APPENDIX

In §V-C2, we claim that SYNTHLC produces a *complete* set of leakage signatures, which capture all violations of *hardware side-channel safety*.

Below we present a formal proof on this claim. To do so, we first presents a microarchitectural execution semantics (§A) so as to define hardware side-channel safety (i.e., Def. V.1) *formally* (§A1). Next we concretize our receiver $R_{\mu\text{PATH}}$ (§V-C2 of main paper) by formally defining $\mathcal{O}_{\mathcal{R}_{\mu\text{PATH}}}$ (§A2). Then we define in §B the μPATHs and decisions formally in order to support the formal proof on leakage signature synthesis in the next section (§C).

A. Microarchitectural Execution Semantics

This section presents an execution semantics for an abstract processor microarchitecture (inspired by [43]) to support such a security definition.

Consider a processor *microarchitecture* $\mathcal{M} = (S_0, S, T)$, where S is a set of *states*, $S_0 \subseteq S$ is a set of *initial states*, and $T \subseteq S \times S$ is a deterministic *transition relation*.

A processor state $\langle \sigma, \mu \rangle \in S$ consists of an architectural state σ and a microarchitectural state μ. An *architectural state* $\sigma = \langle m, a \rangle$ denotes the contents of *main memory* $m : loc \rightarrow \mathcal{V}$ and the contents of the *architectural register file* $a : reg_a \rightarrow \mathcal{V}$, where $\mathcal{V}$ is the set of values that memory locations and registers can assume, loc is a set of memory locations, and reg_a is a set of architectural registers. A *microarchitectural state* $\mu : reg_\mu \rightarrow \mathcal{V}$ denotes the contents of all other state elements in the processor, where reg_μ is a set of microarchitectural registers.

Consider state $s = \langle \sigma, \mu \rangle$, where $\sigma = \langle m, a \rangle$. We use $s(\sigma)$ and $s(\mu)$ to refer to the architectural and microarchitectural state components of s; $\sigma(m)$ and $\sigma(a)$ to refer to the main memory and architectural register file components of σ; $m(j)$ to refer to the value assumed by memory location j in m; $s(\sigma)(m)(j)$ to refer to the value assumed by memory location j in s; and so on.

An execution *trace* $\langle \sigma_0, \mu_0 \rangle, \langle \sigma_1, \mu_1 \rangle, ...$ of M is an infinite sequence of states, such that $\langle \sigma_0, \mu_0 \rangle \in S_0$, and $(\langle \sigma_k, \mu_k \rangle, \langle \sigma_{k+1}, \mu_{k+1} \rangle) \in T$ for $k \geq 0$. By $\{\!|p|\!\}_{\mathcal{M}}^{\langle \sigma, \mu \rangle}$ we denote the execution trace obtained from running a *program* p on M from initial state $\langle \sigma, \mu \rangle$, such that $\sigma(m)$ is initialized with p's instructions and data and $\mu(\texttt{IM\_PC})$ contains the address of the first instruction to fetch in p. IM_PC is the *instruction memory program counter* signal in M [48]. If $\langle \sigma, \mu \rangle \notin S_0$, then $\{\!|p|\!\}_{\mathcal{M}}^{\langle \sigma, \mu \rangle}$ returns the empty trace. By $t[n]$ we denote the state corresponding to the nth *step/cycle* of trace t, i.e., $t[0] = \langle \sigma, \mu \rangle$ for $t = \{\!|p|\!\}_{\mathcal{M}}^{\langle \sigma, \mu \rangle}$.

1) Security Definition: We define a program's susceptibility to hardware side-channel attacks using an *information-flow* property below [40], [76]. Our definition assumes a *static privacy policy* $\pi : loc \rightarrow \{\text{L}, \text{H}\}$ that labels memory locations in m as storing *low* (L, public) or *high* (H, private) data. A program's instruction memory is always low [33], [69].[6]

Hardware side-channel safety is defined with respect to an observer model $\mathcal{O}_\mathcal{R}$. Given state $\langle \sigma, \mu \rangle \in S$, $\mathcal{O}_\mathcal{R}(\langle \sigma, \mu \rangle)$ returns the value of each memory location, architectural register, and microarchitectural register that is assumed to be *visible* to *receiver* $\mathcal{R}$. We lift functions over sequence elements to functions over sequences in the natural way, e.g., for execution trace $t = \langle \sigma_0, \mu_0 \rangle, \langle \sigma_1, \mu_1 \rangle, ...$, we define the *observation trace* obtained by receiver R from t as $\mathcal{O}_\mathcal{R}(t) = \mathcal{O}_\mathcal{R}(\langle \sigma_0, \mu_0 \rangle), \mathcal{O}_\mathcal{R}(\langle \sigma_1, \mu_1 \rangle), ...$ Industrial leakage contracts [2], [11], [49] assume a receiver $\mathcal{R}_C$ that observes M's commit signal, or $\mathcal{O}_{\mathcal{R}_C}(\langle \sigma, \mu \rangle) = \mu(\texttt{commit})$, modeling a receiver that perceives channel modulations through their impact on instruction or program execution time.

a) Program Assumptions: : For virtually all realistic receivers, including $\mathcal{R}_C$, any program that features secret-dependent control-flow can leak the secret. For example, consider the code listing, $\texttt{if(sec)\{seq /*latency L*/\}else\{seq' /*latency L'*/\}}$, which features a secret-dependent conditional branch. Clearly, and independent of microarchitecture, there exists instruction sequences seq and seq′ whose latencies L and L′, respectively, are different ($\text{L} \neq \text{L}'$). Given our focus on *microarchitectural* side-channels, we assume that all programs p feature the same sequence of instructions along all branches of secret-dependent control-flow instructions, i.e., $\texttt{seq} = \texttt{seq}'$ for the code listing above. That is, we assume p satisfying $ArchCtrl(p)$ (§V-C2).

With the above foundation, Def. V.1 in §V-C2 of main paper (copied below) is formally defined.

Definition A.1 (Hardware Side-Channel Safe)**.** A microarchitecture M is *hardware side-channel safe* with respect to receiver R, or SC-Safe(M, R), iff:

$$\begin{aligned} &\forall p. \forall \pi. \forall \sigma, \sigma'. \forall \mu. ArchCtrl(p) \implies \\ &(\sigma \approx_\pi \sigma' \implies O_R(\{\!|p|\!\}_M^{\langle \sigma, \mu \rangle}) = O_R(\{\!|p|\!\}_M^{\langle \sigma', \mu \rangle})) \end{aligned} \tag{2}$$

Eq. 2 quantifies over all programs p and security policies π (which label program inputs as public or secret), all pairs of initial architectural states σ, σ' and all initial microarchitectural states μ. Looking at the second line, the antecedent, $\sigma \approx_\pi \sigma'$, checks that the initial architectural states are *low-equivalent* with respect to π: they agree on the values of low data in π, i.e., p's public data inputs. The consequent, $O_R(\{\!|p|\!\}_M^{\langle \sigma, \mu \rangle}) = O_R(\{\!|p|\!\}_M^{\langle \sigma', \mu \rangle})$, asserts that R obtains identical observation traces when p runs on microarchitecture M from initial states $\langle \sigma, \mu \rangle$ and $\langle \sigma', \mu \rangle$.

Eq. 2 violations indicate the observation trace obtained by receiver R from running program p with privacy policy π on microarchitecture M *indisputably* a function of p's high inputs. Note that secret-dependent control-flow instructions can still cause Eq. 2 violations, e.g., if they trigger operand-dependent squashes.

A violation of Eq. 2 indicates that the observation trace obtained by receiver R from running program p with privacy policy π on microrarchitecture M is *indisputably* a function of p's high inputs. Intuitively, with respect to $\mathcal{R}_C$, Eq. 2 is valid if no instruction in p can influence the commit signal's value as a function of its operands.

Other Security Definitions: Some prior definitions of hardware side-channel safety omit μ in Eq. 2, leaving the initial microarchitectural states for pairs of traces compared by the consequent implicitly unconstrained [32], [33], [90], [91]. However, like prior work [43], [97], we observe that

[6] W.l.o.g., *only* locations in m are assigned security types in *static* privacy policy π, and taint does not propagate, e.g., to elements in r or μ.

constraining these traces' initial microarchitectural states (e.g., to be equivalent)[7] is required to prevent Eq. 2 from *misclassifying* a safe program p running on microarchitecture M as *unsafe* with respect to security policy π and receiver R. As an example, consider $p = \{\texttt{LI r1,3;SUB r1,r1,r1}\}$ executing on M from initial states, $\langle\sigma,\mu\rangle$ and $\langle\sigma',\mu'\rangle$, where $\mu \neq \mu'$ and $\sigma \approx_\pi \sigma'$. Suppose μ and μ' feature an in-flight MUL with at least one zero operand in μ and no zero operands μ', and M implements zero-skip multiplies (§I). It is possible that $\mathcal{O}_{\mathcal{R}_C}(\{\!|p|\!\}_{\mathcal{M}}^{\langle\sigma,\mu\rangle}) \neq \mathcal{O}_{\mathcal{R}_C}(\{\!|p|\!\}_{\mathcal{M}}^{\langle\sigma',\mu'\rangle})$ in Eq. 2, despite p's execution time being *independent* of its high inputs.

Some procedures for verifying hardware side-channel safety side-step the imprecision of a security definition that omits μ by assuming equivalent initial microarchitectural states for all pairs of compared traces in their *implementations* [32], [33]. Other such procedures directly check this imprecise security definition and require users to navigate false counterexamples as a result [90], [91].

2) Receiver Assumptions: In §V-C2 of main paper, we assume a receiver $\mathcal{R}_{\mu\text{PATH}}$ and thus $\mathcal{O}_{\mathcal{R}_{\mu\text{PATH}}}$ abstractly. Here we provide the formal definition of it.

We define $\mathcal{O}_{\mathcal{R}_{\mu\text{PATH}}}$ using one auxiliary definition. We say that "dynamic instruction i *visits* PL pl in state $\langle\sigma,\mu\rangle$," or $visit(i,pl,\langle\sigma,\mu\rangle)$, iff $\mu(pl.\texttt{IIR}) = i.\texttt{II} \wedge \mu(pl.\mu\texttt{FSM}) = pl.\texttt{state}$. Thus, $\mathcal{O}_{\mathcal{R}_{\mu\text{PATH}}}(\langle\sigma,\mu\rangle) = \{pl \mid \exists i.visit(i,pl,\langle\sigma,\mu\rangle)\}$, or $R_{\mu\text{PATH}}$ observes, at each cycle, the set of PLs visited by in-flight instructions.

Informally, for $\mathcal{R}_{\mu\text{PATH}}$, Eq. 2 is valid for an arbitrary program (that satisfies our assumptions in §A1a) if no instruction can create μPATH variability for any instruction (including itself) as a function of its operands on the target microarchitecture. Note that $\mathcal{R}_{\mu\text{PATH}}$ is strictly more powerful than $\mathcal{R}_C$ (§A1), capturing a receiver that perceives channel modulations via their impact on execution time and more [9], [14], [47], [52], [67], [77], [89], [100].

B. μPATHs And Decisions Formalized

1) μPATHs Formalized: We define $\boldsymbol{\mu\text{PATH}}_{\mathcal{M}}^{I}$ as the set of all possible μPATHs that a dynamic instance of instruction *type* I may exhibit when it runs on micraorchitecture M. Each $p \in \boldsymbol{\mu\text{PATH}}_{\mathcal{M}}^{I}$ is a tuple (V,E), where V and E are sequences of *sets of PLs* and *sets of happens-before edges*, respectively, as defined below.

Consider an execution trace t, where a dynamic instruction of type I, i_I, materializes in M (i.e., appears in some PL) in state $t[j]$, and dematerializes (i.e., disappears from all PLs) in state $t[k]$, where $j < k$. Suppose i_I exhibits $p = (V,E) \in \boldsymbol{\mu\text{PATH}}_{\mathcal{M}}^{I}$ from $t[j]$ through $t[k-1]$. We define $V = (v_0, v_1, ..., v_{k-j-1})$, where $v_n = \{pl \mid visit(i_I, pl, t[j+n])\}$ captures the set of PLs visited by i_I in step n of its execution. We define $E = (e_0, e_1, ..., e_{k-j-2})$, where $e_n = \{(pl,pl') \mid con(pl,pl') \wedge pl \in v_n \wedge pl' \in v_{n+1}\}$ captures happens-before ordering between i_I's visits to PLs in consecutive trace steps.

[7]Eq. 2 must constrain the initial microarchitectural states in both traces to be *non-interfering w.r.t. p and* π. Any μ and μ' for which $\mathcal{O}_{\mathcal{R}}(\{\!|p|\!\}_{\mathcal{M}}^{\langle\sigma,\mu\rangle}) = \mathcal{O}_{\mathcal{R}}(\{\!|p|\!\}_{\mathcal{M}}^{\langle\sigma,\mu'\rangle})$ holds in the antecedent of Eq. 2 suffices, but we select $\mu = \mu'$ for simplicity.

Above, the predicate $con(pl,pl')$ returns true iff μFSMs corresponding to pl and pl' are connected purely by combinational logic in M's netlist. It requires that μPATHs exclusively capture *causal* happens-before relationships among visited PLs.

Note that we use upper-case, like I, to denote an instruction *type* (e.g., MUL rd, rs1, rd2) and lower-case, like i to denote a *dynamic* instruction; i_I denotes a *dynamic* instruction of type I.

2) Decisions: Recall a μPATH *decision* $d_{\mathcal{M}}^{I} = (src, \mathbf{dst})$ made by an instruction I on microarchitecture M is a tuple, where src is a single *decision source* PL and $\mathbf{dst}$ is a set of *decision destination* PLs, that distinguishes *at least two* of I's μPATHs on M. We define the set of all decisions $\mathbf{d}_{\mathcal{M}}^{I}$ that I makes on M below.

For μPATH $p = (V,E)$, let $\mathbf{pl}_p = \{pl \mid \exists v_n \in V.pl \in v_n\}$ select the set of all PLs that are visited in p. Let $next(p,pl) = \{pl' \mid \exists e_n \in E.(pl,pl') \in e_n\}$ select the set of all PLs visited the cycle after pl in p. We define the set of I's decision source PLs on M as $\mathbf{src}_{\mathcal{M}}^{I} = \{pl \mid \exists p,p' \in \boldsymbol{\mu\text{PATH}}_{\mathcal{M}}^{I}.\ pl \in \mathbf{pl}_p \wedge pl \in \mathbf{pl}_{p'} \wedge next(p,pl) \neq next(p',pl)\}$. In other words, a decision source $src \in \mathbf{src}_{\mathcal{M}}^{I}$ is followed by a distinct set of *decision destinations* in at least two of I's μPATHs on M. We then define the set of μPATH decisions for I on M as $\mathbf{d}_{\mathcal{M}}^{I} = \{(src, \mathbf{dst}) \mid \exists p \in \boldsymbol{\mu\text{PATH}}_{\mathcal{M}}^{I}.src \in \mathbf{src}_{\mathcal{M}}^{I} \wedge \mathbf{dst} = \{pl \mid pl \in next(p,src)\}\}$.

C. Formal Foundation For Leakage Signature Synthesis

We now prove our claim that SYNTHLC produces a *complete* set of leakage signatures, which capture all violations of *hardware side-channel safety* (Def. A.1).

1) Counterexample Discovery: Finding all *counterexamples* to Eq. A.1 given an M, and $R_{\mu\text{PATH}}$, equates to finding all satisfying *solutions* to:

$$\exists p.\exists \pi.\exists \sigma,\sigma'.\exists \mu.(p) \wedge \sigma \approx_\pi \sigma' \wedge \underbrace{\mathcal{O}_{\mathcal{R}_{\mu\text{PATH}}}(\{\!|p|\!\}_{\mathcal{M}}^{\langle\sigma,\mu\rangle})}_{\mathcal{O}_A} \neq \underbrace{\mathcal{O}_{\mathcal{R}_{\mu\text{PATH}}}(\{\!|p|\!\}_{\mathcal{M}}^{\langle\sigma',\mu\rangle})}_{\mathcal{O}_B} \quad (3)$$

That is, SYNTHLC must find all ways in which a program p with security policy π can produce distinct observations for $R_{\mu\text{PATH}}$, called $\mathcal{O}_A$ and $\mathcal{O}_B$, when running on $\mathcal{M}$ from a pair of low-equivalent architectural states, σ and σ', and microarchitectural state, μ.

To do this, SYNTHLC leverages an important observation about all solutions to Eq. 3. Namely, all distinct valuations of $\mathcal{O}_A$ and $\mathcal{O}_B$ indicate that the same dynamic transponder i_P exhibits distinct decisions at the same decision source PL in traces $t = \{\!|p|\!\}_{\mathcal{M}}^{\langle\sigma,\mu\rangle}$ and $t' = \{\!|p|\!\}_{\mathcal{M}}^{\langle\sigma',\mu\rangle}$, as a function of H-labeled data in π. We provide the proof in section below (§C1a). Hence, all valuations of $\mathcal{O}_A$ and $\mathcal{O}_B$ in satisfying solutions to Eq. 3 are captured by identifying all decisions that may be exhibited by transponder instructions on M.

a) Counterexamples and Transponder Relation: Observe that traces t and t', as defined above, satisfy Eq. 3 iff:

$$\exists k.\forall j.0 \leq j < k \wedge \mathcal{O}_{\mathcal{R}_{\mu\text{PATH}}}(t[j]) = \mathcal{O}_{\mathcal{R}_{\mu\text{PATH}}}(t'[j]) \wedge \mathcal{O}_{\mathcal{R}_{\mu\text{PATH}}}(t[k]) \neq \mathcal{O}_{\mathcal{R}_{\mu\text{PATH}}}(t'[k]) \quad (4)$$

That is, there exists an *earliest* step k of t and t' at which the traces produce distinct observations for $\mathcal{R}_{\mu\text{PATH}}$. Given the

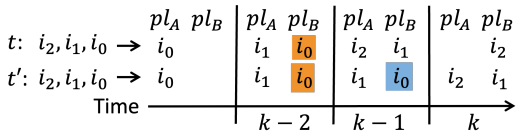

Fig. 8: Delayed observation at step k arising from transponder exhibiting distinct decision earlier than step $k-1$.

definition of $\mathcal{O}_{\mathcal{R}_{\mu\text{PATH}}}$ (§A2), the PLs occupied by all in-flight instructions are the same in t and t' from step 0 through step $k-1$, but different in step k. Distinct observations in step k indicates that some instructions i and i', in-flight in t and t', respectively, visited the same source (s) PL in step $k-1$, but distinct destination (d) PLs in step k:

$$\exists i, i'.\exists s.\exists d.visit(i, s, t[k-1]) \wedge visit(i', s, t'[k-1]) \\ \wedge\ visit(i, d, t[k]) \wedge \neg visit(i, d, t[k]) \quad (5)$$

Suppose i and i' are the *same* in-flight instruction. We say instructions in trace t and t' are the same if they have the same encoding and feature the same program counter. Different decisions made by the same instruction at the same decision source PL necessarily indicates different inputs to some path selector function. Since only high data can differ under Eq. 3's low-equivalence constraint and only program inputs can be H-labeled in π, said path selector function inputs must be operand-dependent on some transmitter. Hence, $i = i'$ is a transponder P (§IV).

Now suppose i and i' are *different* in-flight instructions, which may occur in two cases. We show that under our assumptions, some transponder i_P exhibits distinct decisions at the same decision source PL in t and t'.

Case 1 (Control-Flow): In the first case, t and t' feature distinct *fetch streams*, where a fetch stream denotes the sequence of instructions that populate M's *instruction fetch register* (IFR, §V-A). Recall however, given our program assumptions in §A1a, that the fetch streams are identical in t and t', yielding a contradiction. Note that SYNTHLC still flags control-flow instructions as transmitters in this setting, as they may conditionally flush mis-speculated instructions as a function of their operands.

Case 2 (Delayed Observation): In the second case, some same-instruction pair j and j', in traces t and t', respectively, exhibited different decisions at the same source PL some step *before* $k-1$. Hence, $j = j'$ is a transponder P. However, this μPATH variability was not detected until j and j' transitively impacted the execution of i and i', respectively.

Such scenario is illustrated in Fig. 8. At step $k-2$, $j = j' = i_0$ makes distinct decisions at pl_B in t compared to t'. In t, i_0 disappears from all PLs (i.e., progresses to the "empty" PL). In t', it remains in pl_B. However, distinct observations for $\mathcal{R}_{\mu\text{PATH}}$, involving $i = i_2$ and $i' = i_1$ where $i_1 \neq i_2$, do not manifest until later in step k.

Given the observation above, SYNTHLC's counterexample discovery using RTL2MμPATH over-approximates Eq. 3's solution space by first uncovering all possible μPATHs for each implemented instruction I on $\mathcal{M}$. Those instructions with more than one μPATH are flagged as *candidate transponders*. A set of decisions $\mathbf{d}^P_{\mathcal{M}}$ is collected for each candidate transponder P as described §B2. Enabled by user-annotated μFSMs (§V-A),

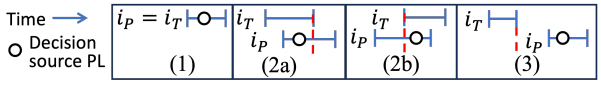

Fig. 9: Constraints on SVA properties to uncover and classify transmitters as (1) intrinsic, (2a/b) dynamic, or (3) static.

SYNTHLC conducts this counterexample discovery step automatically, using commercial model checkers [23] to evaluate *(single-)trace* properties on M (§V).

D. Leakage Signature Discovery

SYNTHLC's counterexample discovery procedure (§C1) over-approximates the solution space to Eq. 3, because unique decisions for *candidate* transponders may not represent legitimate valuations of $\mathcal{O}_A$ and $\mathcal{O}_B$ when p with π executes *from low-equivalent initial states* on M. Finding which decisions do represent valid solutions to Eq. 3, requires finding which decision may depend on H-labeled data in π. Since π labels program inputs, this is equivalent to finding which decisions may depend on *transmitter operands*.

1) Instantiating Eq. 3: To do this, SYNTHLC considers each candidate transponder in turn and systematically derives all of its leakage signatures. A candidate transponder may have *at most one* leakage signatures *per decision* (if it is not a true transponder, it will have none). Consider candidate transponder P and its various decisions on M (in $\mathbf{d}^P_{\mathcal{M}}$) that originate at decision source $src \in \mathbf{src}^P_{\mathcal{M}}$. Our goal is to derive a its leakage signature `P_src` (if one exists), which features four components: the signature's (i) *name* (i.e., "P" concatenated with PL "src"), (ii) *range* (i.e., the set of PL sets $\{\mathbf{dst} \mid (src, \mathbf{dst}) \in \mathbf{d}^P_{\mathcal{M}}\}$), (iii) labeled *transmitters* (i.e., explicit inputs labeled *intrinsic* (`N`), *static* (`S`), or *dynamic* (`D`)), and (iv) *unsafe operands* (i.e., operands of explicit inputs that appear in the function body). Clearly, we already have (i) and (ii).

We can obtain (iii) and (iv) by instantiating Eq. 3 in various ways as follows. First, select a pair of decisions $d_0, d_1 \in \mathbf{d}^P_{\mathcal{M}}$ for transponder P, which feature the same decision source src, but distinct decision destinations. Next, consider all possible (candidate) pairs (T, op) of a transmitter T and its unsafe operand op. For each pair, constrain program p according to the following three cases below (illustrated in Fig. 9) to determine if P exhibits decision d_0 versus d_1 as a function of op. For all cases, assume some $i_T, i_P \in p$ and that the only operand of any instruction in p to be supplied with high data is op of i_T specifically.

Case 1 determines whether T is an intrinsic transmitter with respect to `P_src` by constraining i_T and i_P to be the same dynamic instruction. *Case 2* determines whether T is a dynamic transmitter by constraining i_T to be in-flight when i_P visits its decision source src. *Case 3* determines whether T is a static transmitter by constraining i_T to have materialized and dematerialized in M *before* i_P reaches src. These cases are depicted in Fig.9, where *Case 2* is broken into two subcases: one where i_T materializes in M before i_P (*Case 2a*) and vice versa (*Case 2b*). If Eq. 3 is satisfied in any of these cases, we have *labeled* transmitter T with unsafe operand op.

2) Symbolic IFT: In theory, the various instances of Eq. 3 described can checked on M by: (i) constructing a *product circuit*, where M is instantiated twice, to check the property

directly [33], [97], or (ii) augmenting M with *information flow tracking* (IFT) circuitry to check the property indirectly [90], [91]. The first strategy is regarded as more precise [102]. However, with our goal of attributing transponder decisions to transmitter operands, it exhibits a soundness limitation as we discuss in a separate section (§E).

Hence, SYNTHLC checks Eq. 3 using symbolic IFT by augmenting M with cell-level IFT circuitry [82] to produce $\mathcal{M}_{IFT}$. For *Case 1* (intrinsic) and *Case 2* (dynamic) in §D1, $\mathcal{M}_{IFT}$ features a single "taint" bit per data bit in M's netlist. To enforce that only *op* of i_T may be supplied H-labeled data (§D1), SYNTHLC exclusively sets the taint bit associated with *op* and prevents taint propagation from i_T's output to the input of any other instruction (architecturally or speculatively). For *Case 3* (static) in §D1, $\mathcal{M}_{IFT}$ features two taint bits per data bit in M's netlist (§V-C). This implementation choice allows SYNTHLC to effectively flush "sticky" taint that is associated with *op*'s dynamic influence on transponders μPATHs, thereby considering its static influence exclusively. We briefly describe the implementation as follows.

Symbolic IFT for Static Transmitter: Taint flows exclusively within upper taint bits or lower taint bits. However, when a data bit's upper taint bit is set, it clears the lower taint bit.

To enforce that only *op* of i_T may be supplied with h-labeled data, SYNTHLC exclusively sets the *lower* taint bit associated with *op* and prevents taint propagation from i_T's output to the input of any other instruction (architectural or speculatively).

To uncover static transmitters, SYNTHLC effectively flushes i_T's dynamic influence by setting the *higher* taint bit associated with the operands of all instructions that are fetched after i_T and before i_P. These intermediate instructions are constrained such that when i_P is issued the lower taint bits for all μFSMs are cleared.

Given this two-bit IFT approach, the state space in Table **??** for SYNTHLC is 2^n for path exploration and 2^{3n} for symbolic taint tracking respectively.

3) Synthesizing Leakage Signatures: To synthesize a leakage signatures `P_src`, SYNTHLC considers each decision $(src, \mathbf{dst}) \in \mathbf{d}^P_{\mathcal{M}}$ paired with each (candidate) transmitter. Using commercial model checkers [23], SYNTHLC checks if taint can flow to (the μFSM associated with) any $dst \in \mathbf{dst}$. If so, P is a transponder, and T is recorded as a transmitter (explicit input) along with its label and unsafe operand *op* (§D1).

E. Product Circuit Limitations

In §D2, we note that Eq. 3 can be evaluated with a product circuit or symbolic IFT. SYNTHLC chooses the latter, because we find that the product circuit formulation is vulnerable (in our usage scenario) to *misclassifying* a particular transponder decision as *independent* of some transmitter's operand. We here provide intuition below.

Consider a candidate pair of a transmitter T and its unsafe operand *op*. To establish causality between *op* and a decision made by i_P at decision source $src \in \mathbf{src}^P_{\mathcal{M}}$, *op* must be the only instruction operand in p to be supplied with high data. This requires constraining the operands of all instructions in p, except *op* of i_T, to be equal in both traces. However, once i_T materializes in M, it may create μPATH variability for many instructions *before* i_P reaches src. This variability will soon render the fetch streams of the two traces *out of sync*, i.e., the traces will fetch new instructions in different steps. Such asynchrony makes it difficult to constrain all non-*op* operands in p as required. There are a few plausible solutions to this challenge but all suffer from a soundness issue.

First, by *bounding* the number of instructions in p, it is possible to write a finite set of constraints over the operand values in both traces. This may fail to attribute a transponder decision to *op* if i_T and i_P must be separated by more instructions than the bound permits.

Second, both traces could be constrained to operate in a *lock-step* fashion, such that they always fetch instructions at the same trace steps [32], [33], [90], [91]. Consider the operand-packing optimization from §III-A. If a trace operand-packs after decoding two instructions, it may fetch more instructions in the next cycle, else it may not. Lock-stepping thus enforces that one trace can operand-pack iff the other trace does. Hence, μPATH variability for `ADD`s will be deemed independent of their operands.

Third, a *stuttering circuit* could be used to prevent one trace from fetching instructions when the other lags behind [97]. Suppose i_T is fetched in both traces in step k and i_P is fetched in step $k+1$. Then suppose i_T creates back-pressure in the fetch stream of one trace before i_P has reached its decision source PL $src \in \mathbf{src}^P_{\mathcal{M}}$. The trace that has run ahead will wait for the other to catch up. However, in this time, i_T may progress far enough in its execution such that it can no longer create variability for i_P with respect to src.

In summary, all three approaches may *mis-classify* a particular transponder decision as *independent* of some transmitter's operand, resulting in lack of soundness of synthesized leakage signatures.

[54], [83].