Proving Flow Security of Sequential Logic via Automatically-Synthesized Relational Invariants

Hyoukjun Kwon, William Harris, Hadi Esameilzadeh
Georgia Institute of Technology
hyoukjun@gatech.edu, {wharris, hadi}@cc.gatech.edu

***Abstract*—Due to the proliferation of reprogrammable hardware, core designs built from modules drawn from a variety of sources execute with direct access to critical system resources. Expressing guarantees that such modules satisfy, in particular the dynamic conditions under which they release information about their unbounded streams of inputs, and automatically proving that they satisfy such guarantees, is an open and critical problem.**

To address these challenges, we propose a domain-specific language, named STREAMS, for expressing information-flow policies with declassification over unbounded input streams. We also introduce a novel algorithm, named SIMAREL, that given a core design C and STREAMS policy P, automatically proves or falsifies that C satisfies P. The key technical insight behind the design of SIMAREL is a novel algorithm for efficiently synthesizing relational invariants over pairs of circuit executions.

We expressed expected behavior of cores designed independently for research and production as STREAMS policies and used SIMAREL to check if each core satisfies its policy. SIMAREL proved that half of the cores satisfied expected behavior, but found unexpected information leaks in six open-source designs: an Ethernet controller, a flash memory controller, an SD-card storage manager, a robotics controller, a digital-signal processing (DSP) module, and a debugging interface.

I. INTRODUCTION

As the demand for computation increases [22], the gains from general-purpose processors continue to diminish [21, 25, 63]. To address this challenge, research in both academia and industry has begun to focus on developing *programmable accelerators* [12, 13, 37, 42, 51, 52]. Among programmable accelerators, *Field-Programmable Gate Arrays* (FPGAs) provide large gains in performance and energy efficiency. In particular, Microsoft has deployed FPGA's in its data centers to accelerate its web-search service, Bing [51]. Intel recently acquired a major FPGA vendor for 16.7 billion USD to integrate FPGAs in their data-center products and develop new platforms for Internet of Things (IoT) devices [28]. Commercial products that integrate general-purpose cores (i.e., circuit design modules) with FPGAs have already been released by major chip producers [3, 65] and IoT design platforms based on FPGA's are becoming available to crowds of developers [17].

While FPGAs could provide significant benefits for designing next-generation systems, they present novel security issues that have not been adequately addressed. In particular, to implement highly optimized FPGA controllers, a host system typically provides direct read and write privileges to an FPGA. With such privileges, an FPGA can access critical system resources such as memory, the system bus, and even on-chip network devices without mediation from the operating system. As a consequence, an FPGA containing a security vulnerability, perhaps due to aggressive manual optimization, could constitute a critical target for leaking sensitive information throughout a host system. Moreover, practical core designs consist of complex submodules developed independently by multiple sources. If a single, commonly-used module leaks information in an unexpected way, it can affect the information-flow security of all cores designed to use it.

Approaches that track the flow of information dynamically in hardware [35, 61] are not well-suited for running on reprogrammable hardware, on which resources are tightly limited. Approaches that statically analyze if there is any potential flow of information between data containers like program variables or wires in a hardware design [24, 43–45] often cannot be applied to such designs, which are often expected to release sensitive information under particular conditions. Hardware description languages that express allowed flows with declassification [35, 36, 68] require either reimplementing a target core in a new language or embedding security-related descriptions in appropriate positions in the source code of target core.

In this work, we propose a novel automatic approach for verifying that a given core satisfies desired a information-flow guarantee. Our approach consists of (1) a domain-specific language of information-flow policies, named STREAMS, as external documents expressed purely in terms of the interface of a given module and (2) a novel automatic verifier, named SIMAREL, that proves or falsifies that a given sequential core design satisfies a given STREAMS policy.

The key technical challenge in developing our approach was to design a verifier that can automatically synthesize a proof that a core satisfies an information-flow policy with declassification; such proofs are known to be expressible as relational invariants [15]. Automatic verifiers for information flow with declassification that construct the self-composition of program [7, 59] cannot directly be applied to verify systems that may read an unbounded stream of inputs over an execution, such as sequential circuits (see §IV-C). Constructions of relational invariants as invariants of a product program conventionally require semi-manual constructions of a product program as a subset of the Cartesian product of a program paired with itself [6].

To address this challenge, we developed a novel automatic verifier for relational properties of sequential cores, named SIMAREL, based on symbolic model-checking. SIMAREL avoids verifying the product construction of a given core by using a novel procedure that efficiently proves the correctness of all pairs of core runs up to a bounded length and inspects the proof to determine if it contains a proof of the correctness of the core.

To evaluate if STREAMS and SIMAREL can aid the design of secure cores, we used them to express and attempt to verify expected information-flow policies of 12 FPGA cores developed independently for research and production systems. The cores included both applications that would typically be used to process sensitive information, and optimized implementations of control subsystems.

SIMAREL proved that six cores satisfied their policies and found unexpected information leaks in the other six cores, including an Ethernet controller and a robotics controller (see §V). We investigated each core identified be SIMAREL as insecure and determined that their leaks are caused by subtle design mistakes made by either the original designer of a component core or a designer who integrated multiple core modules. Our results indicate that critical cores often are not adequately designed to account for the security requirements of modular designs run on reprogrammable hardware, and that our approach can significantly aid in the development of secure cores.

The rest of this paper is organized as follows. §II illustrates our approach by example. §III reviews previous work on which our approach is based. §IV describes our approach in detail. §V presents the results of verifying information-flow security of a set of cores of critical applications and subsystems. §VI compares our approach to related work, and §VII concludes.

II. OVERVIEW

In this section, we discuss the attack model under which we assume that systems execute, and illustrate by example the problem that we consider and our approach. In §II-A, we describe an I/O management core of a storage-card controller, named `iomanager`, from the open-source repository opencores.org. In §II-B, we express the information-flow requirements of `iomanager` as a policy `ModalRecall` in our language STREAMS. In §II-C, we illustrate how our information-flow verifier SIMAREL automatically verifies that `iomanager` satisfies `ModalRecall`.

A. An I/O manager for external storage

External storage cards are prevalent in smart phones, digital cameras, and drones equipped with cameras. Even if the storage card is secure, the interface logic that reads and writes data to the card can still contain security vulnerabilities. Such a vulnerability can leak important and personal images from the card's host device.

The OpenCores design repository contains an SD card I/O manager for the host side of a card controller, which we refer to as `iomanager`. `iomanager` takes a target address and data from another FPGA core and a read or write command. When `iomanager` receives a read command, it generates a read control signal specific to the SD card and relays input data from the card. When `iomanager` receives a write command, it generates a write control signal specific to the SD card, along with the data to be written.

```
1  module sd_controller_wb (
2    wb_clk_i, wb_set_i, wb_recall_i, wb_arg_ty, wb_data_i // ins:
3  );
4  // output port:
5  output reg [31 : 0] wb_data_o;
6  // internal registers:
7  reg [31 : 0] argument_reg;
8  reg [31 : 0] cmd_setting_reg;
9  reg [31 : 0] status_reg;
10 // update registers to store input values
11 always @(posedge wb_clk_i) begin
12   if(wb_set_i)
13     case (wb_arg_ty)
14       ‘argument   : argument_reg <= ...wb_data_i...
15       ‘command    : cmd_setting_reg <= ...wb_data_i...
16       ‘status_reg : status_reg <= ...wb_data_i...
17       ...
18     endcase
19 end
20 // output values in stored registers
21 always @(posedge wb_clk_i) begin
22   if(wb_recall_i)
23     case (wb_arg_ty)
24       ‘argument : wb_data_o <= argument_reg
25       ‘command  : wb_data_o <= cmd_setting_reg
26       ‘status   : wb_data_o <= status_reg
27       // further, but non-exhaustive, cases
28     endcase
29 end
30 endmodule
```

Fig. 1: An I/O management core for SD card controllers, `iomanager`, given as a fragment of Verilog code.

The complete implementation of `iomanager` consists of nine Verilog modules with 3,673 lines of Verilog code. Each module implements a state machine that generates specific control signals. Some state machines have as many as 10 states. Because `iomanager` has direct access to the SD card and has a complex and stateful implementation, it is a prime target for security attacks.

Figure 1 contains a simplified excerpt from one of the submodules of `iomanager`; the complete implementation of the submodule alone consists of 310 lines of Verilog. The submodule takes as input a clock signal `wb_clk_i`, a signal to store data `wb_set_i`, a signal to recall data stored `wb_recall_i`, an argument-type field on `wb_arg_ty`, and a data value on wire `wb_data_i`. `iomanager` outputs a data value on register `wb_data_o` (line 5). Internal registers `argument_reg`, `cmd_setting`, and `status_reg` store the last argument, command, and status set, respectively (lines 7—9).

When `iomanager` receives a signal on `wb_set_i` (lines 12), it checks the argument type passed in `wb_arg_ty` (line 13). Depending on the argument type, `iomanager` stores a value computed from `wb_data_i` in one of its internal registers (line 13—18). When the input wire `wb_recall_i` is set (line 22), `iomanager` checks the argument type passed in (line 23). If the input type matches some expected constant value, the set of which do not span the range of all possible values for `wb_arg_ty`, then `iomanager` copies the appropriate value from an internal register (line 23—28). `recall` is set when an agent intends to load the last value that it stored when issuing a command (lines 23—30).

```
1:  levels public < sensitive
2:  default level public
3:  wb_data_i has sensitive
4:  !recall => public
```

Fig. 2: An information-flow policy for `iomanager`, written in Streams, named `ModalRecall`.

B. A policy for the I/O manager

One of the primary goals of our work is to develop techniques that specify and verify that a hardware design releases information-flow securely. We assume that an attacker can directly observe inputs and outputs of a core design on particular wires on particular steps of execution, designated in information-flow policies in our language. In practice, an attacker can do so by implementing a hardware module that snoops the output ports of other hardware components and leaks the information. Designers on FPGA systems typically allow modules from mutually-untrusting developers to co-habitate on a single module to minimize design cost.

Our attack model does not require any assumption on the source of a design itself. In practice, a developer may want to verify the security of a design reused or shared from a benevolent but imperfect source. However, in principle, a developer may also want to verify the security of a design provided by an untrusted source, which may contain a Hardware Trojan. Our policies can express expected properties of such systems as well.

One information-flow policy that `iomanager` is expected to satisfy is that it should only reveal the value provided on its input wires on runs in which `recall` is set. If `iomanager` satisfied such a policy, then it could be used in contexts where only a particular agent can send the `recall` signal to review its history until it relinquishes control and the state of `iomanager` is reset. Such a policy, while simple to formulate and referring to a relatively small component of the complete interface of `iomanager`, cannot be enforced by other hardware information-flow languages without specifying the entire design of the system in the language [36], due to the fact that it can only be satisfied by checking a dynamic condition on program inputs. In general, such policies also rely on maintaining correct state related to sensitive information.

In this work, we introduce a policy language, Streams, that can express such policies for cores implemented in a conventional hardware description language. Figure 2 contains a Streams policy for `iomanager`, named `ModalRecall`, that expresses the above requirements for `iomanager` that were stated informally. We define the complete syntax and semantics of Streams in §IV-A, but the intuition behind the use of its constructs in `ModalRecall` can be understood without a complete definition. In particular, `ModalRecall` contains a definition of two levels of sensitive information, `public` and `sensitive`, with `sensitive` denoting information that is strictly more sensitive than `public` information (line 1). Thus, information that the core receives over a `sensitive` input wire cannot, by default, be revealed on a `public` output wire. `ModalRecall` contains a declaration that each wire that is not explicitly labeled is `public` (line 2), and an explicit labeling of output wire `wb_dat_i` as `sensitive` (line 3). However, `ModalRecall` also contains a directive specifying that wires labeled `public` must reveal `public` inputs only on steps when `recall` is not set (line 4).

C. Checking flow-security of the I/O manager

Unfortunately, `iomanager` in fact does not satisfy `ModalRecall`, and as a result may leak information read over wire `wb_data_i` over wire `wb_data_o` without receiving an appropriate signal on `recall`. The cause of the leak in `iomanager` is a result of inexhaustive case logic for determining the values output to `wb_data_o` (Figure 1, lines 21—26). In particular, `iomanager` checks `wb_arg_ty` against values that it *expects* to read, but does not check `wb_arg_ty` against all *possible* values, which could conceivably be provided by an arbitrary environment. Furthermore, each output register, in particular `wb_data_o`, also acts as an implicit storage register. When Verilog matches a register in a case statement and none of the provided cases match, then the circuit does not update any storage registers involved in the case statement. As a result, if an attacker without the authorization to signal `recall` sends a value for `wb_arg_ty` that does not match any of the values checked in lines 21—26, then `iomanager` will provide the last value that it output, which could potentially have been computed from information read on `wb_data_i`.

In this work, we present, along with our policy language Streams, a verification algorithm Simarel that takes a core C and a Streams policy F and either proves that C satisfies F or provides a counterexample that proves that C does not satisfy F. The key property of Streams that enables the design of Simarel is that, by definition of Streams (see §IV-A), a core C satisfies a Streams policy F if each pair of runs of C satisfies a suitable relational property. A significant consequence of this design choice is that if a given core does not satisfy a given flow policy F, there is a pair of runs of C that demonstrate the violation.

In the case of `iomanager` and `ModalRecall`, a pair of runs of `iomanager` that violates `ModalRecall` is any pair of runs that **(1)** read different values only on register `wb_data_i`, **(2)** do not read in their last step an enabling signal on `recall`, and **(3)** produce different values on output wire `wb_data_o`.

Simarel uses this feature of Streams to search for examples that prove that a policy is violated. In particular, for iteratively larger step bounds n, Simarel constructs a propositional formula φ_n for which each model defines a pair of runs of C that violate F and runs a SAT solver on φ_n. If the solver finds a model, then Simarel returns the model as a counterexample for C and F (see §IV-B5).

Given `iomanager` and `ModalRecall`, Simarel constructs a formula that characterizes all pairs of runs of `iomanager` of length two and finds a counterexample for `iomanager` and `ModalRecall` as two runs whose values for `wb_data_i`, `recall`, and `wb_data_o` satisfy each of the above conditions. In particular, one of the runs takes as input a control signal that is not explicitly matched in Figure 1 lines 23—28.

The fact that each STREAMS policy defines a property over all pairs of runs of a core also guides how SIMAREL verifies that a correct core does satisfy a given flow policy. Let `iomanager`' be `iomanager` as given in Figure 1, but patched to include a default case for the case statement at lines 23—28 that sets `wb_data_o` low. The *combined* state of each pair of runs of `iomanager`', extended with an auxiliary variable InsEq that tracks if the two runs have read equal values over `wb_clk_i`, `wb_set_i`, and `wb_arg_ty`, always satisfies a key *relational* invariant. In particular, the invariant establishes that after any pair of runs, if InsEq holds, and in the next step of each run, the run does not read a high signal on `recall`, then the two runs will send equal values to `wb_data_o`.

The above fact is in fact an *inductive* relational invariant of all pairs of runs of `ioman`', extended with InEq. When SIMAREL is given a core C, such as `ioman`', that satisfies a given flow policy F, then SIMAREL verifies that C satisfies F by proving for iteratively larger bounds n that all pairs of runs of C of length up to n satisfy F. SIMAREL inspects the invariants obtained in such proofs to determine if they are in fact inductive proofs that C satisfies F, and if so determines that C satisfies F (see §IV-B3).

III. BACKGROUND

In this section, we review previous work on which our work is based. In §III-A, we establish basic definitions of the hardware systems that we will consider. In §III-B, we review definitions and fundamental results from propositional logic.

A. Hardware core design

We will describe an approach for specifying and verifying the flow security of sequential hardware core designs with direct physical access to sensitive and public input and output channels. Our work is motivated in particular by the recent development of *Field Programmable Gate Arrays (FPGAs)*, which are integrated circuits designed to be programmed and reconfigured by users or a designer after the chip is fabricated. The design of an FPGA core is generally specified in a *Hardware Description Language* (HDL), such as Verilog. In this work, we assume that our verifier has access to a description of a given circuit in an HDL. This description is a circuit that will be implemented using the pre-fabricated logic elements of the FPGA.

FPGAs can interact with their host system over a rich interface that allows them to read information from and send it to memory and critical system devices such as network controllers. In practical deployments, a single FPGA unit commonly is programmed with multiple colocated cores that perform computation on behalf of multiple mutually-untrusting users. To present our approach, we assume that there are fixed sets of propositional variables that model, for a given core, the input wires to the core (denoted $\mathtt{I}$), output wires from the core (denoted $\mathtt{O}$), and persistent state of the core (denoted $\mathtt{Q}$). The union of the spaces of $\mathtt{I}$ and $\mathtt{O}$ is the space of *I/O wires*, denoted $\mathtt{IOs} = \mathtt{I} \cup \mathtt{O}$. The union of the space of I/O wires and state registers is denoted $\mathtt{Wires} = \mathtt{IOs} \cup \mathtt{Q}$. The space of all cores is denoted Cores.

For each $C \in \mathsf{Cores}$, the *transition relation* of C defines how in each step, C uses its current state and input to update its state and generate an output. For each space of propositional variables X (i.e., a *vocabulary*), we denote the space of all evaluations of X as $\mathsf{Evals}[X] = X \to \mathbb{B}$. The initial states $\mathsf{Inits}_C \subseteq \mathsf{Evals}[\mathtt{Q}]$ are the states in which C may begin a run. The transition relation of C is a binary relation $\rho_C \subseteq (\mathsf{Evals}[\mathtt{I}] \times \mathsf{Evals}[\mathtt{Q}]) \times (\mathsf{Evals}[\mathtt{Q}] \times \mathsf{Evals}[\mathtt{O}])$. For $I \in \mathsf{Evals}[\mathtt{I}]$, $Q, Q' \in \mathsf{Evals}[\mathtt{Q}]$, and $O \in \mathsf{Evals}[\mathtt{O}]$, we denote the fact $((I, Q), (Q', O)) \in \rho_C$ alternatively as $(I, Q) \to_C (Q', O)$.

A *run* is a sequence of triples of input evaluations, states, and output evaluations; i.e., the space of runs is $\mathsf{Runs} = (\mathsf{Evals}[\mathtt{I}] \times \mathsf{Evals}[\mathtt{Q}] \times \mathsf{Evals}[\mathtt{O}])^*$. A *trace* is a sequence of pairs of input and output evaluations; i.e., the space of traces is denoted $\mathsf{Traces} = (\mathsf{Evals}[\mathtt{I}] \times \mathsf{Evals}[\mathtt{O}])^*$. For $I_0, \ldots, I_n \in \mathsf{Evals}[\mathtt{I}]$, $Q_0, \ldots, Q_n \in \mathsf{Evals}[\mathtt{Q}]$, and $O_0, \ldots, O_n \in \mathsf{Evals}[\mathtt{O}]$, let $(I_0, Q_0, O_0), \ldots, (I_n, Q_n, O_n)$ be such that $Q_0 \in \mathsf{Inits}_C$ and for each $0 \leq i < n$, it holds that $(I_i, Q_i) \to_C (Q_{i+1}, O_{i+1})$. Then $(I_0, Q_0, O_0), \ldots, (I_n, Q_n, O_n)$ is a *run of* C and $(I_0, O_0), \ldots, (I_n, O_n)$ is a *trace of* C.

B. Propositional logic

For each vector of propositional variables X, we denote the space of Boolean formulas over X as $\mathsf{Forms}[X]$. We denote a disjoint set of variables corresponding to the variables in X as X'. For all vectors of propositional variables X and Y of equal size and each formula $\varphi \in \mathsf{Forms}[X]$, we denote φ with each variable in X replaced with its corresponding variable in Y as $\varphi[Y/X]$. For each $\varphi \in \mathsf{Forms}[X]$, we denote $\varphi[Y/X]$ alternatively as $\varphi[Y]$. For each $\varphi \in \mathsf{Forms}[X]$ and assignment $\sigma : X \to \mathbb{B}$, we denote the fact that σ *satisfies* φ as $\sigma \vdash \varphi$. For all propositional formulas $\varphi_0, \ldots, \varphi_n, \varphi \in \mathsf{Forms}[X]$, if each satisfying assignment of formulas $\varphi_0, \ldots, \varphi_n$ is also a satisfying assignment of φ, then $\varphi_0, \ldots, \varphi_n$ *entail* φ, denoted $\varphi_0, \ldots, \varphi_n \models \varphi$.

In general, checking if a given propositional formula is satisfiable or checking if a given set of propositional formulas entail a given propositional formula is NP-complete in the combined size of the formulas. However, in practice, many instances of SAT that arise in hardware verification can be solved efficiently using well-studied heuristics [18, 19]. We present SIMAREL as using a generic SAT decision procedure, referred to as ISSAT.

1) Modeling core semantics in logic: For each $C \in \mathsf{Cores}$, the initial states of C (see §III-A) are represented as the propositional formula $\mathsf{IsInit}_C \in \mathsf{Forms}[\mathtt{Q}]$. The transition relation of C, ρ_C, (see §III-A) is represented as a propositional formula $\psi_C \in \mathsf{Forms}[\mathtt{I}, \mathtt{Q}, \mathtt{Q}', \mathtt{O}]$. For each evaluation $\sigma_I \in \mathsf{Evals}[\mathtt{I}]$, each evaluation $\sigma_O \in \mathsf{Evals}[\mathtt{O}]$ of output variables, and all evaluations of state variables $\sigma_Q, \sigma_{Q'} \in \mathsf{Evals}[\mathtt{Q}]$, if $(\sigma_I, \sigma_Q) \to_C (\sigma_{Q'}, \sigma_O)$, then $\sigma_I, \sigma_Q, \sigma_{Q'}, \sigma_O \vdash \psi_C$.

$$\mathsf{Stream} := (\texttt{Lvs} \texttt{ < } \texttt{Lvs})^* (\texttt{IOs has Lvs})^* (\mathsf{Conds} \texttt{ => } \texttt{Lvs})^* \quad (1)$$
$$\mathsf{Conds} := \texttt{I}\ \mathsf{OP}\ \texttt{I} \mid \neg\mathsf{Conds} \mid \mathsf{Conds} \land \mathsf{Conds} \quad (2)$$

Fig. 3: Syntax of STREAMS, defined over spaces of levels Lvs and input wires I, introduced in §III-B1.

2) Interpolation: An interpolant I of mutually-unsatisfiable formulas φ_0 and φ_1 is a formula that explains their mutual unsatisfiability in their common vocabulary.

Definition 1. *For all vocabularies X and Y and formulas $\varphi_0 \in \mathsf{Forms}[X]$ and $\varphi_1 \in \mathsf{Forms}[Y]$ such that $\varphi_0, \varphi_1 \models \mathsf{False}$, an* interpolant *of φ_0 and φ_1 is a formula $I \in \mathsf{Forms}[X \cap Y]$ such that* **(1)** *$\varphi_0 \models I$ and* **(2)** *$I, \varphi_1 \models \mathsf{False}$.*

For all formulas φ_0 and φ_1 that are mutually unsatisfiable, φ_0 and φ_1 have an interpolant. It is unknown whether there is interpolant of size less than exponential in the combined sizes of φ_0 and φ_1. However, in practice, SAT solvers can often generate interpolants of size close to the size of their input formulas [39, 41]. SIMAREL is defined using a procedure ITP that, given two mutually-unsatisfiable SAT formulas φ_0 and φ_1, returns an interpolant of φ_0 and φ_1.

IV. Technical Approach

In this section, we give a language STREAMS for expressing information-flow policies of cores (§IV-A). We then give an algorithm SIMAREL for verifying that a given core satisfies a given STREAMS policy (§IV-B).

A. A policy language for flow security of sequential cores

1) Syntax: The syntax of STREAMS is given in Figure 3 as a grammar in Extended Bachus Normal Form (EBNF). In particular, a policy is **(1)** a sequence of clauses that declare a flows-to relation over levels, **(2)** a sequence of clauses that each binds an I/O wire to a level, and **(3)** a sequence of clauses that each associate a level with an *enabling condition* (Eqn. 1). A condition is a Boolean combination of binary predicates over input wires (Eqn. 2). The space of operations OP contains standard bitwise comparisons.

2) Semantics: The semantics of STREAMS define if a given core satisfies a given STREAMS policy. The level declarations of a STREAMS policy F define a *flows-to* relation over levels. Let the relation $\rightarrow \subseteq \texttt{Lvs} \times \texttt{Lvs}$ be such that for all levels $L_0, L_1 \in \texttt{Lvs}$, if $L_0 < L_1 \in F$, then $L_0 \rightarrow L_1$. For each $w \in \texttt{IOs}$ and $L \in \texttt{Lvs}$ such that w $\texttt{has L} \in F$, w *has level* L in F, denoted $\mathsf{Lv}[F](w) = L$. For each condition $\mathsf{En} \in \mathsf{Conds}$ and level $L \in \texttt{Lvs}$ such that $\mathsf{En} \texttt{ => } L$ is in F, L has enabling condition En in F, denoted alternatively as $\mathsf{En}[F, L]$. For F to be *well-formed*, $\rightarrow$ must be a partial order and the has-level and enabling-condition relations must be functions. For the rest of the paper, we only consider policies that are well-formed.

For the remainder of this section, let $C \in \mathsf{Cores}$, $F \in$ STREAMS, and $L \in \texttt{Lvs}$ be a fixed core, flow policy, and level. Let the set of all input wires with $L' \in \texttt{Lvs}$ in F such that $L' \rightarrow^* L$ be denoted LvIns, and let the set of all output wires with such a level be denoted LvOuts.

For each trace t (§III-A), the subtrace of t visible at L in F is the sequence of all input-output pairs in t in which the input satisfies the enabling condition of L in F.

Definition 2. *For $t, t' \in \mathsf{Traces}$, let t' be the maximal subsequence of t such that for each $i \in \mathsf{Evals}[\texttt{I}]$ and $o \in \mathsf{Evals}[\texttt{O}]$ with $(i, o) \in t'$, it holds that $i \vdash \mathsf{En}[F, L]$. Then t' is the* subtrace of t visible *at L in F.*

Traces t_0 and t_1 are input-equivalent at level L in F if their corresponding inputs are equal at each input wire that flows to L in F.

Definition 3. *Let $t_0, t_1 \in \mathsf{Traces}$ with $n = |t_0| = |t_1|$ be such that for each $0 \leq k < n$, and $i_0, i_1 \in \mathsf{Evals}[\texttt{I}]$ and $o_0, o_1 \in \texttt{O}$ with $(i_0, o_0) = t_0[k]$ and $(i_1, o_1) = t_1[k]$, and each input wire $w \in \mathsf{LvIns}$, it holds that $i_0(w) = i_1(w)$. Then t_0 and t_1 are* input equivalent *at L in F.*

Output equivalence is defined similarly.

C satisfies F at L if all of its traces with visible subtraces at L that are input equivalent at L are output-equivalent at L.

Definition 4. *If for all $t_0, t_1 \in \mathsf{Traces}[C]$ with subtraces t'_0 and t'_1 visible at L in F (Defn. 2) that are input equivalent at L (Defn. 3), t'_0 and t'_1 are output-equivalent at L, then C* satisfies *F at L.*

The fact that C satisfies F at L is denoted $C \vdash_L F$. If for each $L' \in \texttt{Lvs}$, $C \vdash_{L'} F$, then C *satisfies* F. For the remainder of this paper, we will only consider the problem of determining if a given core satisfies a given flow policy at a given level, in order to simplify the presentation.

Our definition of policy satisfaction models an attacker who can directly observe the outputs of a circuit C at each step in which a condition does not satisfy an enabling condition. The attacker succeeds if they can distinguish two sequences of input-equivalent traces using only such observations. Note that the attacker can only observe the output at particular steps, not the time taken by C to generate such outputs.

Our actual implementation of SIMAREL supports a richer syntax with several constructs that are useful for expressing a policy succinctly. In particular, the full language supports clauses that bind all wires to a default level and a default semantics that sets the enabling condition of each level not set explicitly to be True. Conditions can also be defined over inputs received in previous time steps of execution.

B. Verifying policy satisfaction

1) Relational invariants: SIMAREL, given C, F, and L, determines if $C \vdash_L F$. SIMAREL operates on symbolic relations over a vocabulary that models pairs of circuit runs, denoted as run 0 and run 1. The relation is represented as a formula over variables that model the current state of each run in the pair, along with auxiliary variables InEq and OutEq that track if the subtraces of the runs visible at L in F are input-equivalent and output-equivalent, respectively. The space of auxiliary variables is denoted $\mathsf{Eqs} = \mathsf{InEq} \cup \mathsf{OutEq}$. The space of symbolic relations is denoted $\mathsf{SymRels} = \mathsf{Forms}[\mathtt{Wires}_0, \mathtt{Wires}_1, \mathsf{Eqs}]$. The enabling condition of L in F is denoted En.

Indexed relational invariants are a map from pairs of step indices to symbolic relations that soundly model **(1)** the initial condition of C, **(2) & (3)** steps on inputs not visible at L in F, and **(4)** steps in a pair of runs on inputs visible at L in F, for pairs of runs of length up to k. The space of indexed symbolic relations is denoted $\mathsf{IdxRels} = \mathbb{N} \times \mathbb{N} \hookrightarrow \mathsf{SymRels}$.

Definition 5. *Let $I \in \mathsf{IdxRels}$ be such that **(1)***

$$\mathsf{IsInit}_C[\mathtt{Q}_0], \mathsf{IsInit}_C[\mathtt{Q}_1], \mathsf{InEq}, \mathsf{OutEq} \models I(0,0)$$

***(2)** for $i < k-1$ and $j < k$ such that $(i,j), (i+1,j) \in \mathsf{Dom}(I)$ (where $\mathsf{Dom}(I)$ denotes the domain of I),*

$$I(i,j), \psi_C[\mathtt{I}_0, \mathtt{Q}_0, \mathtt{Q}_0'], \neg\mathsf{En}[\mathtt{I}_0] \models I(i+1,j)[\mathtt{Q}_0'/\mathtt{Q}_0]$$

***(3)** for $i < k$ and $j < k-1$ such that $(i,j), (i,j+1) \in \mathsf{Dom}(I)$,*

$$I(i,j), \psi_C[\mathtt{I}_1, \mathtt{Q}_1, \mathtt{Q}_1'], \neg\mathsf{En}[\mathtt{I}_1] \models I(i,j+1)[\mathtt{Q}_1'/\mathtt{Q}_1]$$

***(4)** for $i, j < k-1$ such that $(i,j), (i+1,j+1) \in \mathsf{Dom}(I)$,*

$$I(i,j), \psi_C[\mathtt{I}_0, \mathtt{Q}_0, \mathtt{Q}_0'], \psi_C[\mathtt{I}_1, \mathtt{Q}_1, \mathtt{Q}_1'], \mathsf{En}[\mathtt{I}_0], \mathsf{En}[\mathtt{I}_1], (\mathsf{InEq} \land \mathsf{LvIns}_0 = \mathsf{LvIns}_1 \implies \mathsf{InEq}'), (\mathsf{OutEq} \land \mathsf{LvOuts}_0 = \mathsf{LvOuts}_1 \implies \mathsf{OutEq}') \models I(i+1,j+1)[\mathtt{Q}_0', \mathtt{Q}_1', \mathsf{Eqs}']$$

Then I are indexed relational invariants.

The space of indexed relational invariants is denoted $\mathsf{IdxInvs}$. For $I \in \mathsf{IdxInvs}$, if **(1)** I contain a symbolic relation for the initial index 0 paired with itself and **(2)** for each pair of indices i and j, either $I(i,j)$ entails the relation at a distinct pair of indices or the successor of steps (i,j) in run 0 or run 1 is defined by I, then I are *inductive* indexed invariants.

Definition 6. *Let $I \in \mathsf{IdxInvs}$ be such that **(1)** $(0,0) \in \mathsf{Dom}(I)$ and **(2)** there is some anti-symmetric $C \subseteq (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N})$ such that for each $i, j \in \mathbb{N}$ such that $(i,j) \in \mathsf{Dom}(I)$, either **(a)** there are some $i', j' \in \mathbb{N}$ such that $((i,j),(i',j')) \in C$ and $I(i,j) \models I(i',j')$, **(b)** $(i+1,j) \in \mathsf{Dom}(I)$, or **(c)** $(i,j+1) \in \mathsf{Dom}(I)$. Then I are* inductive *indexed relational invariants.*

Indexed relational invariants I satisfy F at L if they map each pair of indices to a symbolic relation that implies that along all pairs of runs, if inputs are equivalent at L in F, then outputs are equivalent at L in F.

Definition 7. *Let $I \in \mathsf{IdxInvs}$ be such that for all $i, j < k$ such that $(i,j) \in \mathsf{Dom}(I)$,*

$$I(i,j), \mathsf{LvIns}_0 = \mathsf{LvIns}_1 \models \mathsf{LvOuts}_0 = \mathsf{LvOuts}_1$$

Then I satisfies *F at L.*

For $I \in \mathsf{IdxInvs}$, the fact that I satisfies F and L is denoted $I \vdash_L F$.

Inductive relational invariants of C that satisfy F at L are evidence that C satisfies F at L (see §A, Lemma 4). As a result, SIMAREL proves or disprove that C satisfies F at L by searching for inductive indexed relational invariants that satisfy F at L.

2) Verification algorithm: Alg. 1 contains pseudocode for SIMAREL, which given $C \in \mathsf{Cores}$, $F \in \mathrm{STREAMS}$, and $L \in \mathsf{Lvs}$, determines if C satisfies F at L. SIMAREL defines a recursive procedure SIMREC that takes a natural number k and returns either (1) False to denote that C does not satisfy F at L for some pair of runs of length no less than k or (2) True to denote that C satisfies F at L (line 2—line 9). SIMAREL calls SIMREC on 0 and returns the result (line 10).

```
Input  : $C \in$ Cores, $F \in$ STREAMS, $L \in$ Lvs
Output : Decision as to whether $C \vdash_L F$.
1  Procedure SIMAREL ($C$, $F$, $L$)
2      Procedure SIMREC($k$)
3          switch CHK($C, F, L, k$) do
4              case Unsafe: return False ;
5              case $I \in$ IdxRels:
6                  if HASIND($I$) then return True ;
7                  else return SIMREC($k + 1$) ;
8              end
9          endsw
10     return SIMREC($0$) ;
```

Algorithm 1: SIMAREL: given $C \in \mathsf{Cores}$, $F \in \mathrm{STREAMS}$, and $L \in \mathsf{Lvs}$, determines if $C \vdash_L F$.

SIMREC, given input k, runs a procedure CHK on C, F, L, and k which returns either Unsafe to denote that C does not satisfy F at L on all pair of runs of length k or indexed relational invariants up to k that satisfy F (line 3). The design of CHK is given in §IV-B3. If CHK returns Unsafe, then SIMREC returns that C does not satisfy F at L (line 4).

Otherwise, if CHK returns indexed relational invariants I that satisfy F at L, then SIMREC runs procedure HASIND on I. If HASIND returns that I contain inductive indexed relational invariants, then SIMREC returns that C satisfies F at L (line 6; the implementation of HASIND is given in §IV-B4). Otherwise, SIMREC recurses on $k+1$ and returns the result (line 7).

3) Finding indexed relational invariants up to a bound: CHK, given $k \in \mathbb{N}$, attempts to construct indexed relational invariants I from the results of a series of interpolation queries (§III-B, Defn. 1), defined as follows. For each $n \in \{0,1\}$ and $j < k$, let $\mathtt{I}_j^n$ model the values read by run i in step j, and let $\mathtt{Q}_j^n$ model the state of run n after taking step j.

For all $i, j < k$, the symbolic relation in I at i and j is constructed from an interpolant of two formulas: **(1)** the *pre-formula* at (i,j), denoted $\varphi_{i,j}^-$, and **(2)** the *post-formula* at (i,j), denoted $\varphi_{i,j}^+$. Each pair of i steps of run 0 and j steps of run 1 corresponds to a model of $\varphi_{i,j}^-$, defined casewise on i and j as follows. $\varphi_{0,0}^-$ is the initial condition of C instantiated on the state variables of **(1)** run 0 and **(2)** run 1, combined with **(3)** the fact that initially, the (empty) visible suffixes of all pairs of runs after 0 steps are input-equivalent and output-equivalent in F at L. I.e., $\varphi_{0,0}^-$ is

$$(\mathbf{1})\mathsf{IsInit}_C[\mathtt{Q}_0^0] \land (\mathbf{2})\mathsf{IsInit}_C[\mathtt{Q}_0^1] \land (\mathbf{3})\mathsf{InEq}_{0,0} \land \mathsf{OutEq}_{0,0}$$

For pairs of indices $(i,j) \neq (0,0) \in \mathbb{N} \times \mathbb{N}$, $\varphi_{i,j}^-$ is defined as follows. For each $0 \leq i, j < k-1$, $\mathsf{InvisStep}_{i,j}^0$ relate **(1)** the states of run 0 after i steps and run 1 after j steps, combined with **(2)** the semantics of run 0 taking a step, on **(3)** inputs invisible at L to the resulting states after run 0 takes $i+1$

steps and run 1 takes j steps. I.e., $\mathsf{InvisStep}^0_{i,j}$ is

$$(\mathbf{1}) I(i, j+1) \land (\mathbf{2}) \psi_C[\mathtt{Q}^0_i, \mathtt{I}^0_{i+1}, \mathtt{Q}^0_{i+1}] \land$$
$$(\mathbf{3}) \neg \mathsf{En}[\mathtt{I}^0_{i+1}] \land \mathsf{Eqs}_{i+1,j+1} = \mathsf{Eqs}_{i,j+1}$$

$\mathsf{InvisStep}^1_{i,j}$ is defined symmetrically. For $0 \leq i < k-1$, $\varphi^-_{i+1,0}$ is $\mathsf{InvisStep}^0_{i,0}$. For $0 \leq j < k-1$, $\varphi^-_{0,j+1}$ is $\mathsf{InvisStep}^1_{0,j}$.

$\mathsf{VisSteps}_{i,j}$ relates **(1)** the states after i steps of run 0 and j steps of run 1, combined with the semantics of **(2)** run 0 and **(3)** run 1 taking a step on **(4)** inputs visible at in F at L to states after run 0 takes $i+1$ steps and run 1 takes $j+1$ steps. I.e., $\mathsf{VisSteps}_{i,j}$ is

$$(\mathbf{1}) I(i,j) \land (\mathbf{2}) \psi_C[\mathtt{Q}^0_i, \mathtt{I}^0_{i+1}, \mathtt{Q}^0_{i+1}] \land (\mathbf{3}) \psi_C[\mathtt{Q}^1_j, \mathtt{I}^1_{j+1}, \mathtt{Q}^1_{j+1}] \land$$
$$(\mathbf{4}) \mathsf{En}[\mathtt{I}^0_{i+1}] \land \mathsf{En}[\mathtt{I}^1_{j+1}] \land$$
$$(\mathsf{InEq}_{i,j} \land \mathsf{LvIns}^0_{i+1} = \mathsf{LvIns}^1_{j+1} \implies \mathsf{InEq}_{i+1,j+1}) \land$$
$$(\mathsf{OutEq}_{i,j} \land \mathsf{LvOuts}^0_{i+1,j+1} = \mathsf{LvOuts}^1_{i+1,j+1} \implies \mathsf{OutEq}_{i+1,j+1})$$

For $0 \leq i, j < k-1$, $\varphi^-_{i+1,j+1}$ is

$$\mathsf{InvisStep}^0_{i,j+1} \lor \mathsf{InvisStep}^1_{i+1,j} \lor \mathsf{VisSteps}_{i,j}$$

Each suffix of run 0 after step i and run 1 after step j of runs that do not satisfy F at L corresponds to a model of the post-formula $\varphi^+_{i,j}$, defined as follows. Each suffix of run n from step j to step k corresponds to a model of the formula Rest^n_j:

$$\bigwedge_{j \leq j' < k-1} \psi_C[\mathtt{Q}^n_{j'}, \mathtt{I}^n_{j'}, \mathtt{Q}^n_{j'+1}]$$

Each pair of suffixes that correctly updates Eqs corresponds to a model of $\mathsf{UpdEqs}_{i,j}$, defined as follows. $\mathsf{UpdInInvis}^0_{i,j}$ constrains that if at steps i and j (of runs 0 and 1), runs 0 and 1 have read input streams equivalent at L and run 0 next reads an input invisible at L, then at $i+1$ and j, the runs have read input streams equivalent at L:

$$\mathsf{InEq}_{i,j} \land \neg \mathsf{En}[\mathtt{I}^0_{i+1}] \implies \mathsf{InEq}_{i+1,j}$$

$\mathsf{UpdInInvis}^1_{i,j}$ constrains $\mathsf{InEq}_{i,j}$ and $\mathsf{InEq}_{i,j+1}$ symmetrically to model steps of run 1.

$\mathsf{UpdInVis}_{i,j}$ constraints that if runs 0 and 1 both next read inputs visible at L, then at $i+1$ and $j+1$, the runs have read input streams equivalent at L. I.e., $\mathsf{UpdInVis}_{i,j}$ is

$$\mathsf{InEq}_{i,j} \land \mathsf{En}[\mathtt{I}^0_{i+1}] \land \mathsf{En}[\mathtt{I}^1_{j+1}] \land \mathsf{LvIns}^0_{i+1} = \mathsf{LvIns}^1_{j+1} \implies \mathsf{InEq}_{i+1,j+1}$$

$\mathsf{UpdOutInvis}^n_{i,j}$ and $\mathsf{UpdOutVis}_{i,j}$ symmetrically constrain $\mathsf{OutEq}_{i,j}$. $\mathsf{UpdEqs}_{i,j}$ is

$$\bigwedge_{\substack{i<i'<n-1\\ j<j'<n}} \mathsf{UpdInInvis}^0_{i,j} \land \mathsf{UpdOutInvis}^0_{i,j} \land$$
$$\bigwedge_{\substack{i<i'<n\\ j<j'<n-1}} \mathsf{UpdInInvis}^1_{i,j} \land \mathsf{UpdOutInvis}^1_{i,j} \land$$
$$\bigwedge_{\substack{i<i'<n-1\\ j<j'<n-1}} \mathsf{UpdInVis}_{i,j} \land \mathsf{UpdOutVis}_{i,j}$$

Each pair of suffixes that satisfy F at L after each step corresponds to a model of $\mathsf{PolSat}_{i,j}$, defined as

$$\bigwedge_{i<i'<n, j<j'<n} \mathsf{InEq}_{i',j'} \implies \mathsf{OutEq}_{i',j'}$$

The complete post-formula $\varphi^+_{i,j}$ is:

$$\mathsf{Rest}^0_i \land \mathsf{Rest}^1_j \land \mathsf{UpdEqs}_{i,j} \land \neg \mathsf{PolSat}_{i,j}$$

```
Input  : $I \in \mathsf{IdxInvs}$
Output : Decision as to whether $I$ contains inductive
         indexed relational invariants.
Procedure HasInd($I$)
    Procedure HasIndAux($O, D$)
        if $O = \emptyset$ then return True ;
        $((i,j), O') :=$ Choose($O$) ;
        if $(i,j) \notin \mathsf{Dom}(I)$ then return False ;
        $D' := D \cup \{(i,j)\}$ ;
        if $\bigvee \{ I(i,j) \models I(i',j') \mid (i',j') \in D \}$ then
            return HasIndAux($O', D'$)
        end
        $\iota_0 :=$ HasIndAux($O' \cup \{(i+1,j)\}, D'$) ;
        $\iota_1 :=$ HasIndAux($O' \cup \{(i,j+1)\}, D'$) ;
        return $\iota_0 \lor \iota_1$ ;
    return HasIndAux($\{(0,0)\}, \emptyset$) ;
```

Algorithm 2: HasInd: given $I \in \mathsf{IdxInvs}$, returns whether or I contains inductive indexed relational invariants.

Chk first runs IsSat on $\varphi^-_{0,0} \land \varphi^+_{0,0}$, and if IsSat returns that the formula is satisfiable, returns Unsafe. Otherwise, Chk iteratively computes relational invariants for each $0 \leq i, j < n$ in any topological ordering of the space of pairs in $\mathbb{Z}_k \times \mathbb{Z}_k$. In each iteration, Chk sets $I(i,j)$ to be $\textsc{Itp}(\varphi^-_{i,j}, \varphi^+_{i,j})[\mathtt{Q}_0, \mathtt{Q}_1]$.

The correctness of SimaRel is supported by the fact that, given $k \in \mathbb{N}$, if Chk returns Unsafe, then C does not satisfy F at L (§B, Lemma 5), and otherwise returns indexed relations that prove that all pairs of runs of C up to length k satisfy F at L (§B, Lemma 6).

4) Finding inductive indexed relational invariants: HasInd (Alg. 2), given $I \in \mathsf{IdxRels}$, returns whether or not I contains inductive indexed relational invariants. HasInd contains a procedure HasIndAux (Alg. 2, line 2—line 12) that, given *obligation* and *discharged* pairs of indices $O, D \subseteq \mathbb{N} \times \mathbb{N}$, returns whether I contains inductive indexed relational invariants that must contain O and may contain any elements in D. HasInd runs HasIndAux on obligations consisting of only 0 paired with itself and no discharged pairs, and returns the result (line 13).

HasIndAux, checks if O is empty, and if so, returns True (line 3). Otherwise, if O is not empty, then HasIndAux removes a pair of indices (i,j) from O to generate obligations O' (line 4), and checks if (i,j) have invariants in I; if not, then HasIndAux returns False (line 5).

Otherwise, if (i,j) have invariants in I, then HasIndAux extends D with (i,j) to generate discharged pairs D' (line 6), and checks if there are indices (i',j') such that $I(i,j)$ entails $I(i',j')$ (line 7); if so, then HasIndAux recurses on O' and D', and returns the result (line 8).

Otherwise, if there are no such indices i' and j', HasIndAux recurses on O' extended with index pair $(i'+1, j)$ and D' (line 10), recurses on O' extended with index pair $(i, j+1)$ and D' (line 11), and returns True if either recursive call returns True (line 12).

HasInd soundly determines if given indexed relational invariants contain inductive indexed relational invariants (§A, Lemma 7).

5) Synthesizing policy violations: To simplify the presentation of SIMAREL, we have presented it as an algorithm that takes a given core C, flow policy F, and level L and returns only a decision as to whether C satisfies F at L. Our actual implementation of SIMAREL, when given a core C that does not satisfy F at level L, returns a counterexample that witnesses non-satisfaction, represented as a pair of runs of C that are input-equivalent at L, but not output-equivalent at L. In particular, if CHK runs ISSAT on $\varphi^{-}_{0,0} \land \varphi^{+}_{0,0}$ and ISSAT determines that the formula has a model m, then CHK returns the pair of runs of C that correspond to m.

C. Discussion

SIMAREL is a sound flow verifier.

Theorem 1. *If* $\text{SIMAREL}(C, F, L) = \mathsf{True}$, *then* $C \vdash_L F$.

For a proof of Thm. 1, see §D.

SIMAREL is also a complete flow verifier.

Theorem 2. *If* $\text{SIMAREL}(C, F, L) = \mathsf{False}$, *then* $C \nvdash_L F$.

In principle, SIMAREL will eventually terminate on any given core C and flow policy F. Termination follows from the observation that each circuit is a finite-state machine. Thus if SIMAREL considers a sufficiently large number n of steps, it will either find a pair of runs of C that do not satisfy F, or it will synthesize invariants that imply that each pair of runs of length less than n must reach some state in a cycle. Such a cycle manifests in SIMAREL as inductive indexed relational invariants. The maximum number of steps that SIMAREL may need to consider is bounded by the number of pairs of states in a given circuit (i.e., by $|2^{2|\mathsf{Q}|}|$). In practice, when SIMAREL successfully verifies flow security or finds a policy violation, it typically finds a proof of pairs of runs up to a bound that it significantly lower than the maximum number of steps required to verify a core.

One approach for verifying that a program P satisfies a property F over a bounded number of runs is to verify that the *self-composition* of P, denoted P^2 satisfies a safety property F^2 derived from F [7, 59]. In particular, **(1)** P^2 reads an initial state and stores it in variables $\mathtt{vars}_0$. **(2)** P^2 runs P on the initial state and stores the final result in variables $\mathtt{vars}'_0$. **(3)** P^2 reads a second initial state and stores it in variables $\mathtt{vars}_1$. **(4)** P^2 runs P on the second initial state and stores the result in variables $\mathtt{vars}'_1$. A safety verifier is then run to determine if P^2 satisfies $F^2 \equiv \varphi_F[\mathtt{vars}_0, \mathtt{vars}_1] \implies \mathtt{vars}'_0 = \mathtt{vars}'_1$.

Self-composition can potentially verify the flow security of programs that take a bounded vector of inputs, such as a finite tuple of bounded integers. However, applying the approach to verify flows that operate over unbounded streams of inputs would be non-trivial: a key operation of the self-composition is to sequentially store two complete copies of its input and test them under a relational predicate only after executing a second run of the program.

The inability of self-composition to verify flow-security of programs that operate over unbounded inputs implies that it cannot be directly adapted to verify non-trivial flow security properties of sequential hardware circuits, all of which operate on unbounded streams of inputs. Moreover, we found that the technique could not be directly applied to verify the flow security of any of the cores and flow policies that we collected from sources of actual cores (§V), including the example core and policy introduced in §II.

V. EVALUATION

We performed an empirical evaluation of our approach to answer the following research questions. **(1)** Can the information-flow security requirements of practical, security-critical cores be expressed in STREAMS? **(2)** If practical, security-critical cores and their flow policies are given to SIMAREL, can SIMAREL efficiently either verify that the cores satisfy their policies or generate inputs on which they do not?

In summary, our results answer the above questions positively. We used STREAMS to express the flow requirements of 12 open-source cores drawn from different application domains and hosted on the open-source repositories OpenCores [47] and AxBench [1], and used SIMAREL to attempt to either verify the flow security of the cores or find flow vulnerabilities. SIMAREL verified flow security of six modules and found flow vulnerabilities in six other modules.

The sources of the found vulnerabilities are either in the implementation of particular modules from individual sources (in five cases) or in logic that integrates modules from multiple sources (in one case). Such sources of vulnerabilities are critical concerns in the design of FPGA's, which typically rely on reusing cores from different sources to implement complex system-level functionality, or which attempt to optimize a core after integrating multiple cores. These results not only shows the efficacy of our approach but also the necessity of our solution at a time when FPGAs are being deployed and integrated in both data centers and embedded devices. We also implemented patches that address the vulnerabilities found using our technique and have submitted the patches to OpenCores and AxBench.

A. Methodology

We implemented SIMAREL by extending the ABC hardware model checker [10] to solve interpolation queries and check for inductive proofs, as described in §IV-B. The input to ABC is a low-level logic representation of the core in Berkeley Logic Interchange Format (BLIF) [9]. We used the Yosys open source synthesis tool [64] to generate the BLIF file from the high-level Verilog description of the cores.

We collected a set of open-source cores from OpenCores and AxBench. the features of the benchmarks are summarized in Table I. OpenCores is a large collection of open-source hardware cores; a wide verity of customers already deploy the cores that it hosts [48]. AxBench is another open source repository of cores developed in hardware-design research [1].

Benchmark Features					Policy Features		Performance		
Name	**Domain**	**Origin**	**LoC (Verilog)**	**KLoC (BLIF)**	**Levels**	**Clauses**	**Secure**	**Time (s)**	**Mem. (MB)**
Wishbone flash cntrl.	Embedded Comp.	opencore	129	7	2	1	No	0.18	12
SD card controller	Storage mgmt.		4,080	36	2	11		26.05	652
UART	Debugging		253	6	2	1		0.16	23
Ethernet controller	Comm. controller		205	2	2	1		1.10	29
AntiLog2	DSP		110	3	2	1		0.69	27
F-kinematics	Robotics	axbench	18,282	1,755	2	4		44.15	1,010
Wishbone flash cntrl. (patch)	Embedded Comp.	opencore	130	7	2	1	Yes	0.26	15
SD card controller (patch)	Storage mgmt.		4,147	36	2	11		39.85	650
UART (patch)	Debugging		267	927	2	1		6.27	310
Ethernet controller (patch)	Comm. controller		223	2	2	1		1.19	29
AntiLog2 (patch)	DSP		122	3	2	1		2.29	30
F-kinematics (patch)	Robotics	axbench	18,426	1,823	2	4	-	-	-
Reed-Solomon	Error correction	opencore	4,054	119	2	1	Yes	254.44	765
RR arbiter	H/W controller		268	0.3	2	1		0.11	16
Gaussian noise gen.	DSP		1,064	26	2	1		2.78	92
Interrupt controller	System controller		248	2	2	1		1.16	57
FIR-filter	Communication	axbench	101	191	2	1		1.77	66
Sobel filter	Image processing		386	404	2	1		3.39	146

TABLE I: The results of our evaluation. Under heading "Benchmark Features,' ' column "Name" contains the name of the benchmark; column "Origin" contains the benchmark's origin of development; column "LoC" contains the number of lines of Verilog code and BLIF code for the benchmark core's design. Under the heading "Policy Features", column "Levels" contains the number of levels used in the policy; "Clauses" contains the number of allows and prohibits. Under the heading "Performance", column "Time" contains the time SIMAREL consumed for each benchmark; column "Mem." contains the peak amount of memory that SIMAREL used.

The benchmarks we collected implement complete hardware cores and follow industry standards. Thus, they are suitable for integration with other cores inside an FPGA for industrial purposes. To measure the complexity of each benchmark, we counted the number of lines of code (LoC) of both the original Verilog file and synthesized BLIF file. Because a core's BLIF file represents the actual physical structure of hardware synthesized, the size of the BLIF file is a more consistent metric of complexity than Verilog, which describes the core at comparatively high level.

For each benchmark core, we wrote an information-flow policy in STREAMS that describes on which inputs the core may release a information about its sensitive inputs, based on the core's documentation. We wrote policies using an extension of STREAMS as described in §IV-A that additionally supports references to values read in a time step that is some constant offset before the current step. We then checked that each core satisfies its corresponding policy by running SIMAREL on the target core and policy. For each benchmark, we ran SIMAREL on a machine running as its operating system Linux Ubuntu with kernel 3.16.0-38-generic. The machine used for benchmarks contains an Intel Core i7 4720HQ processor that runs at 2.4 GHz and contains 8 GB of memory.

B. Results

In this section, we describe insecure benchmark cores that STREAMS specified, the STREAMS policy that we wrote for the cores, and the results of running SIMAREL on the cores and their policies. We omit the SD card controller example because we discussed it in §II.

1) Insecure benchmarks:

a) Wishbone flash memory controller: The Wishbone flash memory controller implements a critical subsystem that enables an FPGA to access the main storage of an embedded platform, which can be, for example, an IoT device. The inputs to the Wishbone controller are read or write commands and addresses generated by other modules in the FPGA. The output is the control signals that let the flash storage perform the FPGA's read or write commands. The core implements a finite state machine with 16 states that generates the control signals.

We wrote a STREAMS policy that only allows information to flow from the input data port to flash when the controller receives a write-enable signal. If the controller does not satisfy this policy, several attacks are feasible. In particular, a malicious circuit processing sensitive information on the FPGA can use the Wishbone flash controller to leak the information to the flash storage. Also, when the FPGA implements a processor, software executing on the processor that polls the write data can gain the complete information about the data written.

SIMAREL found that the controller does not satisfy such a policy in less than a second. The results confirm that each of the attacks given above is indeed feasible.

b) Universal asynchronous receiver/transmitter: A *Universal Asynchronous Receiver Transmitter* (UART) is a widely used serial communication mechanism [46]. UART connects systems with different clock domains using a shift register, which prevents unexpected data loss from clock timing synchronization failure during data exchange. UART prepares data for transmission or retrieval using units that serialize and deserialize messages. Some UART modules, including the module that we analyzed, provide an interface for collecting debugging information from systems. Given that debuggers typically are granted near-complete access to a system's internal data, it is critical that the module satisfies expected requirements for information flow. In a network router design, for instance, the debug interface may leak the routing table entries that decide where the router sends packets. Thus, attackers may track the paths of packets and identify the senders and receivers.

We wrote a STREAMS policy that specifies that a given UART module must not reveal read data except on intended clock cycles. If the module violates the policy, then an attacker can observe internal data in a system, even after the system has reverted to perform normal operations. SIMAREL revealed this vulnerability using our STREAMS. We found that the leak is due to incomplete case-handling logic, similar to the vulnerable code for the SD card controller discussed in §II.

c) Ethernet controller: Modern FPGAs typically include a physical Ethernet port. However, they usually do not include a built-in Ethernet controller. Thus, FPGA cores that require Ethernet functionality often use a third-party Ethernet controller developed independently.

We analyzed an Ethernet-controller core with several submodules. In particular, the receive (RX) module transfers incoming data from outside of the system to the host. Because such a module is typically shared by many users, it should only provide information read in the most recent step to the system agent issuing the current request. If the core does not satisfy such a policy, then an attacker could learn information read from the network in the past. That is, adversaries can access data previously sent to other users.

We expressed the above information-flow policy as a STREAMS policy and ran SIMAREL to determine if the RX module satisfies the given policy. SIMAREL determined that the module does not satisfy the policy, which indicates that the RX module leaks the incoming data from the network to the future time slots.

d) Forward kinematics for a robot arm: The *forward kinematics* core takes the angles of the joints of a robot arm and computes the coordinates of the end-point of the arm by solving kinematics equations. In particular, the core implements trigonometric functions and arithmetic operations. The trigonometric functions are implemented as hard-coded lookup tables. The arithmetic operations are implemented using fast, optimized implementations developed in Aoki Laboratories [31].

The core is pipelined into two stages, which means that it generates an output for the input from the two clock cycles before the current cycle. Hence, we wrote a policy that specifies that only the input from the two clock cycles before the current cycle may flow to the output. SIMAREL found inputs on which the core does not satisfy the given policy. With further investigations, we observed that the core's output is not initialized immediately and generates a large value under a reset signal. As a result, an attacker can jerk the robot arm by continuously commanding the module to reset since output value directly affects the movement of the robot arm.

We identified that logic containing the vulnerability is in a multiplier module deeply nested in the core. We compared the multiplier in the core to the original implementation of the multiplier provided by Aoki Laboratories [31]. From the comparison, we found that the designer of the forward kinematics core made such an unintended vulnerability during the optimization by pipelining. However, the optimized multiplier leaks partial information about its arguments if they satisfy a particular arithmetic constraint. Our experience analyzing the forward kinematics module indicates that the integration and optimization of even mature modules can result in unexpected information leaks.

e) Antilogarithm operation: Computing the *antilogarithm* (antilog) of a number is one of the most frequent operations in scientific computing and multimedia applications [49]. If a module that implements such a frequent operation leaks any information, it will compromise the security of many cores that include the module. In addition, because operands in such an operation can be sensitive numeric values in many settings, a malicious anti algorithm core can be one of the major source of information leak in hardware design with the core.

We analyzed an open-source module that computes antilogarithms in two cycles. We wrote a STREAMS policy that only allows the information flow from input data received two cycles before the current cycle to the output. Otherwise, an attacker can observe partial information about the history of the values passed to this module. That is, an attacker could infer information about previous values provided to the operator or infer how frequently operator has been invoked.

We ran SIMAREL on the core and discovered it leaks information about previous input values after computing and outputting the desired result. We identified that the module does not support reset and as a result, an attacker can easily observe information about previous inputs. While resetting the state of a module after each computation is a natural operation, it is not supported by the module and is not upheld by default operation. As a result, an attacker can easily observe information about inputs given in previous invocations.

2) Secure benchmarks: We now describe the core designs and policies on which we evaluated SIMAREL.

a) Round-robin arbiter: An arbiter manages multiple requests to access a hardware resource such as a bus, buffer, or I/O port. The arbiter ensures that only one requester at a time gains access to the shared resource. The round-robin arbiter ensures fairness by granting access based on a round-robin schedule. The router has dedicated grant signal wires for each requester. When it decides a winner, than the grant signal wire for the winner becomes active.

When no agent requests to access a resource, the arbiter should not reveal any information about the agents that were granted access to the resource in previous rounds. If the arbiter does not satisfy such a policy, then an agent that should not have access to any information about the resource, and in particular cannot request to access it, can still learn information about the resource, in particular how often other agents have accessed it. E.g., an attacker could count the number of requests made to access a motion sensor and estimate how many people visited a location under surveillance.

We wrote a STREAMS policy that formalizes the above policy. SIMAREL verified that the arbiter module satisfies the policy in less than a second.

b) Gaussian noise generator for random number generation: Generating a large number of normally distributed random samples is a crucial process in molecular dynamics simulation or financial modeling [32]. A Gaussian noise generator implemented in hardware can improve the efficiency and performance of these modeling and simulation applications. We analyzed a core that takes a one-bit request signal and outputs random number.

We wrote a STREAMS policy that checks if the core enumerates the random numbers independent of the number of request signal. The request signal generates an output signal that indicates the validity of current output. Such a valid signal is common in hardware design for communication protocols between modules, especially when generating requested data takes longer than one cycle. Without the valid signal, the requester does not know if the incoming data is the requested data or intermediate data. Without the independence property, the random number generator contains an information leak. That is, an attacker can count the number of requests and infer information about the application. SIMAREL verified that the core satisfies its policy.

c) Reed-Solomon decoder for bitwise error correction: A Reed-Solomon code is an error correcting code (ECC), which contains redundant data to correct possible errors during data communications [53]. This error correction system has been an industry standard for a long time and is used for CDs and DVDs [11], RAID systems [50], and other communication protocols [33].

We analyzed a core that implements a Reed-Solomon decoder as provided on OpenCores. This core takes encoded data as well as several control and enable signals, and generates decoded data. If the core does not receive an enable signal, it should not leak any information from its input ports to the output port. If the core does not satisfy such a policy, an attacker could use the module as a passthrough channel to leak privileged information when the enable signal is not set.

We wrote a STREAMS policy that expresses the above policy. SIMAREL successfully verified that the module satisfied the given STREAMS policy.

d) Programmable interrupt controller: Interrupts are fundamental communication mechanism in modern computer systems. Programmable interrupt controllers manage events generated by different units. Critical I/O devices such as storage and networking interfaces usually depend on such controllers to communicate with a host system.

We analyzed an open-source core that implements an interrupt controller that can service up to eight hardware components. The interrupt request from hardware components at the previous cycle should only affect whether the core sends an interrupt signal to the CPU depending on the enable signal. Simultaneously, the CPU's interrupt mask should be clear, and there should not be previously triggered pending interrupts. If the core does not satisfy such a policy, then an attacker could stall the processor by continuously injecting void interrupt signals. Moreover, because the CPU prioritizes interrupts over normal execution, such a vulnerability could potentially compromise the security of the entire system.

We wrote a STREAMS policy that formalizes the above policy. SIMAREL verified that the module satisfies the above policy.

e) Edge detection in images using Sobel filter: The Sobel filter takes a greyscale image as input and identifies pixels depicting edges in the image. The Sobel filter slides a three-by-three window over the image, and estimates the gradient of the center with respect to the neighboring pixels. In other words, the algorithm performs a two-dimensional convolution over the image to calculate the gradient. If the gradient is greater than a threshold, the filter identifies the pixel as part of an edge.

The implementation of a Sobel filter that we analyzed is a prime target for attacks that might corrupt the memory (e.g., out-of-bound accesses) because it directly accesses its host's physical memory without mediation from the operating system. Such a design is common in the FPGA design since the ability to perform direct accesses is critical for the efficiency of the module.

We wrote a STREAMS policy that specifies that information from the accessible region of memory should only flow to the specified output region. Such design is common in the FPGA design since the ability to perform direct accesses is critical for the efficiency of the module. Users can configure the prohibited region in the input image of the Sobel filter implementation.

We wrote a STREAMS policy specifying that information from the accessible region should only flow to the output. For instance, some region inside the input image might belong to kernel memory segment if a wrong image pointer is applied or an attacker intentionally places a pointer near the memory border. In such cases, the filter should exclude such region during its operation. We ran SIMAREL on the design, and SIMAREL proved that the design satisfies the policy.

f) Digital filter for radio communication: A *Finite Impulse Response* (FIR) filter is one of the primary types of filters used in *Digital Signal Processing* (DSP). FIR filters are commonly used for communicating with Bluetooth devices. We analyzed a design for a FIR filter available in the AxBench repository [1]. The FIR filter takes an eight-bit signal as input and chooses a frequency for radio communication based on the signal. The filter samples the input signal over four consecutive clock cycles. The filter then calculates a linear combination of the four samples using a sequence of addition and multiplication operations.

We wrote a STREAMS policy that specifies that the FIR module may only sample an input signal for at most four clock cycles. Alternatively, the attacker can cause an endpoint to send information on a chosen frequency that is under their control. We ran SIMAREL on the design of the FIR filter using the above policy, and it proved that the filter indeed satisfies the policy.

3) Patches for insecure benchmarks: We have implemented patches for benchmarks that SIMAREL determined were insecure. SIMAREL proved that our patches for five insecure benchmarks satisfy the STREAMS policy we wrote. SIMAREL took slightly more time on the patches to verify the security under given STREAMS policies than the original vulnerable circuits. This is natural since patches do not involve any policy

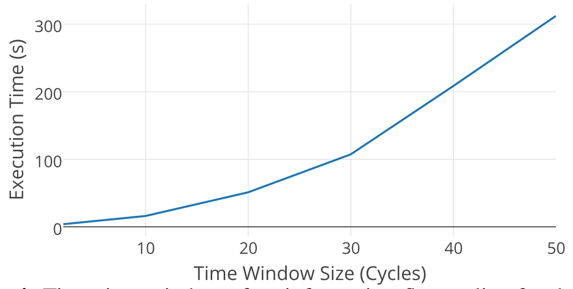

Fig. 4: Time time window of an information-flow policy for the Sobel filter versus the time taken by SIMAREL to verify flow-security of the filter.

violations that interrupts the model checker. Thus the model checker runs until it finishes proving satisfaction.

One of the patches for forward kinematics could not be verified within one hour. It generates an extraordinarily large set of variables and clauses even in the first step of model checking. In particular, it generated 45,882 variables and 169,648 clauses at the first step, about four times as large as the second largest core with 14,063 variables and 47,721 clauses. In addition, the multiplier has 1,469 module definitions and there are even more number of instances of those modules. Such complexity dramatically increases the complexity as the size of generated BLIF file for forward kinematics core indicates. Due to this complexity, the model checker will require tremendous time to give an answer if the model checker does not find any bug or proof within a few steps.

C. *Execution time as a function of policy features*

The performance of SIMAREL does not appear to depend directly on the size of a given core design or policy. E.g., the patched SD card controller and Reed-Solomon decoder have similar sizes, but SIMAREL uses significantly more time to verify the decoder.

Based on our experience using SIMAREL, we suspect that a key factor in determining its performance is the number of cycles across which the policy relates input or output values, which we refer to as the policy's *time window*. To explore the relationship between a policy's time window and the performance of SIMAREL, we wrote a tunable policy for the Sobel filter that compares values across a parameterized number of steps of its execution. We varied the size of time window in the policy and measured the resulting verification time of SIMAREL.

Figure 4 shows the results of the evaluation. The execution time increases along a super-linear progression with the time window of the given policy.

VI. Related work

a) Checking information flow in hardware: Several hardware description languages and hardware designs have been proposed for tracking the flow of information throughout a hardware system [35, 60, 61]. The strength of such approaches is that they can track the taint of information through large hardware cores. However, such approaches are not an ideal solution for checking information flow through FPGA circuits for two key reasons. First, space on an FPGA is a precious resource, and thus any approach that induces area overhead is not desirable. Second, most practical FPGA cores must, under particular conditions, release predicates about their sensitive information under particular conditions. Our approach to checking flow in FPGA cores was designed to directly address these constraints. SIMAREL is completely static, thus inducing no runtime overhead, and reasons about conditional release policies accurately using an automatic theorem prover.

Other hardware description languages, such as Caisson, extend traditional hardware description languages with mechanisms for describing state machines [36]. Similar to STREAMS, Caisson can be used to express some information flow policies involving timed release of information. The key distinction of our approach from Caisson is that our approach can be used to verify critical flow properties of cores written in a conventional hardware design language and integrated from multiple sources, accompanied by a relatively small flow policy.

A hardware design can be written in SecVerilog [68] and type-checked to prove that it conditionally releases sensitive information. A programmer can only prove flow security by labeling all registers in the design that may store such information with dependent labels. In our approach, a programmer only needs to declare expected conditional flows from input to output. Then, SIMAREL automatically infers suitable relational invariants over all internal state. For example, the STREAMS policy for the Ethernet controller contains only one label ordering and one clause clause. However, the same policy could be enforced in SecVerilog would require three dependent labels on three registers, in addition to 22 labels on other wires and registers. An interesting direction for future work could be to automatically extract valid dependently-typed designs from the relational invariants synthesized by SIMAREL.

Star-logic [60] is a combination of static and dynamic information flow checking used during software-hardware co-design. It automatically generates a GLIFT [61] dynamic information tracking circuit based on a security lattice given by a programmer. Star-logic tests the information flow of a target system considering target software. That is, it tests all the possible execution cases of the target software. Star-logic is designed, and well-suited, to verifiably enforce strict non-interference. However, its label-based system cannot be applied to verify conditional release of sensitive information.

Hardware description languages and analyses have been proposed for detecting and mitigating timing attacks in hardware [16, 30, 67, 68]. Previous work has also proposed attacks and analyses for determining if an FPGA core leaks sensitive information through the power that it uses [57, 58]. These approaches primarily address flow-checking problems that are orthogonal to the one that we consider in this work. However, while it is feasible that approaches developed originally to mitigate timing channels could be adapted to reason about partial release, the language-based approaches developed in previous work require extensive annotations from core developers. In contrast, SIMAREL is fully automatic.

b) Verifying conditional information release: A *taint-tracking* analysis takes a labeling of the system's inputs as tainted or untainted and determines how tainted information explicitly flows through program memory over an execution. Systems have been proposed that track the flow of tainted values at runtime [20, 56] or statically [23]. Taint-tracking analyses are well-suited for analyzing programs, which often use distinct regions of memory operated on by distinct program regions to handle sensitive or insensitive data. Similarly, such approaches are likely well-suited to analyze large, fixed hardware cores that may allocate different logical units accessed with different channels to perform operations on sensitive or insensitive data. However, such approaches cannot be easily adapted or extended to precisely analyze cores for FPGA's, which typically treat the same input channel as a source of sensitive or insensitive information, depending on application-specific conditions on input values. When applied to analyze such cores, such analyses would raise a prohibitive number of false positives. SIMAREL is designed to precisely verify flow safety for policies that express conditions on input data under which sensitive information should be declassified. SIMAREL verifies such cores and policies by finding inductive relational invariants over pairs of core states, using precise symbolic techniques to avoid such inaccuracies.

Several lines of work have pursued generalizing strict non-interference to enable the expression of the release of partial information about sensitive inputs that may be declassified [5, 34, 55, 62, 66]. Policy languages and programming languages have been proposed that generalize non-interference by extending it with mechanisms that can be used to express under what conditions sensitive information can be declassified, to what channels it may be declassified, and after what program operations it may be declassified [55]. Such policy languages are related in their goal to STREAMS, in particular approaches that are formalized as equivalence relations over system states [4]. However, many of the above languages [34, 62, 66] were developed originally as extensions to program type systems [5]; adapting the concepts developed for such languages to specify properties of hardware systems has not been fully developed. The conditioned directives in STREAMS in particular are similar to features in such languages that define after what actions information may be declassified. The key distinction between STREAMS and such languages is that in STREAMS, conditions can be arbitrary conditions expressed in propositional logic, but such policies can be verified fully automatically by SIMAREL.

Previous work has also developed proof calculi for proving general relational properties of multiple program executions, founded on *relational Hoare-logic* [8]. Relational Hoare logic provides a powerful framework for proving rich properties of programs, which need not necessarily use bounded storage. However, it does not provide techniques for inferring proofs in the system automatically. Our approach is related, in that we view our policy-satisfaction problem as verifying a property over all pairs of program runs. However, our approach is restricted to a much more limited domain than that addressed by relational Hoare logic, namely sequential circuits that operate over a bounded space of memory. Focusing on this domain has enabled us to develop a fully automatic prover.

Previous work has shown that the problem of verifying that a program P satisfies a two-trace property Q can be reduced to verifying that the self-composition of P satisfies a safety property derived from Q [7, 59]. Self-composition is primarily applicable to verifying the flow safety of programs that do not operate over input streams; as a result, it is not directly applicable to verifying conditional information release of sequential circuits with bounded storage (see §IV-C). Previous work has proposed that relational properties of two programs P_0 and P_1 can be verified by constructing a suitable product program of P_0 and P_1 and deriving inductive invariants of the product [6]. SIMAREL, which considers pairs of runs of a circuit, uses a similar observation. However, previous work assumes that product programs are constructed manually, whereas SIMAREL verifies the flow security of circuits automatically.

c) Symbolic verification: Several model checkers have been proposed that attempt to efficiently verify a core or program by modeling the transition relation of the system as a symbolic formula [14, 38, 41]. In particular, several approaches have been proposed for finding a proof that a core satisfies a specification by synthesizing interpolants that prove that particular executions of the system are correct [27, 29, 39]. Interpolation problems have been introduced that accurately model the problem of proving the safety of intraprocedural traces [40], and interprocedural traces [26], to find flow-sensitive invariants [2], and to simultaneously prove that multiple traces of a program satisfy a given safety property [54].

SIMAREL is similar to the above approaches, in that it proves that a give core C is flow-secure by generalizing from proofs that bounded runs of C are flow-secure, which are synthesized from interpolants. However, unlike the above approaches, SIMAREL checks if a given core satisfies a property defined over all *pairs* of runs. As a result, SIMAREL synthesizes a proof that all pairs of runs up to a bound are flow secure, using a novel procedure (§IV-B1) that cannot be efficiently simulated using techniques proposed in the above work.

VII. CONCLUSION

We have presented a policy language, named STREAMS, for expressing information flow policies with declassification for sequential core designs. We have also presented an automatic verifier, named SIMAREL, that determines if a given core design satisfies a given STREAMS policy. Proving that a given system satisfies such a property amounts to synthesizing invariants of a suitable product system. SIMAREL finds relational invariants for such a system efficiently by using a novel procedure for efficiently synthesizing relational invariants that prove the flow security of all pairs of runs of a system up to a bounded length.

We have written policies in STREAMS for cores that implement several application and control subsystems. We used SIMAREL to determine that several open-source cores satisfy expected information flow policies. We also used it to prove that other designs, in particular a flash controller, an SD card controller, a robotics controller, a DSP module, a debugging interface, and an Ethernet controller, allow surprising leaks of sensitive information.

Acknowledgements

We thank the anonymous reviewers for their insightful comments and feedback. We also thank David Heath and Amir Yazdanbakhsh for their close and ongoing collaboration on related work. This work was supported in part by NSF awards CNS # 1526211, CCF # 1553192, ECCS # 1609823; Semiconductor Research Corporation contract # 2015-TS-2636; and gifts from Google, Qualcomm, and Microsoft.

References

[1] AxBench: approximate computing benchmarks. http://axbench.org, 2016.
[2] A. Albarghouthi, A. Gurfinkel, and M. Chechik. Craig interpretation. In *SAS*, 2012.
[3] Altera Corporation. Altera SoCs. http://www.altera.com/devices/processor/soc-fpga/overview/proc-soc-fpga.html, 2016.
[4] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *SP*, 2007.
[5] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *SP*, 2008.
[6] G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *FM*, 2011.
[7] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW*, 2004.
[8] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *POPL*, 2004.
[9] U. Berkeley. Berkeley logic interchange format (BLIF). *Oct Tools Distribution*, 2:197–247, 1992.
[10] R. K. Brayton and A. Mishchenko. ABC: an academic industrial-strength verification tool. In *CAV*, 2010.
[11] H.-C. Chang, C. B. Shung, and C.-Y. Lee. A Reed-Solomon product-code (RS-PC) decoder chip for DVD applications. *SSC*, 36(2):229–238, 2001.
[12] E. S. Chung, J. D. Davis, and J. Lee. LINQits: Big data on little clients. In *ISCA*, 2013.
[13] E. S. Chung, J. C. Hoe, and K. Mai. CoRAM: An in-fabric memory architecture for FPGA-based computing. In *FPGA*, 2011.
[14] E. M. Clarke, K. L. McMillan, S. V. A. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *CAV*, 1996.
[15] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6), 2010.
[16] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *SP*, 2009.
[17] Crowd Supply. Snickerdoodle dev board boasts arm processor with onboard FPGA. https://www.crowdsupply.com/krtkl/snickerdoodle, 2016.
[18] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.
[19] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, 2003.
[20] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, 2010.
[21] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.
[22] J. Gantz and D. Reinsel. Extracting value from chaos. http://www.emc.com/collateral/analyst-reports/idc-extracting-value-from-chaos-ar.pdf.
[23] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, 2015.
[24] J. Graf, M. Hecker, M. Mohr, and G. Snelting. Tool demonstration: Joana. In *POST*, 2016.
[25] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward dark silicon in servers. In *MICRO*, 2011.
[26] M. Heizmann, J. Hoenicke, and A. Podelski. Nested interpolants. In *POPL*, 2010.
[27] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.
[28] Intel Corporation. Intel completes acquisition of Altera. https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/, 2015.
[29] R. Jhala and K. L. McMillan. Interpolant-based transition relation approximation. In *CAV*, 2005.
[30] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *CRYPTO*, 1996.
[31] A. laboratory. ARITH project: High-level design methodology for integer/Galois-field arithmetic circuits for embedded systems. http://www.aoki.ecei.tohoku.ac.jp/arith/.
[32] D.-U. Lee, J. D. Villasenor, W. Luk, and P. H. Leong. A hardware Gaussian noise generator using the Box-Muller method and its error analysis. *IEEE Computers*, 55(6):659–671, 2006.
[33] H. Lee. A high-speed low-complexity Reed-Solomon decoder for optical communications. *TCAS-II*, 52(8):461–465, 2005.
[34] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL*, 2005.
[35] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong. Sapper: a language for hardware-level security policy enforcement. In *ASPLOS*, 2014.
[36] X. Li, M. Tiwari, J. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf. Caisson: a hardware description language for secure information flow. In *PLDI*, 2011.
[37] D. Mahajan, J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. Kim, and H. Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. In *HPCA*, 2016.
[38] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *CAV*, 2002.
[39] K. L. McMillan. An interpolating theorem prover. In *TACAS*, 2004.
[40] K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.
[41] K. L. McMillan. Interpolants and symbolic model checking. In *VMCAI*, 2007.
[42] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. SNNAP: Approximate computing on programmable socs via neural acceleration. In *HPCA*, 2015.
[43] A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, 1999.
[44] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, 1997.
[45] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *SP*, 1998.
[46] J. Norhuzaimin and H. Maimun. The design of high speed UART. In *AEMC*, pages 306–310, 2005.
[47] opencores.org. http://opencores.org/project, 2016.
[48] opencores.org. http://opencores.org/project,opencores,announcement, 2016.
[49] S. Paul, N. Jayakumar, and S. P. Khatri. A fast hardware approach for approximate, efficient logarithm and antilogarithm computations. *IEEE VLSI*, 17(2):269–277, 2009.

[50] J. S. Plank et al. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Softw Pract Exp.*, 27(9):995–1012, 1997.

[51] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J. Kim, S. Lanka, J. R. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*, 2014.

[52] A. R. Putnam, D. Bennett, E. Dellinger, J. Mason, and P. Sundararajan. CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures. In *FPGA*, 2008.

[53] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[54] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for horn-clause verification. In *CAV*, 2013.

[55] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW-18*, 2005.

[56] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *SP*, 2010.

[57] F. Standaert, S. B. Örs, J. Quisquater, and B. Preneel. Power analysis attacks against FPGA implementations of the DES. In *FPL*, 2004.

[58] F. Standaert, L. van Oldeneel tot Oldenzeel, D. Samyde, and J. Quisquater. Power analysis of FPGAs: How practical is the attack? In *FPL*, 2003.

[59] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *SAS*, 2005.

[60] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *ISCA*, 2011.

[61] M. Tiwari, H. M. G. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *ASPLOS*, 2009.

[62] J. A. Vaughan and S. Chong. Inference of expressive declassification policies. In *SP*, 2011.

[63] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: Reducing the energy of mature computations. In *ASPLOS*, 2010.

[64] C. Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.

[65] Xilinx, Inc. All programmable SoC. http://www.xilinx.com/products/silicon-devices/soc/, 2016.

[66] S. Zdancewic and A. C. Myers. Robust declassification. In *CSFW-14*, 2001.

[67] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *PLDI*, 2012.

[68] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for timing-sensitive information-flow security. In *ASPLOS*, 2015.

Appendix

Let $C \in \mathsf{Cores}$, $F \in$ STREAMS, and $L \in \mathsf{Lvs}$ be fixed for the remainder of this section, as in §IV. In this section, we provide a correctness proof of SIMAREL by proving results concerning C, F, and L.

A. Indexed invariants as evidence of policy satisfaction

Relational invariants of C are a symbolic relation that **(1)** is entailed by the initial condition of C and **(2)** combined with the semantics of taking a step in each of two copies of state, entail itself.

Definition 8. *Let* $R_0, R_1 \in \mathsf{SymRels}$ *be such that:*

$$\mathsf{IsInit}_C[\mathtt{Q}_0], \mathsf{IsInit}_C[\mathtt{Q}_1], \mathsf{InEq}, \mathsf{OutEq} \models R_0 \vee R_1 \quad (3)$$

$$R_0, \psi_C[\mathtt{I}_0, \mathtt{Q}_0, \mathtt{Q}_0'], \neg\mathsf{En}[\mathtt{I}_0] \models (R_0 \vee R_1)[\mathtt{Q}_0'/\mathtt{Q}_0] \quad (4)$$

$$R_1, \psi_C[\mathtt{I}_1, \mathtt{Q}_1, \mathtt{Q}_1'], \neg\mathsf{En}[\mathtt{I}_1] \models (R_0 \vee R_1)[\mathtt{Q}_1'/\mathtt{Q}_1] \quad (5)$$

$$\begin{aligned} &R_0 \vee R_1, \psi_C[\mathtt{I}_0, \mathtt{Q}_0, \mathtt{Q}_0'], \psi_C[\mathtt{I}_1, \mathtt{Q}_1, \mathtt{Q}_1'], \\ &\mathsf{En}[\mathtt{I}_0], \mathsf{En}[\mathtt{I}_1], \\ &(\mathsf{InEq} \wedge \mathsf{LvIns}_0 = \mathsf{LvIns}_1 \implies \mathsf{InEq}'), \\ &(\mathsf{OutEq} \wedge \mathsf{LvOuts}_0 = \mathsf{LvOuts}_1 \implies \mathsf{OutEq}') \models \\ &R_0 \vee R_1[\mathtt{Q}_0', \mathtt{Q}_1', \mathsf{Eqs}'] \end{aligned} \quad (6)$$

Then (R_0, R_1) *are* inductive relational invariants *of* C.

Relational invariants R satisfy F at L if R, combined with the assumption that two arbitrary input streams visible at L are equivalent, imply that the resulting output streams visible at L are equivalent.

Definition 9. *For* $R \in \mathsf{SymRels}$ *such that* $R, \mathsf{InEq} \models \mathsf{OutEq}$, R satisfies F *at* L.

The fact that R satisfies F at L is denoted $R \vdash_L F$.

Inductive relational invariants of C that satisfy F at L are evidence that C satisfies F at L.

Lemma 1. *If there are* $R_0, R_1 \in \mathsf{SymRels}$ *such that* (R_0, R_1) *are inductive relational invariants (Defn. 8) and* $R_0 \vee R_1 \vdash_L F$*, then* $C \vdash_L F$ *(Defn. 4).*

Proof. (Sketch) The claim can be established directly by double induction on the pairs of runs of C. The inductive step follows from the inductive hypothesis and the definitions of inductive relational invariants (Defn. 8) and policy satisfaction by inductive relational invariants (Defn. 9). □

For $I \in \mathsf{IdxRels}$, let R_I^0 be

$$\bigvee\{I(i,j) \mid i,j \in \mathbb{N}, (i,j),(i+1,j) \in \mathsf{Dom}(I)\}$$

Let R_I^1 be

$$\bigvee\{I(i,j) \mid i,j \in \mathbb{N}, (i,j),(i,j+1) \in \mathsf{Dom}(I)\}$$

Lemma 2. *For* $I \in \mathsf{IdxInvs}$, (R_I^0, R_I^1) *are inductive relational invariants (Defn. 8).*

Proof. (Sketch) Apply the definition of inductive relational invariants (Defn. 8). Eqn. 3 holds by the fact that IsInit_C entails $I(0,0)$ (§IV-B1, Defn. 5) and the definition of R_I^0 and R_I^1. For Eqn. 4—Eqn. 6 are proven by applying the fact that R_I^0 and R_I^1 are disjunctions of clauses indexed in I, and applying the fact that each clause in I satisfies an analogous condition, by Defn. 5. □

Lemma 3. *For* $I \in \mathsf{IdxRels}$*, if* $I \vdash_L F$*, then* $R_I^0 \vee R_I^1 \vdash_L F$.

Proof. R_I^0 and R_I^1 are disjunctions of clauses. For each such clause C, $C, \mathsf{LvIns}_0 = \mathsf{LvIns}_1 \models \mathsf{LvOuts}_0 = \mathsf{LvOuts}_1$. Thus, $R_I^0, \mathsf{LvIns}_0 = \mathsf{LvIns}_1 \models \mathsf{LvOuts}_0 = \mathsf{LvOuts}_1$, and similarly for R_I^1. Thus $R_I^0 \vee R_I^1 \vdash_L F$, by Defn. 9. □

If C has inductive indexed relational invariants that satisfy F at L, then C satisfies F at L.

Lemma 4. *If there are $I \in \mathsf{IdxInvs}$ such that I are inductive indexed relational invariants of C (Defn. 6) and $I \vdash_L F$ (Defn. 7), then $C \vdash_L F$ (Defn. 4).*

Proof. (R_I^0, R_I^1) are inductive invariants, by Lemma 2. $R_I^0 \vee R_I^1 \vdash_L F$, by Lemma 3. Thus, $C \vdash_L F$, by Lemma 1. □

B. *Correctness of* CHK

The following lemmas concerning CHK are sufficient to prove the soundness and completeness of SIMAREL.

Lemma 5. *For $k \in \mathbb{N}$, if $\text{CHK}(k) = \mathsf{Unsafe}$, then $C \not\vdash_L F$.*

Proof. (Sketch) If $\text{CHK}(k) = \mathsf{Unsafe}$, then $\varphi_{0,0}^- \wedge \varphi_{0,0}^+$ has a model m, by the definition of CHK (§IV-B3). The interpretation of the variables $\mathtt{I}_i^0$, $\mathtt{Q}_i^0$, and $\mathtt{O}_i^0$ defines a run r_0 of C that starts in an initial state, by the use of IsInit_C in $\varphi_{0,0}^-$, and the use of Rest_0^0 in $\varphi_{0,0}^+$. Similarly, the interpretation of the variables $\mathtt{I}_i^1$, $\mathtt{Q}_i^1$, and $\mathtt{O}_i^1$ defines a run r_1 of C.

There are some indices $i, j < k$ such that $\mathsf{InEq}_{i,j}$ holds and $\mathsf{OutEq}_{i,j}$ does not hold. It can be established by induction on step indices that the interpretations of $\mathtt{I}_i^0$ and $\mathtt{I}_j^1$ that satisfy the enabling condition of L in F are equal, but the interpretations of $\mathtt{O}_i^0$ and $\mathtt{O}_j^1$ are not equal. As a result, run 0 up to i and run 1 up to j are runs of C on inputs that are equivalent at L that result in outputs that are not equivalent at L. Thus, C does not satisfy F at L. □

CHK, given k, only returns relational invariants if they are indexed relational invariants of C, F, and L that prove that C satisfies F at L up to k.

Lemma 6. *For $k \in \mathbb{N}$, if $I = \text{CHK}(k) \in \mathsf{IdxRels}$, then **(1)** $I \in \mathsf{IdxInvs}$ (Defn. 6) and **(2)** $I \vdash_L F$ (Defn. 7).*

Proof. (Sketch) Conclusion **(1)** follows by induction on the topological ordering T of $\mathbb{Z}_k \times \mathbb{Z}_k$ in which CHK finds relational invariants. The claim established by induction is that at the current indices i and j, I restricted to all pairs of indices in T up to (i, j) are inductive relational invariants. The base case and inductive case both follow from the construction of $\varphi_{i,j}^-$ and $\varphi_{i,j}^+$ (§IV-B3) and the definition of interpolants (§III-B2, Defn. 1).

Conclusion **(2)** follows by induction on the topological ordering of $\mathbb{Z}_k \times \mathbb{Z}_k$. The claim established by induction is that the current indices i and j, at all indices i', j' such that (i', j') occurs before (i, j) in T, $I(i', j')$ combined with $\mathsf{LvIns}_0 = \mathsf{LvIns}_1$, entails $\mathsf{LvOuts}_0 = \mathsf{LvOuts}_1$. The claim is established in the inductive step by using the definition of $\varphi_{i,j}^+$, in particular $\mathsf{PolSat}_{i,j}$. □

C. *Correctness of* HASIND

The following lemma concerning HASIND is sufficient to prove correctness of SIMAREL.

Lemma 7. *For $I \in \mathsf{IdxRels}$, if $\text{HASIND}(I) = \mathsf{True}$, then $I \in \mathsf{IdxInvs}$ (Defn. 6).*

Proof. Proof by induction on the evaluation of HASINDAUX run on obligations O and discharged pairs D. The claim established by induction is that I, restricted to the index pairs in D and extended with indexed symbolic relations that contain O, are inductive indexed invariants. For the base case, HASIND is called on $\{(0, 0)\}$ and $\emptyset$ (Alg. 2), which combined with the definition of inductive relational invariants (Defn. 6), implies the claim.

For the inductive case, for each set of obligations O and discharged indices D on which HASIND is called, it calls itself recursively on a set of obligations O', construted as O with some pair of indices (i, j) removed. If $I(i, j)$ entails $I(i', j')$ for some $i', j' \in \mathbb{N}$ and $(i', j') \in D$, then HASIND calls itself recursively on O' and D' (Alg. 2, line 8). In this case, the claim is established by Defn. 6, clause **(a)**. Otherwise, HASIND calls itself recursively on O' extended with the successor of (i, j) in run 0 (Alg. 2, line 10) or run 1 (Alg. 2, line 11). In these cases, the claim is established by Defn. 6, clauses **(b)** and **(c)**, respectively.

HASIND returns True, which it implies that it was evaluated on the empty set of obligations, by Alg. 2. This fact, combined with the claim established by induction, implies that $I \in \mathsf{IdxInvs}$. □

D. *Correctness of* SIMAREL

A proof of Thm. 1 is as follows.

Proof. If $\text{SIMAREL}(C, F, L) = \mathsf{True}$, then for some $k \in \mathbb{N}$ and $I \in \mathsf{IdxRels}$, $I = \text{CHK}(C, F, L, k)$, by Alg. 1. I satisfies F at L by Lemma 6. $\text{HASIND}(I) = \mathsf{True}$, by Alg. 1. I are inductive indexed relational invariants, by Lemma 7. C satisfies F at L, by Lemma 4. □