fully software-defined policies with unbounded metadata and low overhead.

A control flow integrity (CFI [2]) policy ensures that the control flow of a software application satisfies some predetermined property (e.g., the program’s dynamic behavior respects its static control-flow graph, or the program never jumps outside a sandboxed region of the address space). Static analysis of a program’s source code – or its associated binary – is one method by which to enforce CFI. However, static analyzers can be incomplete (not all program properties are decidable at compile-time) and may occasionally generate false positives (i.e., they may raise a violation of the policy when no such violation would occur at runtime). Runtime program monitors, on the other hand, enforce CFI by inserting themselves into the control flow of untrusted software, either permitting or denying the untrusted software’s attempt to execute a security-relevant action such as a function call or branch.

While runtime program monitors provide fine-grained control over the implementation of security policies like CFI and information flow, in practice these policies tend to grow complex as requirements change in response to new attacks. Moreover, the presence of contending security policies (a CFI and information-flow monitor running on the same machine, for example) can lead to unpredictable errors. Sophisticated techniques are required to guarantee the correctness not only of individual runtime monitors, but also of the composition of such monitors, together with application-level guarantees that the resulting composed monitors will detect and isolate all violations of the high-level composed policy. A critical challenge is to develop security policies that are both simple enough to comprehend and enforce, yet provide sufficiently strong guarantees against undiscovered software vulnerabilities, all while minimizing impact on processor performance. Previous work on language-based support for runtime monitors, such as micro-policies [3], defined formal semantics and derived high-level correctness guarantees for PUMP-style tag-checking policies but did not focus specifically on either the modularity or power-efficiency.

This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2018 and appears as part of the ESWEEK-TCAD special issue.
of such policies.

In this paper, we describe a new high-level security policy language, GARUDA, together with a compiler toolchain that maps GARUDA’s security policies to power-efficient and low-latency Verilog modules. GARUDA enables the modular construction of energy-efficient secure hardware runtime monitors that provide fine-grained control over the implementation of security policies while providing application-level guarantees. The security policies we consider in this work include software fault isolation (SFI), secure control flow (SCF, a control-flow integrity policy), and dynamic information flow (taint tracking). In order to minimize the dynamic power consumed by the hardware monitors, we activate the monitors on-demand when security relevant instructions are executed. We perform experiments on the full-system simulator Sniper to evaluate the energy and performance tradeoffs of the security policies we have implemented so far, across a number of Splash-2 benchmarks. To summarize, the major contributions of this work are:

- **Modular security policy language**: We define a new high-level security policy language, GARUDA (Section II) that is expressive (we have so far implemented policies ranging from software-fault isolation to taint tracking), modular (GARUDA policies are composable by design), and compilable to power-efficient hardware.

- **Compiler toolchain**: We implement a compiler from GARUDA to Verilog (Section III), demonstrating that policy programs in GARUDA can be efficiently implemented in hardware.

- **Energy-efficient hardware monitors**: GARUDA policies can be inserted directly into hardware pipelines, with low overhead. We measure the simulated overheads of three GARUDA policies with respect to both energy and instruction per cycle (IPC) for varying number of multicores and frequencies.

**Organization**: In the next section (§II), we introduce the GARUDA methodology and policy language with reference to a few examples. Section III provides an overview of the compiler toolchain. Sections IV and V present details of the case-study policies we have implemented so far and evaluate their overhead across a range of benchmarks. Sections VI and VII describe related work and conclude.

### II. GARUDA BY EXAMPLE

Figure 1 presents the components of the GARUDA system. At the top (high), we define a high-level security policy language, pol, in which we build modular implementations of security policies such as software fault isolation (SFI). We compile high-level policies to an imperative intermediate representation modeling a subset of hardware description languages like Verilog, which we then extract to Verilog modules that can be inserted into hardware pipelines. We present GARUDA’s syntax first, then describe how the language works with reference to a few example policies.

![Fig. 1: GARUDA compiles high-level security policies (pol) through a policy intermediate representation (IR) to generate Verilog modules (hdl) that do runtime monitoring of CPU pipelines.](image)

Listing 1 gives the syntax of the GARUDA high-level policy language, as defined by the inductive datatypes binop (binary operators), pred (predicates), and pol (policies).

The binary operators are standard and have the usual bitwise semantics as in Verilog. OXor, for example, is bitwise exclusive or of two 64-bit unsigned integers. OShr, OShru, and OShl are shift operations with OShru being unsigned.

The datatype pred defines the abstract syntax of a small language of predicates over 64-bit unsigned integers. BBinop lifts a binary operator, say b, to two predicates, say e1 and e2. In the definition of the datatype, the declaration

\[
\text{BBinop : binop \rightarrow pred \rightarrow pred \rightarrow pred}
\]
says that BBinop is a function (technically, a constructor) taking as arguments a binary operator and two predicates and producing a new predicate (BBinop b e1 e2) as result. BZero is the predicate that always fails, or returns false, for all inputs. BNeg is predicate negation. BField f v, for fld f and bit vector v, returns true if the value of an input at field f equals v, and false otherwise. The semantics of the BField predicate relies on two auxiliary definitions, offset and size, which define, respectively, the offset and size in bits of the fields f that may appear in predicates.

The datatype pol, for policy, defines the GARUDA high-level security policy language, mapping 64-bit inputs to 64-bit outputs. The first syntax constructor, PId, behaves as the identity function mapping inputs to equal outputs. PDrop, by contrast, simply discards its inputs, never producing any outputs. The predicate PTest e p checks whether the input satisfies a predicate e, then behaves either as policy p if the predicate e succeeded, or as PDrop otherwise. The PUpd constructor makes it possible to embed arbitrary intermediate-representation expressions into the high-level policy language, by applying a user-defined function (of type id TVec64 → exp TVec64, implemented in the meta-language Coq) to the input (TVec64 is the intermediate-language type of 64-bit unsigned integers). PChoice and PConcat are the policy language’s only binary operators: PChoice nondeterministically chooses between two policies, say p1 and p2, whereas PConcat p1 p2 implements sequential composition (p1 then p2).

Example 1: As an example of a GARUDA program, consider the following small policy for preventing untrusted user programs from jumping to a particular address range (such a policy might be useful to enforce isolation on a low-power embedded machine without virtual memory support):

\[
\text{Definition sec_jmp : pol} \triangleq \\
\text{PChoice} \\
(\text{PTest op_j (PTest (BNeg sec_addr) PId)) \; // \; } p_1 \\
(\text{PTest (BNeg op_j) PId}). \; // \; p_2
\]

Policies in the GARUDA language define functions mapping input bit vectors (of size 64 bits) to output vectors (also of size 64 bits). The sec_jmp policy, which maps instructions to instructions, is built by composing (PChoice) two independent sub-policies, \(p_1\) and \(p_2\). Sub-policy \(p_1\) first tests (PTest op_j) whether the input to the policy (a 64-bit vector) satisfies a predicate, \(op_j\), which tests whether an instruction word in the low-order 32-bits of the input vector has a jump opcode (in MIPS, either J or JAL). If \(op_j\) succeeds, \(p_1\) proceeds to perform a second test, (PTest (BNeg sec_addr)), which succeeds when the address field of the instruction is not (BNeg) in some address range as specified by sec_addr. If both tests succeed, \(p_1\) behaves as PId, the identity policy that passes its input to its output unchanged. If either test fails, the policy simply discards its input.

The \(p_2\) policy succeeds if the instruction’s opcode is not of jump type (BNeg op_j), in which case \(p_2\) behaves as the identity policy. Thus the composition of \(p_1\) with \(p_2\) yields an overall policy that forwards unchanged non-jump and jump instructions to addresses outside the sec_addr range while dropping all (insecure) jumps to locations within sec_addr.

Example 2: As a second example, consider the following slightly larger policy enforcing a kind of software fault isolation, or SFI [4]:

\[
\text{Definition sfi : pol} \triangleq \\
\text{PChoice} \\
(\text{PTest op_store (PConcat (PUpd mask) (PUpd force_range))) \; // \; } p_1 \\
(\text{PTest (BNeg op_store) PId}). \; // \; p_2
\]

In SFI, the goal is to prevent untrusted code from escaping a sandboxed region, either by writing or jumping to an address outside the sandbox. The policy above prevents just writes outside the sandbox (ignoring control flow). However, it could be composed with the sec_jmp policy described above (or a variant thereof) to also enforce control-flow integrity.

The SFI policy, which works by masking store addresses so that they lie within a particular addresses range, is structured as a PChoice composition of two sub-policies, again called \(p_1\) and \(p_2\). The first policy, \(p_1\), applies only to stores (PTest op_store). The predicate op_store, which checks whether the instruction’s opcode is of store type, is itself implemented as the logical or of a number of simpler predicates:

\[
\text{Definition op_store} \triangleq \\
\text{BField OpCode instr_SBI} \lor \text{BField OpCode instr_SC} \lor \ldots \lor \text{BField OpCode instr_SWR}
\]

together enumerating the various kinds of store instructions in MIPS. (BField OpCode instr_SBI check whether the opcode field of the instruction equals a particular value, in this case instr_SBI). If the instruction is not a store (\(p_2\)), it’s simply forwarded unchanged.

The heart of the \(p_1\) policy are the sub-policies PUpd mask and PUpd force_range. The policy constructor PUpd applies a function \(f\) to the policy’s input \(i\), outputting \(f(i)\) as a result. In the case of the first PUpd, the function mask is defined as:

\[
\text{Definition mask e} \triangleq \text{EBinop OAnd (EVal lfd_mask) e}
\]

where \(e\) is the argument to the function. The application of mask to an input \(e\) results in an output in which \(e\) has been bitwise anded with a bitvector lfd_mask (which here enforces that the higher-order 8 bits of the effective address of the store are set to zero). The second update, force_range, applies a second function that “ors in” the higher-order 8-bit prefix of an address range associated with the current sandbox. The ultimate effect of the \(p_1\) policy is to force out-of-range stores to lie within the sandbox while leaving unchanged those stores that were in range to begin with (in which case the updates have no effect).

Expressivity: The GARUDA policy language, while small, is expressive enough to implement conditional control flow, a feature required in security policies that apply to, e.g., only a subset of their inputs. Consider, for example, the following derived policy implementing conditional evaluation:

\[
\text{Definition PIfThenElse (e:pred) (p1 p2:pol) : pol} \triangleq \\
\text{PChoice (PTest e p1) (PTest (ENot e) p2)}
\]

If \(e\) is true, then PChoice (PTest true p1) (PTest false p2) evaluates through PTest true p1 to p1 whereas if \(e\) is false, the policy evaluates symmetrically to p2. More generally, because GARUDA is embedded in a general programming language
(Coq), we can use all the features of the host language (higher-order functions, bounded recursion, etc.) to construct GARUDA policies at compile time. PIfThenElse as defined above is an example of one such construction, in which we use Coq’s Definition mechanism to define a GARUDA macro implementing conditional control flow. On the technical side, the GARUDA policy language is closely related to Kleene algebra with tests (KAT, [5]) except that it does not support general iteration (Kleene star, *). We discuss the relationship to KAT and related projects such as NetKAT [6] in Section VI.

III. Compiler Toolchain

The previous section presented the GARUDA policy language with reference to a few examples (namely, the sec_jmp and sfi policies). In this section, we describe our toolchain for mapping GARUDA policies to Verilog by compiling the SFI policy from Section II through the GARUDA intermediate representation (IR) to Verilog (Figure 2).

A. Compiling to Verilog

To compile GARUDA policies, we first generate code in an intermediate representation (IR) using the algorithm presented in Figure 3, then translate the IR reasonably directly to Verilog. Figure 2 illustrates this process by presenting each of these steps applied to the concrete SFI policy we presented in Section II – from high-level policy (2a) through IR (2b) to Verilog (2c). We explain each phase in turn.

From GARUDA to IR (2a-2b). The IR we target in (2b) is a small imperative language with features like variable declarations (Local vec internal2), conditional control flow (SITE, or if-then-else), sequential composition of statements (s1; s2), and assignments (SAssign ... internal2 (mask ri)). The SSkip is a nop (it has no effect on the current state).

The outermost operator of the sfi policy (2a) is PChoice, implementing the (nondeterministic) composition of the two sub-policies labeled p1 and p2. Assuming the two branches of the PChoice are disjoint, we can compile the composition of these policies as the bitwise or of their results, an operation implemented by the assignment statement:

SAssign o (EBinop OOr internal0 internal1))

which reads “assign the bitwise or of vectors internal0 and internal1 to output vector o”.

The sub-policies p1 and p2 are both of the form PTest e p, for some predicate e and policy p. The behavior of PTest e p is to first test the predicate e, then behave as p if the predicate e evaluated to true (and to behave as the PDrop policy otherwise, which does nothing – not forwarding its input to its output channel). To compile p2, we generate a conditional statement, SITE, with conditional expression E[ BNeg op_store ], “then” branch SAssign internal1 i and “else” branch SSkip. The code

\[ E[ \text{BNeg op\_store} ] \]

generates an intermediate expression equivalent to BNeg op\_store, the negation of the op\_store predicate which returns true if the instruction in the low-order bits of the input vector i has a store-style opcode.

To compile p1, which is a PTest, we also generate a conditional statement SITE, this time with conditional expression E[ op\_store ]. The “else” branch is SSkip as in p2. The “then” branch is compiled from an expression of the form PConcat p1a p1b and therefore a bit more complicated. The informal behavior of a PConcat policy is to execute p1a first, generating outputs that then become the inputs of p1b. To implement this behavior, we allocate a fresh intermediate buffer internal2 (called m in the compilation algorithm of Figure 3), then compile p1a with respect to input i and output internal2 and p1b with respect to input internal2 and output internal0 (the overall output vector of the outer policy p1). The result is two assignment statements, the first of which masks i – storing the result in internal2 – and the second of which applies the force_range function of Section II to internal2, storing the result in internal0.

From IR to Verilog (2b–2c). Figure 2c presents the Verilog code (partially elided) generated by our translation from the intermediate representation in (2b). This translation is reasonably straightforward: each local variable in our intermediate code gets declared in the generated Verilog module as either an input, output, or internal register variable. The operative part of the module is an always block that uses two parallel if-then-else blocks and blocking assignments to implement the logic of the IR program in (2b). For example, the intermediate code for sub-policy p1, which contained two assignments for p1a and p1b, is translated to sequenced blocking assignments the first of which masks the higher-order 8 bits of input vector i (64’h00FFFFFFFFFFFFFF & i) and the second of which assigns to vector internal0 the result of “oring in” to internal2 the address prefix associated with the sandbox enforced by the SFI policy. The conditional expressions in the if-then-else blocks test whether (or not) the incoming instruction has a store-style opcode. For example, the expression ((i >> 26) & (−1 >> 58)) == 40 tests whether the input vector i contains an instruction with opcode SB = 40 (store byte). The elided expressions in ... test whether the opcode is of another store type (e.g., SW).

Our compiler is implemented inside the Coq theorem prover. To actually generate Verilog code from IR, we use Coq’s extraction mechanism [7] to convert programs in the intermediate representation to Haskell programs which, when evaluated, print the associated Verilog code. This strategy makes it possible to use partial evaluation to compute certain parts of the generated programs (e.g., statically computable arithmetic expressions) at compile time.

B. Composing Monitors

The GARUDA policy language is naturally compositional, by which we mean that it is possible to compose two policies,
**Definition** \( sfi : \text{pol} \triangleq \)
\( \text{PChoice} \)
\( \langle \ast p_1 \rangle (\text{PTest op_{store}}) \)
\( \langle \ast p_2 \rangle (\text{PTest (BNeg op_{store}) PId}) \).

(a) GARUDA SFI policy

**Local** vec internal2 \( \triangleq 0; \) **Local** vec internal1 \( \triangleq 0; \)
**Local** vec internal0 \( \triangleq 0; \)
**Input** vec \( i \triangleq 0; \) **Output** vec \( o \triangleq 0; \)
\( \langle \ast p_1 \rangle \) SITE \( E_i[\text{op_{store}}] \)
\( \langle \ast p_2 \rangle \) SITE \( E_i[\text{BNeg} \text{op_{store}}] \)
\( \langle \ast p_1 \rangle \) (SAssign internal2 (mask \( i \));
\( \langle \ast p_2 \rangle \) (SAssign internal0 (force_range internal2))
\( \langle \ast p_1 \rangle \) SSkip;
\( \langle \ast p_2 \rangle \) SAssign \( o \) (EBinop OOr internal0 internal1)

(b) Intermediate representation of (2a)

```verilog
module SFI(i, o);
    input [63:0];
    output [63:0] o, reg [63:0] o;
    reg[63:0] internal0;
    reg[63:0] internal1;
    reg[63:0] internal2;
    always @(*) begin
        internal2 = 0; internal1 = 0; internal0 = 0; o = 0;
        if (((i >> 26) & (-1 >> 58)) == 40) . . .
            begin
                internal2 = (64'h00FFFFFFFFFFFF & i);
                internal0 = (64'hA200000000000000 | internal2);
            end
        else begin end
            if (((i >> 26) & (-1 >> 58)) == 40) . . .
                begin
                    internal1 = i; end
        else begin end
            o = (internal0 | internal1);
        end
    endmodule
```

(c) Verilog representation of (2b)

Fig. 2: Compiling the GARUDA SFI policy (Section II) through IR to Verilog

```
C_{(i,o)}[\text{PId}] \triangleq \text{SAssign} o (\text{EVar} i)
C_{(i,o)}[\text{PDrop}] \triangleq \text{SSkip}
C_{(i,o)}[\text{PTest} e \ p] \triangleq \text{SITE} E_i[e] C_{(i,o)}[\text{P }] \text{SSkip}
C_{(i,o)}[\text{PUpd} f \ i] \triangleq \text{SAssign} o (f i)
C_{(i,o)}[\text{PChoice} p_1 p_2] \triangleq
    \text{let } o_1 \triangleq \text{fresh\ channel() in}
    \text{let } o_2 \triangleq \text{fresh\ channel() in}
    \text{let } s_1 \triangleq C_{(i,o)}[p_1] \text{ in}
    \text{let } s_2 \triangleq C_{(i,o)}[p_2] \text{ in}
    \text{let } e_r \triangleq \text{EBinop OOr (EVar o_1) (EVar o_2) in}
    \text{let } s_1; s_2; \text{SAssign} o e_r
C_{(i,o)}[\text{PConcat} p_1 p_2] \triangleq
    \text{let } m \triangleq \text{fresh\ channel() in}
    \text{let } s_1 \triangleq C_{(i,m)}[p_1] \text{ in}
    \text{let } s_2 \triangleq C_{(m,o)}[p_2] \text{ in } s_1; s_2
```

Fig. 3: GARUDA compiler. The function \( C_{(i,o)}[\ p] \) compiles a policy \( p \) against 64-bit input channel \( i \) and 64-bit output channel \( o \). \( E_i[e] \) is an auxiliary compilation function mapping predicates over \( i \) to expressions (exp) in the GARUDA intermediate representation.

say \( p_1 \) and \( p_2 \), either in parallel or sequentially to produce a new policy that combines the behavior of \( p_1 \) and \( p_2 \).

Consider, for example, the following sequential composition of the sec_jmp and sfi policies from Section II, implemented simply as the concatenation of the two policies.

say \( p_1 \) and \( p_2 \), either in parallel or sequentially to produce a new policy that combines the behavior of \( p_1 \) and \( p_2 \).

Consider, for example, the following sequential composition of the sec_jmp and sfi policies from Section II, implemented simply as the concatenation of the two policies.

```
TABLE I: GARUDA policies described in Section IV and evaluated in Section V.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>taint</td>
<td>propagates taint bits through ALU operations</td>
</tr>
<tr>
<td>SFI</td>
<td>implements software fault isolation</td>
</tr>
<tr>
<td>SCF</td>
<td>implements a form of control-flow integrity (CFI)</td>
</tr>
</tbody>
</table>
```

**Definition** \( \text{sec}_\text{jmp}_\text{sfi} : \text{pol} \triangleq \text{PConcat} \text{sec}_\text{jmp} \text{sfi} . \)

To generate intermediate code for the composed policy, we allocate a 64-bit vector \( m \), then generate intermediate code for \( \text{sec}_\text{jmp} \) with respect to output vector \( m \) and for \( \text{sfi} \) with respect to \( m \) as input vector. To ensure that this composition makes sense, it should be the case that the two composed policies treat their input and output vectors uniformly when performing field accesses. That is, both policies assume (as is the case, e.g., for \( \text{sec}_\text{jmp} \) and \( \text{sfi} \)) that the high-order 32 bits of the 64-bit input vector store an effective address whereas the low-order 32 bits store the current instruction.

It is also possible to compose policies in parallel. For example, we could use \( \text{PChoice} \) together with two additional \( \text{PTests} \) to compose \( \text{sec}_\text{jmp} \) and \( \text{sfi} \) in the following way:

**Definition** \( \text{sec}_\text{jmp}_\text{sfi} : \text{pol} \triangleq \text{PChoice} (\text{PTest op_{j} sec}_\text{jmp}) (\text{PTest (BNeg op_{j}) sfi}) . \)

The additional \( \text{PTests} \) ensure that the two branches of the \( \text{PChoice} \) are disjoint.

IV. CASE STUDIES

In this section, we describe the three GARUDA policies we have implemented so far (Figure I) and explain where the
three policies would be placed in a typical hardware pipeline, as depicted in Figure 4. This is a typical 5-stage MIPS pipeline (instruction fetch [IF], instruction decode [ID], execution [EX], memory [MEM] and write-back [WB]) with the newly added blocks (SCF, SFI, taint, comparator) shown. We use three colors for activation (purple), control (orange, dotted) and data (green) signals respectively. It must be noted that we are only depicting newly added blocks and signals for implementing the secure policies.

The secure control flow (SCF) policy is similar to the sec_jmp policy of Section II except that it ensures that control-flow instructions – including branches, function calls, and returns – lie outside a particular secured range (e.g., the non-text segments of a process’s address space, or the address space of a competing process on an embedded system without virtual memory). The taint policy propagates tag bits through the operands of ALU instructions, in order to track the provenance of data (in our experiments and taint policy, we leave the source of the taint bits abstract; we also support only two information-flow levels). The SFI policy (also explained initially in Section II) masks the effective addresses of writes to ensure that writes remain within a predetermined fault domain.4 SCF could therefore be used in conjunction with SFI to achieve a form of process isolation on embedded systems without memory protection, by forcing both memory writes and control flow to remain within a particular logical protection domain.

Figure 4 depicts the locations of the three policies described above in a 5-stage pipeline. The SFI policy operates on the effective addresses of write operations and therefore appears after the execution stage. Since the effective address is generated by the execution stage of the pipeline, and the memory address bits needs to be masked before accessing memory so that all writes remain within a predetermined fault domain, one cycle penalty is inserted by this policy for all store instructions. As writes cause the system state to change irrevocably, we believe that this penalty is acceptable for store operations. The taint policy operates on ALU inputs and therefore lies in the execution stage. SCF, which deals with branches and control flow, lies in the instruction decode stage of the pipeline. Both of the control-flow policies loop back to the instruction fetch stage. If a violation is detected in the SCF, the control bit will select the new PC address from the default interrupt handler address instead of the PC and flush the pipeline.

The monitors in Figure 4 need to activate only on instructions of the appropriate type – e.g., writes for SFI and control instructions like branches for SCF. To increase energy efficiency, we activate the monitors by introducing additional Comparator hardware, in the decode stage, that activates each monitor when the Comparator detects an opcode of the appropriate type. The hardware required to implement the comparator module imposes additional power demands, which we analyze experimentally in Section V. We design the comparator block to compare the opcode of every instruction to those affected by the security policies (SCF, SFI or taint) and then XOR all the bits together, followed by a NOR gate. When the output of the NOR gate is high, the selected monitor will be activated and inspect the instruction. As the comparator is in the decode stage, the SCF block is activated immediately. For taint and SFI, the activation bits are carried forward to the execution and memory stage respectively where the blocks will be activated.

The placement of the hardware compiled from the policies of Table I must currently be done by hand, at the level of the generated Verilog code. Nevertheless, monitor placement could in principle be accomplished at the source-code level if we were to integrate GARUDA with a more general hardware description language, e.g. ReWire [8], in which we could express the entire pipeline of Figure 4. While we currently program the Comparator unit of Figure 4 by hand in Verilog, it could be generated automatically from a list of GARUDA policies by statically analyzing each policy program to identify expressions like $P_{Test}$ that activate on particular opcodes.

---

4Reads from locations outside the sandboxed region cannot be leaked out of the sandbox by a write, and therefore need not be checked.
TABLE II: Power, area and timing at 2 GHz from Synopsys.

<table>
<thead>
<tr>
<th>Module</th>
<th>Power (µW)</th>
<th>Timing (nsec)</th>
<th>Area (µm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SFI</td>
<td>6.32</td>
<td>0.16</td>
<td>13.23</td>
</tr>
<tr>
<td>taint</td>
<td>0.404</td>
<td>0.02</td>
<td>1.06</td>
</tr>
<tr>
<td>SCF</td>
<td>29.3</td>
<td>0.32</td>
<td>113.95</td>
</tr>
</tbody>
</table>

V. PERFORMANCE EVALUATION

In this section, we evaluate the power, timing, area and performance (in terms of instructions per cycle (IPC)) of the proposed GARUDA policies. The runtime performance of the high-level security policies compiled from GARUDA to Verilog is evaluated across a range of Splash-2 benchmarks (fmm, radioisoty, cholesky, water-spatial, barnes, and radix) [9]. The modular implementations of the SFI, SCF, and taint policies are evaluated, in addition to the Comparator module (see IV). Each module is compiled in Synopsys Design Compiler (at 40nm technology node) to obtain the power, area and timing overheads of its implementation. For each module, we obtained data from Synopsys at 1, 2 and 4 GHz clock rates respectively. Table II shows the results obtained at 2 GHz clock frequency for the policy modules. From the simulation results, we observe that the power consumed by the modules ranges from 0.4 µWatt for taint to 29.5 µWatt for SCF. The maximum power consumed is for the secure control flow since it targets all control flow instructions such as branches, function calls, and returns. In terms of timing, the longest stage is the SCF which clocked at 0.32 nsec, which is still lower than the 0.5 nsec of 2 GHz clock frequency that was used for running the modules in Synopsys. Similarly, the area overhead of the SCF is also the largest since it requires most combination circuit logic blocks. However, the combined area overhead of the modules is insignificant compared to the total processor area (75.6 mm²) for our simulation configurations.

We used the Sniper Multi-core Simulator (Sniper 6.1) [10] to run its integrated Splash-2 benchmark suite. McPAT version 1.0, which is integrated with Sniper, was used to obtain the power and area model of each benchmark simulation [11]. McPAT power runs for the benchmarks were done at the 40nm technology node to ensure compatibility with results from Synopsys. We also recorded representative traces of each benchmark in Sniper. From the trace files for each benchmark, we obtained the total instruction count, as well as the count of instructions monitored by each policy, i.e. ALU instructions for taint, conditional branches for SCF. Thus, for each benchmark trace, we were able to obtain a breakdown of the instructions monitored by each policy as a percentage of the total instructions in the trace. Figure 5 shows the breakdown for 4 such benchmarks. For example, the fmm benchmark has 30% of instructions that are affected by the SFI policy (stores), 11% of instructions affected by taint (ALU), and 21% of instructions affected by SCF. On average we observe that the SFI policy impacts the largest percentage of instructions across our Splash-2 benchmarks, ranging from 27% to 32%. Branches and control flow range between 17% to 21% and ALU instructions are the least affected at 10% to 25%. While we confined our benchmarks to Splash-2 suite for this work, we plan to extend the work to include various benchmarks used in multicore and embedded systems domain (PARSEC, SPEC CPU 2006 and MediaBench) in the future.

Figure 6 shows the combined power cost of the modules at each tested clock frequency (1, 2 and 4 GHz). From Figure 6, we observe that the highest power is the SCF module when compared to SFI, taint or the Comparator modules at all frequencies. The power consumed was negligible (less than 1%) in comparison to the total power consumed by the processor core(s) and caches for each benchmark we simulated. The combined power costs of the modules given by Figure 6 would be under the assumption that each module runs every instruction in the pipeline. However, since the Comparator module ensures each security policy module will only run for the instructions it monitors, the combined power cost of the modules at each clock frequency would be lower than the respective values in Figure 6. Thus, we can implement a broad class of security policies with a very power-efficient hardware implementation.

The energy cost of running each module was calculated as a product of its power and timing costs obtained from Synopsys. We obtained the energy cost for each module at 1, 2 and 4 GHz clock rates respectively. For each benchmark we simulated, we varied the core frequency (1.2 and 4 GHz) on one core and the number cores at a fixed core frequency (2 GHz), keeping all other simulation configurations the same in each case. Then, we...
Thus, we were able to calculate the energy overhead of each benchmark due to the increasing module power costs, even though the instruction count remains the same. Ocean has a much lower instruction count (around 2.6 million) compared to the other benchmarks, whereas radiosity has a much higher instruction count than the other benchmarks (around 406 million). Hence, the combined energy cost is the lowest for Ocean and the highest for radiosity at each core frequency. We can clearly see that the energy costs double by doubling the clock frequency. For example, at 1 GHz clock the geometric mean is 0.025 J whereas at 2 GHz, the total energy consumed doubles to 0.06 J and is 0.12 J at 4 GHz clock. Even though energy cost would scale with the core frequencies we tested, the overhead would still be negligible compared to overall processor energy consumption. As seen in Figure 7(b), we run each benchmark simulation on varying number of cores to note the scaling behavior of our monitors’ energy cost in a multicore setting. As we run the same benchmark on different number of cores, the total instruction count still remains similar, and therefore the combined energy cost of the modules do not vary significantly with the core count, as seen in the figure. Thus, if an application is run using multiple cores, the energy cost of running it would remain the same even if we were to implement our monitors in each core, as the energy overhead of our implementation would only vary with the instruction count. As expected, the combined monitor energy cost for each benchmark in Figure 7 is negligible compared to the total energy cost of the benchmark simulation. The monitor power cost would scale proportionally with the number of cores if we were to implement them in each core, but the combined monitor power cost (Figure 6) would be negligible compared to the processor power consumption.

The SFI module, implemented between the execution and memory units of the processor pipeline, will add an additional delay cycle to the pipeline for the instructions it monitors. Therefore, to emulate the implementation of SFI, we added an additional cycle to memory store instructions in our Sniper core model. The IPC obtained from simulating the benchmarks using this core model is denoted as SFI in the Figure 8 plots. We compared the SFI IPC with the IPC using the default core model (denoted as Baseline in the Figure 8 plots) across a number of simulations by keeping all other configuration parameters the same for each case. The average decrease in the geometric mean IPC from the Baseline to SFI of the two Figure 8 plots is about 2.17%. This demonstrates that the implementation of the SFI module will have minimal impact on pipeline latency. It must be noted that with the SCF policy, in case of policy violations, the pipeline will be flushed and the PC will be updated with the default interrupt handler address. Therefore, while monitoring does not consume additional cycles, policy violations will terminate the program.

A. Other Methods/Monitors

Quantitative comparison with other systems that do hardware runtime monitoring of security policies is complicated by the fact that each system may implement different features and/or different policies. That said, GARUDA’s energy and area overheads are not large when compared to those of existing systems. The authors of [12], for example, built a hardware monitor that can be connected to an embedded processor pipeline to enforce secure program control flow and instruction stream integrity. The maximum area overhead of enforcing all their policies was 9.07% of processor area with the average overhead being 5.59%. The corresponding area overhead to implement our runtime security policies would be less than 1%. Our policies are also configurable using a high-level language unlike their security policy monitor. In [13], the authors propose to supplement the microprocessor with two hardware mechanisms which work together at runtime to prevent the execution of instructions which have been tampered with and/or result from invalid control flow. A CFI mechanism is first used

![Figure 7(a)](image1.png)

![Figure 7(b)](image2.png)

Fig. 7: (a) shows energy consumed by secure hardware monitors for different frequencies (1, 2 and 4 GHz) running on a single core and (b) shows the energy by the secure hardware monitors consumed by multicores (1, 2 and 4) at 2 GHz for Splash-2 suite.

We used the percentage breakdown of affected instructions obtained from the respective benchmark trace files (mentioned earlier) to find the approximate number of instructions affected by each policy from the total instruction count of each simulation. Thus, we were able to calculate the energy overhead of each module for the benchmark simulation run as a product of its energy cost (calculated from Synopsys) and the number of instructions affected by it. We use the Synopsys data from the same clock frequency as the benchmark simulation in each calculation.

From, Figure 7(a), we can see that the combined energy for the modules increases with the core frequency for each benchmark due to the increasing module power costs, even though the instruction count remains the same. Ocean has a much lower instruction count (around 2.6 million) compared to the other benchmarks, whereas radiosity has a much higher instruction count than the other benchmarks (around 406 million). Hence, the combined energy cost is the lowest for Ocean and the highest for radiosity at each core frequency. We can clearly see that the energy costs double by doubling the clock frequency. For example, at 1 GHz clock the geometric mean is 0.025 J whereas at 2 GHz, the total energy consumed doubles to 0.06 J and is 0.12 J at 4 GHz clock. Even though energy cost would scale with the core frequencies we tested, the overhead would still be negligible compared to overall

![Figure 7(b)](image3.png)
We compare the Baseline with the SFI software verification of security and information-flow properties; turn, highlighting the most closely related work in each area. Such as software-defined networks. We describe each strand in security policy languages and languages for related domains such as information flows as an extension to Verilog [15], [16], [17], [18]. Some enable the programmer to specify and enforce information flows. Others [19], [21] extend theorem provers such as Coq to enable programmers to formally verify arbitrarily complex properties of hardware description programs.

One key difference between systems like SecVerilog [17] and the current work is that GARUDA was designed from the ground up to facilitate the compositional design of security runtime monitors, which we then compile to Verilog. SecVerilog, in contrast, is an extension of Verilog with information-flow relevant security labels that is primarily suited to the specification and verification of information-flow properties of Verilog designs. Because SecVerilog extends Verilog, the user of the SecVerilog system need not learn a new programming language (thus making SecVerilog perhaps more easily adoptable by hardware designers). Verilog perhaps are, however, not naturally compositional.

Other projects, e.g. [8], [21], design new programming languages built specifically for high-assurance hardware design. ReWire [8], for example, exposes a hardware description language embedded in a high-level functional programming language (Haskell). ReWire hardware description programs are compiled to VHDL state machines through an intermediate representation called PreHDL. While less general than ReWire, one advantage of GARUDA is that runtime monitors expressed in the GARUDA policy language are synthesizable as combinational circuits rather than (sequential) state machines, as in ReWire. As we demonstrate in Section V, the combinational circuits generated from our case-study GARUDA policies are both low power and low latency.

### VI. RELATED WORK

There are a number of research projects that do verification of hardware designs before synthesis, at the level, e.g., of hardware description languages like VHDL, Verilog, or Bluespec [14]. Many are extensions to other languages [15], [16], [17], [18]. Some enable the programmer to specify and enforce information flows as an extension to Verilog [15], [16], [17], [19], [20], [18]. Others [19], [21] extend theorem provers such as Coq to enable programmers to formally verify arbitrarily complex properties of hardware description programs.

![Fig. 8: (a) IPC for Splash-2 benchmarks on one core at 2 GHz, and (b) IPC for Splash-2 benchmarks on four core at 2 GHz. We compare the Baseline with the SFI policy since this impacts the IPC.](image)

A. Hardware Verification

There are a number of research projects that do verification of hardware designs before synthesis, at the level, e.g., of hardware description languages like VHDL, Verilog, or Bluespec [14]. Many are extensions to other languages [15], [16], [17], [18]. Some enable the programmer to specify and enforce information flows as an extension to Verilog [15], [16], [17], [19], [20], [18]. Others [19], [21] extend theorem provers such as Coq to enable programmers to formally verify arbitrarily complex properties of hardware description programs.

One key difference between systems like SecVerilog [17] and the current work is that GARUDA was designed from the ground up to facilitate the compositional design of security runtime monitors, which we then compile to Verilog. SecVerilog, in contrast, is an extension of Verilog with information-flow relevant security labels that is primarily suited to the specification and verification of information-flow properties of Verilog designs. Because SecVerilog extends Verilog, the user of the SecVerilog system need not learn a new programming language (thus making SecVerilog perhaps more easily adoptable by hardware designers). Verilog perhaps are, however, not naturally compositional.

Other projects, e.g. [8], [21], design new programming languages built specifically for high-assurance hardware design. ReWire [8], for example, exposes a hardware description language embedded in a high-level functional programming language (Haskell). ReWire hardware description programs are compiled to VHDL state machines through an intermediate representation called PreHDL. While less general than ReWire, one advantage of GARUDA is that runtime monitors expressed in the GARUDA policy language are synthesizable as combinational circuits rather than (sequential) state machines, as in ReWire. As we demonstrate in Section V, the combinational circuits generated from our case-study GARUDA policies are both low power and low latency.

B. Runtime Verification

PUMP [1] is a tag-based software/hardware architecture for monitoring against attacks like return-oriented programming (ROP). The PUMP (or Programmable Unit for Metadata Processing) is a two-level associative map inserted after the memory stage of a single-core pipeline (misses in the second level of this associative map invoke a software miss handler). PUMP policies are implemented as functions mapping, e.g., the current opcode, PC, operands, etc., to a possibly new PC, a result, and an allow? bit that determines whether the operation is allowed to proceed. PUMP policies, while composable (in the sense that multiple such functions can be installed simultaneously in the PUMP cache), are lower level than GARUDA policies. Unlike PUMP, GARUDA supports multiple kinds of policy composition (PConcat and PChoice). GARUDA policies can also be combined in arbitrarily complex ways while remaining compilable to hardware. Prior work, such as [12], straddled the software/hardware divide by using static analysis of source programs to generate hardware runtime monitors designed to flag security violations at runtime.
Recent work on “micro-policies” [3] does formal verification (in Coq) of tag-based runtime monitors by defining such monitors as high-level symbolic machines, thus eliding hardware-specific details of the implementations of the monitors when reasoning about their correctness. In one sense, the micro-policies work farther than we do here, in that they verify – given an implementation of a tag-based monitor for, e.g., dynamic sealing [22] – that the monitor satisfies a higher-level specification in the form of an abstract sealing machine. The GARUDA language can support such proofs in principle – it has a formal semantics in the Coq theorem prover, for example (not elaborated upon in this paper). However, we leave such proofs to future work.

C. Language Design

The design of the GARUDA policy language was inspired in part by work on languages for related domains such as software-defined networking. NetKAT [6], for example, defines a policy language for expressing network protocols (e.g., “block all SSH packets while forwarding non-SSH packets unchanged”) that has its basis in Kleene algebra with tests [5] (the mathematical theory underlying regular expressions, extended to support Boolean tests). In future work, we plan to design and validate an equational theory for GARUDA (as was done for NetKAT) which could be used to justify the correctness of GARUDA policy optimizations. Previous work on languages like Polymer [23] showed that it is possible to build composable security policy languages for software runtime monitors while achieving low overhead on the JVM.

VII. CONCLUSION

This paper presents GARUDA, a high-level language and associated compiler for implementing hardware runtime monitors to enforce security policies such as software fault isolation, control-flow integrity, and dynamic information flow tracking. The GARUDA language is modular; it permits the composition of security policies by design. GARUDA policies are also efficiently implementable in hardware by compilation through an intermediate representation to Verilog, resulting in hardware modules that can be integrated into a standard pipeline. To optimize energy efficiency, the GARUDA architecture activates runtime monitors on-demand so that dynamic energy usage scales only in the number of instructions affected by each policy. To evaluate the hardware generated by the GARUDA toolchain, we measure the energy usage and latency of three GARUDA policies across a range of Splash-2 benchmarks. We believe that GARUDA can be extended to include many more high-level policies across multiple instruction and data streams for both multicore and embedded system architectures.

ACKNOWLEDGMENT

This work was partially supported by a grant from the Ohio Federal Research Network and National Science Foundation grants CCF-1420718, CCF-1513606 and CCF-1703013.

REFERENCES


