2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) | 978-1-6654-0296-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE43902.2021.00079

Data-Driven Synthesis of Provably Sound Side Channel Analyses

Jingbo Wang, Chungha Sung, Mukund Raghothaman and Chao Wang
University of Southern California

***Abstract*—We propose a *data-driven* method for synthesizing static analyses to detect side-channel information leaks in cryptographic software. Compared to the conventional way of manually crafting such static analyzers, which can be tedious, error prone and suboptimal, our *learning-based* technique is not only automated but also provably sound. Our analyzer consists of a set of type-inference rules learned from the training data, i.e., example code snippets annotated with the ground truth. Internally, we use *syntax-guided synthesis (SyGuS)* to generate new recursive features and *decision tree learning (DTL)* to generate analysis rules based on these features. We guarantee soundness by proving each learned analysis rule via a technique called *query containment checking*. We have implemented our technique in the LLVM compiler and used it to detect *power side channels* in C programs that implement cryptographic protocols. Our results show that, in addition to being automated and provably sound during synthesis, our analyzer can achieve the same empirical accuracy as two state-of-the-art, manually-crafted analyzers while being 300X and 900X faster, respectively.**

I. Introduction

Static analyses are being increasingly used to detect security vulnerabilities such as *side channels* [1]–[4]. However, manually crafting static analyzers to balance between accuracy and efficiency is not an easy task: even for domain experts, it can be labor intensive, error prone, and result in suboptimal implementations. For example, we may be tempted to add expensive analysis rules for specific sanitized patterns without realizing they are rare in target programs. Even if the analysis rules are carefully tuned to a corpus of code initially, they are unresponsive to changing characteristics of the target programs and thus may become suboptimal over time; manually updating them to keep up with new programs would be difficult.

One way to make better accuracy-efficiency trade-offs and to dynamically respond to the distribution of target programs is to use data-driven approaches [5], [6] that automatically synthesize analyses from labeled examples provided by the user. However, checking soundness or compliance with user intent (generalization) has always formed a significant challenge for example-based synthesis techniques [7]–[11]. The lack of soundness guarantees, in particular, hinders the application of such learned analyzers in security-critical applications. While several existing works [12]–[15] try to address this problem, rigorous soundness guarantees have remained elusive.

To overcome this problem, we propose a learning-based method for synthesizing a *provably-sound* static analyzer that detects side channels in cryptographic software, by inferring a *distribution* type for each program variable that indicates if its value is statistically dependent on the secret. The overall flow

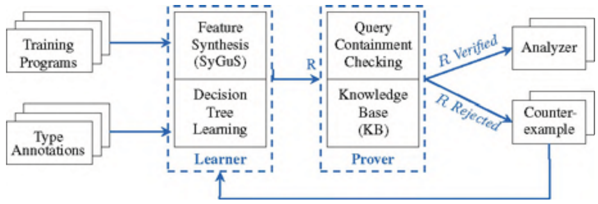


Fig. 1. The overall flow of GPS, our data-driven synthesis method.

of our method, named GPS, is shown in Fig. 1. The input is a set of training data and the output is a learned analyzer. The training data are small programs annotated with the ground truth, e.g., which program variables have leaks.

Internally, GPS consists of a *learner* and a *prover*. The *learner* uses syntax guided synthesis (SyGuS) to generate recursive features and decision tree learning (DTL) to generate type-inference rules based on these features; it returns a set R of Datalog formulas that codify these rules. The *prover* checks the soundness of each learned rule, i.e., it is not only consistent with the training examples but also valid for any unseen programs. This is formulated by solving a *query containment checking* problem, i.e., each rule must be justified by existing proof rules called the knowledge base (KB). Since only proved rules are added to the analyzer, the analyzer is guaranteed to be sound. If a rule cannot be proved, we add a counter-example to prevent the *learner* from generating it again.

We have implemented GPS in LLVM and evaluated it on 568 C programs that implement cryptographic protocols and algorithms [16]–[18]. Together, they have 2,691K lines of C code. We compared our learned analyzer with two state-of-the-art, hand-crafted side-channel analysis tools [1], [2]. Our experiments show that the learned analyzer achieves the same empirical accuracy as the two state-of-the-art tools, while being several orders-of-magnitude faster. Specifically, GPS is, on average, 300× faster than the analyzer from [1] and 900× faster than the analyzer from [2].

To summarize, this paper makes the following contributions:

- We propose the first data-driven method for learning a provably sound static analyzer using syntax guided synthesis (SyGuS) and decision tree learning (DTL).
- We guarantee soundness by formulating and solving a Datalog query containment checking problem.
- We demonstrate the effectiveness of our method for detecting side channels in cryptographic software.

In the remainder of this paper, we begin by presenting the technical background in Section II and our motivating example

978-1-6654-0296-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00079

in Section III. We then describe the *learner* in Section IV and the *prover* in Section V, followed by the experimental results in Section VI. Finally, we survey the related work in Section VII and conclude in Section VIII.

II. Preliminaries

A. Power Side-Channels

Prior works in side-channel security [19]–[21] show that variance in the power consumption of a computing device may leak secret information; for example, when a secret value is stored in a physical register, its number of logical-1 bits may affect the power consumption of the CPU. Such side-channel leaks are typically mitigated by *masking*, e.g., using d random bits $(r_1, \ldots, r_d)$ to split a key bit into $d+1$ secret shares: $key_1 = r_1, \ldots, key_d = r_d$, and $key_{d+1} = r_1 \oplus r_2 \ldots \oplus r_d \oplus key$, where $\oplus$ denotes the logical operation *exclusive or* (**XOR**). Since all $d+1$ shares are uniformly distributed in the $\{0,1\}$, in theory, this *order-d masking* scheme is secure in that any combination of less than d shares cannot reveal the secret, but combining all $d+1$ shares, $key_1 \oplus key_2 \oplus ...key_{d+1} = key$, recovers the secret.

In practice, masking countermeasures must also be implemented properly to avoid de-randomizing any of the secret shares accidentally. Consider $t = t_L \oplus t_R = (r_1 \oplus key) \oplus (r_1 \oplus b) = key \oplus b$. While syntactically dependent on the two randomized values t_L and t_R, t is in fact leaky because, semantically, it does not depend on the random input r_1. In this work, we aim to learn a static analyzer that can soundly prove that *all intermediate variables of a program* that implements masking countermeasures are free of such leaks.

B. Type Systems

Type systems prove to be effective in analyzing power side channels [1], [2], e.g., by certifying that all intermediate variables of a program are *statistically independent* of the secret. Typically, the program inputs are marked as public (`INPUB`), secret (`INKEY`) or random (`INRAND`), and then the types of all other program variables are inferred automatically.

The type of a variable v, denoted `TYPE(v)`, may be `RUD`, `SID`, or `UKD`. Here, `RUD` stands for random uniform distribution, meaning v is either a random bit or being masked by a random bit. `SID` stands for secret independent distribution, meaning v does not depend on the secret. While an `RUD` variable is, by definition, also `SID`, an `SID` variable does not have to be `RUD` (e.g., variables that are syntactically independent of the secret). Finally, `UKD` stands for unknown distribution, or potentially leaky; if the analyzer cannot prove v to be `RUD` or `SID`, then it is assumed to be `UKD`.

Type systems are generally designed to be sound but not necessarily complete. They are sound in that they never miss real leaks. For example, by default, they may safely assume that all variables are `UKD`, unless a variable is specifically elevated to `SID` or `RUD` by an analysis rule. Similarly, they may conservatively classify `SID` variables as `UKD`, or classify `RUD` variables as `SID`, without missing real leaks. In general, the sets of variables that can be marked as the three types form a hierarchy: $S_{\mathtt{RUD}} \subseteq S_{\mathtt{SID}} \subseteq S_{\mathtt{UKD}}$.

C. Relations

A program in static single assignment (SSA) format can be represented as an abstract syntax tree (AST). Static analyzers infer the type of each node x of the program's AST based on various *features* of x. In this work, pre-defined features are represented as *relations*.

- Unary relations $\mathtt{INPUB}(x)$, $\mathtt{INKEY}(x)$, and $\mathtt{INRAND}(x)$ denote the given security level of a program input x, which may be public, secret, or random.
- Unary relations $\mathtt{RUD}(x)$, $\mathtt{SID}(x)$, and $\mathtt{INRAND}(x)$ denote the inferred type of a program variable x, which may be uniformly random, secret independent, or unknown.
- Unary relation $\mathtt{OP}(x)$ denotes the operator type of the AST node x, e.g., $\mathtt{OP}(x) := \mathtt{ANDOR}(x) \mid \mathtt{XOR}(x)$, where $\mathtt{ANDOR}(x)$ means that x's operator type is either *logical and* or *logical or*, and $\mathtt{XOR}(x)$ means that x's operator type is *exclusive or*;
- Binary relations $\mathtt{LC}(x, L)$ and $\mathtt{RC}(x, R)$ indicate that the AST nodes L and R are the left and right operands of x, respectively.
- Binary relation $\mathrm{supp}(x, y)$ indicates that the AST node y is used in the computation of x syntactically, while $\mathrm{dom}(x, y)$ indicates that random program input y is used in the computation of x semantically.

III. Motivation

Consider the program in Fig. 2a, which computes the χ function from Keccak, a family of cryptographic primitives for the SHA-3 standard [22], [23]. It ultimately computes the function `n1 = i1 ⊕ (¬ i2 ∧ i3)`, where $\oplus$ means **XOR**. Unfortunately, a straightforward implementation could potentially leak knowledge of the secret inputs `i1`, `i2` and `i3` if the attacker were able to guess the intermediate results `¬ i2` and `¬ i2 ∧ i3` via the *power* side-channels [24], [25]. The masking countermeasures in the implementation therefore use three additional random bits `r1`, `r2` and `r3` to prevent exposure of the private inputs while still computing the desired function.

A. Problem Setting

Given such a masked program, users want to determine if they succeed in eliminating side-channel vulnerabilities: in particular, if each intermediate result is uniformly distributed (`RUD`) or at least independent of the sensitive inputs (`SID`). The desired static analysis thus associates each variable x (e.g., n_1) with the elements of a three-level abstract domain, `RUD`, `SID` or `UKD`, indicating that x is uniformly distributed (`RUD`), secret independent (`SID`), or unknown (`UKD`) and therefore potentially vulnerable.

The decision tree in Fig. 2b represents the desired static analyzer, which accurately classifies most variables of the training corpus, and is also sound when applied to new programs. Given variable x, the decision tree leverages the features of x—such as the operator type of x ($\mathtt{OP}(\mathtt{x}) := \mathtt{ANDOR}(\mathtt{x}) \mid \mathtt{XOR}(\mathtt{x})$) and the

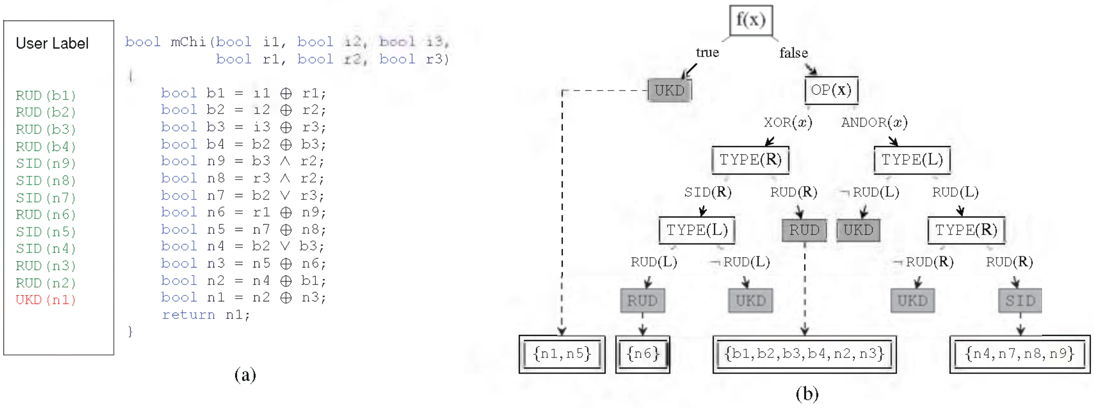


Fig. 2. The program on the left is a perfectly masked χ function from MAC-Keccak. The decision tree on the right represents the static analyzer that the user would like to synthesize. Here, x is a program variable, whose type is being computed; L and R are its left and right operands, respectively, and $f(x)$ is a synthesized feature shown in Fig. 3a (represented by recursive Datalog program).

$$
\begin{array}{lrcl}
R_1: & \mathtt{RUD}(x) & \leftarrow & \mathtt{XOR}(x) \wedge \mathtt{RC}(x,R) \wedge \mathtt{RUD}(R) \wedge \neg f(x) \\
R_2: & f(x) & \leftarrow & \mathtt{LC}(x,L) \wedge \mathtt{RC}(x,R) \wedge g_1(L,r_L) \wedge \\
 & & & g_2(R,r_R) \wedge r_L = r_R \\
R_3: & g_1(r,r) & \leftarrow & \mathtt{INRAND}(r) \\
R_4: & g_1(x,r) & \leftarrow & \mathtt{LC}(x,y) \wedge g_1(y,r) \\
R_5: & g_1(x,r) & \leftarrow & \mathtt{RC}(x,y) \wedge g_1(y,r) \\
R_6: & g_2(r,r) & \leftarrow & \mathtt{INRAND}(r) \\
R_7: & g_2(x,r) & \leftarrow & \mathtt{LC}(x,y) \wedge g_2(y,r) \wedge \mathtt{XOR}(x) \\
R_8: & g_2(x,r) & \leftarrow & \mathtt{RC}(x,y) \wedge g_2(y,r) \wedge \mathtt{XOR}(x)
\end{array}
$$

(a) Excerpt of rules learned by the GPS tool.

$$
\begin{array}{lrcl}
M_1: & \mathtt{RUD}(x) & \leftarrow & \mathtt{XOR}(x) \wedge \mathrm{dom}(x,\mathrm{res}) \wedge \mathrm{res} \neq \emptyset \\
M_2: & \mathrm{supp}(x,x) & \leftarrow & \mathtt{INRAND}(x) \vee \mathtt{INKEY}(x) \vee \mathtt{INPUB}(x) \\
M_3: & \mathrm{supp}(x,\mathrm{res}) & \leftarrow & \mathtt{LC}(x,L) \wedge \mathtt{RC}(x,R) \wedge \mathrm{supp}(L,S_L) \wedge \\
 & & & \mathrm{supp}(R,S_R) \wedge \mathrm{res} = S_L \cup S_R \\
M_4: & \mathrm{dom}(x,x) & \leftarrow & \mathtt{INRAND}(x) \\
M_5: & \mathrm{dom}(x,\emptyset) & \leftarrow & \mathtt{INKEY}(x) \vee \mathtt{INPUB}(x) \\
M_6: & \mathrm{dom}(x,\mathrm{res}) & \leftarrow & \mathtt{XOR}(x) \wedge \mathtt{LC}(x,L) \wedge \mathtt{RC}(x,R) \wedge \\
 & & & \mathrm{dom}(L,S_{DL}) \wedge \mathrm{dom}(R,S_{DR}) \wedge \\
 & & & \mathrm{supp}(L,S_L) \wedge \mathrm{supp}(R,S_R) \wedge \\
 & & & \mathrm{res} = (S_{DL} \cup S_{DR}) \setminus (S_L \cup S_R)
\end{array}
$$

(b) Corresponding expert written rules from SCInfer [2].

Fig. 3. Comparing the rules learned by GPS (Fig. 3a) to manually crafted rules from SCInfer (Fig. 3b). Observe that the learned rules are *sound*, i.e., every variable which potentially leaks information is assigned the distribution type UKD, while still managing to draw non-trivial conclusions such as RUD(b4). The learned rules (R_2—R_8) in Fig. 3a are used to define the new feature $f(x)$ in Fig. 2b.

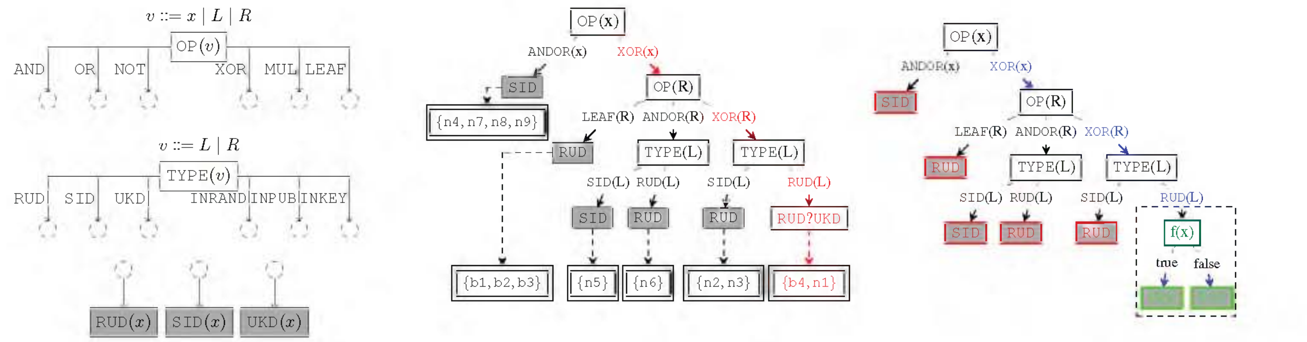
types of x's operands (e.g. TYPE(L), TYPE(R))—and maps x to its corresponding distribution type. The white nodes of Fig. 2b represent pre-defined features, while the grey nodes represent output classes (associated types). Each path from the root to leaf node corresponds to one analysis rule. The set of pre-defined features used in this work is shown in Fig. 4a.

Designing side-channel analyses has been the focus of intense research, see for example [1]–[3], [16], [25]–[28]. Unfortunately, it requires expert knowledge in both computer security and program analysis, and invariably involves delicate trade-offs between accuracy and scalability. Our goal in this work is to assist the analysis designer in automating the development. This problem has also been the subject of exciting research [5], [29]; however, these approaches typically either require computationally intensive deductive synthesis or cannot guarantee soundness and thus produce errors in both directions, including false alarms and missed bugs.

In contrast, GPS combines inductive synthesis from user annotations with logical entailment checking against a more comprehensive, known-to-be-correct set of proof rules that form the knowledge base (KB). It takes as input training programs like the one in Fig. 2a, where the labels correspond to the types of program variables (RUD/SID/UKD for intermediate results and INRAND/INPUB/INKEY for inputs). The users are free to annotate as many or as few of these types as they wish: this affects only the precision of the learned analyzer and not its soundness. Second, GPS also takes as input the knowledge base KB, consisting of proof rules that describe axioms of propositional logic (Fig. 8) and properties of the distribution types (Fig. 10). In return, GPS produces as output a set of Datalog rules which simultaneously achieves high accuracy on the training data and provably sound with respect to KB.

The proof rules for KB were collected from published papers on masking countermeasures [1], [2], [16]. We emphasize that KB is not necessarily an executable static analyzer since repeated application of these proof rules need not necessarily reach a fixpoint and terminate in finite time; Furthermore, even in cases where it does terminate, KB may be computationally expensive and infeasible for application to large programs.

As a concrete example, we compare excerpts of the rules learned by GPS in Fig. 3a to the corresponding rules from SCInfer [2]—a human-written analysis—in Fig. 3b. $\mathtt{LC}(x, L)$ and $\mathtt{RC}(x, R)$ arises in both versions, indicating that L and R are the left and right operands of x respectively. Specifically, in Fig. 3b, $\mathrm{supp}(x, y)$ indicates that y is used in the computation of x syntactically while $\mathrm{dom}(x, y)$ denotes that random variable y is used in the computation of x semantically. Observe the computationally expensive set operations in the human-written version to the simpler rules learned by GPS without loss of soundness or perceptible loss in accuracy. These points are also borne out in our experiments in Table II, where SCInfer takes >45 minutes on some Keccak benchmarks, while our



(a) (b) (c)

Fig. 4. The classifier of Fig. 4b is learned only using the features in Fig. 4a. Because of the limited expressive power of these features, the learned analysis necessarily misclassifies either b4 or n1. Fig. 4c denotes the candidate analyzer produced after one round of feature synthesis. The blue paths corresponds to the rule $\texttt{RUD}(x) \leftarrow \texttt{XOR}(x) \wedge \texttt{XOR}(R) \wedge \texttt{RUD}(L) \wedge \neg f(x) \wedge LC(x, L) \wedge RC(x, R)$. Unfortunately, even though this analysis (Fig. 4c) achieves 100% training accuracy, the leaf nodes highlighted in red correspond to unsound predictions.

learned analysis takes <5 seconds.

GPS consists of two phases: First, it learns a set of type-inference rules—alternatively represented either as Datalog programs or as decision trees—that are consistent with the training data. Second, it proves these rules against the knowledge base. In the next two subsections, we will explain the learning and soundness proving processes respectively.

B. Feature Synthesis and Rule Learning

The learned analyzer associates each node x of a program's abstract syntax tree (AST) with an element of the distribution type $\{\texttt{UKD}(x), \texttt{SID}(x), \texttt{RUD}(x)\}$. We may therefore interpret the analyzer as a decision tree that, by considering various features of an AST node, maps it to a type. With a pre-defined set of features, such as those shown in Fig. 4a, analyzers of this form can be learned with classical decision tree learning (DTL) algorithms. Fig. 4b shows such an analyzer, learned from the labeled program of Fig. 2a.

Unfortunately, the pre-defined features may not be strong enough to distinguish between nodes with different training labels, e.g., b4 and n1 from the training program, which have distinct labels RUD(b4) and UKD(n1), but after being sifted into the node highlighted in red in Fig. 4b, cannot be separated by any of the features from Fig. 4a. To ensure soundness, the learner would be forced to conservatively assign the label $\texttt{UKD}(x)$, which sacrifices the accuracy.

GPS thus includes a *feature* synthesis engine, triggered whenever the learner fails to distinguish between two differently labeled variables. In tandem with recursive feature synthesis, *GPS* overcomes the limited expressiveness of DTL by enriching syntax space to capture more desired patterns. Observe that paths of a decision tree can be represented as Datalog rules, e.g., the red path in Fig. 4b is equivalent to

$$\texttt{UKD}(x) \leftarrow \texttt{XOR}(x) \wedge \texttt{XOR}(R) \wedge \texttt{RUD}(L) \wedge \texttt{LC}(x, L) \wedge \texttt{RC}(x, R).$$

Viewing this in Datalog also allows us to conveniently describe *recursive* features, and reduce *feature synthesis* to an instance of syntax-guided synthesis (SyGuS). Syntactically, the analysis rules corresponding to new features are instances of a pre-defined set of *meta-rules*, and the target specification is to produce a Datalog program for a relation $f(x)$ that has strictly positive information gain for the variables under consideration (see Section IV for details).

In our running example, the synthesizer produces the feature $f(x)$ shown in Fig. 3a, which intuitively indicates that some random input r is used to compute both operands of x. With this new feature, the learner can distinguish between b4 and n1, and produce the rule shown in Fig. 4c, which correctly classifies all variables of the training program. Observe that the rules defining $f(x)$ in Fig. 3a involve a newly introduced predicate $g(x, r)$ and recursive structure that can classify variables based on arbitrarily deep properties of the abstract syntax tree.

	OP(x)	OP(L)	OP(R)	TYPE(L)	TYPE(R)	$f(x)$
CE_1	ANDOR	-1	-1	-1	-1	-1
CE_2	XOR	-1	LEAF	-1	-1	-1
CE_3	XOR	-1	XOR	SID	-1	-1
CE_4	XOR	-1	ANDOR	RUD	-1	-1
CE_5	XOR	-1	ANDOR	SID	-1	-1

Fig. 5. Abstract counter-examples produced during the soundness verification of the candidate analyzer shown in Fig. 4c.

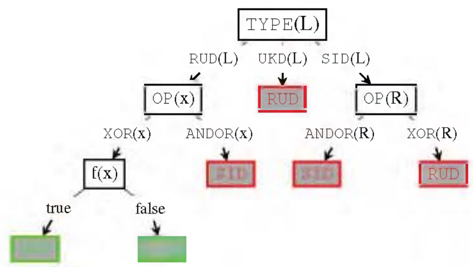


Fig. 6. Candidate analysis learned after one round of feedback from the soundness verifier. The leaves shown in green and red correspond to sound and unsound analysis rules respectively.

C. Proving Soundness of Learned Analysis Rules

While the learned analysis rules are *correct by construction* for the training examples, they may still be unsound when applied to unseen programs. We observe this, for example, in the leaves highlighted in red in Fig. 4c. Thus, GPS tries to confirm their soundness against the domain-specific knowledge base KB. In the context of our running example—confirming soundness means proving that every variable x that is assigned type $\texttt{RUD}(x)$ (resp. $\texttt{SID}(x)$) by the learned analysis rule α is also certified $\texttt{RUD}(x)$ (resp. $\texttt{SID}(x)$) by KB.

We formalize the soundness proof as a Datalog query containment problem, and propose an algorithm based on bounded unrolling and k-induction to check it.

When applied to the candidate analysis of Fig. 4c, the check results in the five counter-examples $CE_1, \ldots, CE_5$ with distribution type $\texttt{UKD}(CE_i)$ shown in Fig. 5. Each counter-example indicates the unsoundness of one path from the root of the decision tree to a classification node. These are *abstract* counter-examples in that they contain *missing features* and consequently do not define concrete ASTs. Thus, each of these *abstract* counter-examples is a set of feature valuations $\pi = \{f_1 \mapsto v_1, f_2 \mapsto v_2, \ldots, f_k \mapsto v_k\}$ that the current candidate analysis misclassifies, and feeding them back to the learner can prohibit subsequent candidate analyses from classifying variables that satisfy π.

With these new constraints from abstract counter-examples, the learner learns the new candidate analysis shown in Fig. 6. This new candidate analysis still has four unsound candidate rules, which results in additional abstract counter-examples when it is subjected to the soundness check. We repeat this back-and-forth between the rule learner and the soundness prover: after 11 iterations and after processing 27 counter-examples in all, GPS learns the rules initially presented in Fig. 2b, all of which have been certified to be sound.

D. Overall Architecture of the GPS System

We summarize the architecture of GPS in Fig. 1. The learner repeatedly applies DTL and SyGuS to learn candidate analyses that correctly classify training samples and are consistent with newly-added abstract counter-examples. Next, the prover checks the soundness of the learned analysis. Each subsequent counter-example is fed back to the learner which restarts the rule learning process on augmented dataset, until either all synthesized rules are sound or a time bound is exhausted.

IV. Learning the Inference Rules

We formally describe the analysis rule learner in Algorithm 1. The input consists of a set of labeled examples, $\mathcal{E}$, and a set of pre-defined features, $\mathcal{F}$. The output $\mathcal{T}$ is a set of type-inference rules consistent with training examples. Each training example $(x, \texttt{TYPE}(x)) \in \mathcal{E}$ consists of an AST node x in a program and its distribution type $\texttt{TYPE}(x)$.

At the top level, the learner uses the standard *decision tree learning (DTL)* algorithm [30] as the baseline. However, if it finds that the current set $\mathcal{F}$ of classification features is insufficient, it invokes a *syntax-guided synthesis (SyGuS)* algorithm to synthesize a new feature f with strictly positive information gain to augment $\mathcal{F}$. This allows the learner to combine the efficiency of techniques that learn decision trees with the expressiveness of syntax guided synthesis; similar ideas have been fruitfully used in other applications of program synthesis, see for example [31].

While the top-level classifier (e.g., Fig. 2b, 4b, 4c and 6) has a bounded number of decision points, the synthesized features (e.g., Fig. 3a) may be recursive. Furthermore, the newly synthesized features f are inducted as first-class citizens of $\mathcal{F}$, and can subsequently be used at any level of the decision tree (see, for example Fig. 2b and 6). Next, we present the DTL and SyGuS subroutines respectively.

Algorithm 1 $\text{DTL}(\mathcal{E}, \mathcal{F})$ — Decision Tree Learning.

Input: Examples, $\mathcal{E} = \{(x_1, \texttt{TYPE}(x_1)), \ldots, (x_n, \texttt{TYPE}(x_n))\}$
Input: Pre-defined features, $\mathcal{F} = \{f_1, f_2, \ldots, f_k\}$
Output: Classifier $\mathcal{T}$ which is consistent with provided examples
1: **if** all examples $(x, \texttt{TYPE}(x)) \in \mathcal{E}$ have the same label $\texttt{TYPE}(x) = t$ **then**
2: **return** $\mathcal{T} = \text{LeafNode}(t)$
3: **end if**
4: **if** $\nexists f \in \mathcal{F}$ such that $H(\mathcal{E} \mid f) < H(\mathcal{E})$ **then**
5: $\mathcal{F} := \mathcal{F} \cup \text{FEATURESYN}(\mathcal{E})$
6: **end if**
7: $\mathcal{T} = \text{DecisionNode}(f^*)$, where $f^* = \arg\min_{f \in \mathcal{F}} H(\mathcal{E} \mid f)$
8: **for** valuation i of feature $f*$ **do**
9: $\mathcal{T}_i = \text{DTL}(\mathcal{E}|_{f^*(x)=i}, \mathcal{F} \setminus \{f^*\})$
10: Add edge from $\mathcal{T}$ to $\mathcal{T}_i$ with label $f^*(x) = i$
11: **end for**
12: **return** $\mathcal{T}$

A. The Decision Tree Learning Algorithm

Recall that our pre-defined features (Fig. 4a) include properties of the AST node, such as $\texttt{OP}(x)$, and properties referring its left and right children, such as $\texttt{OP}(L) \wedge \texttt{LC}(x, L)$. The choice requires some care: having very few features will cause the learning algorithm to fail, while having too many features will increase the risk of overfitting. Our synergistic combination of DTL with SyGuS-based on-demand feature synthesis can be seen as a compromise between these extremes.

$\text{DTL}(\mathcal{E}, \mathcal{F})$ is an *entropy-guided* greedy learner [30], where the entropy and conditional entropy of a set (defined below) are used to measure the diversity of its labels:

$$H(\mathcal{E}) = -\sum_{t \in \texttt{TYPE}} \Pr(\texttt{TYPE}(x) = t) \ \log(\Pr(\texttt{TYPE}(x) = t))$$

$$H(\mathcal{E} \mid f) = \sum_{i \in \text{Range}(f)} H(\mathcal{E} \mid f(x) = i)$$

Algorithm 1 thus divides the set of training examples $\mathcal{E}$ using the feature $f = f^*$ that minimizes the conditional entropy $H(\mathcal{E} \mid f)$ (Lines 7–12), and recursively invokes the learning algorithm on each subset, $\text{DTL}(\mathcal{E}|_{f^*(x)=i}, \mathcal{F} \setminus \{f^*\})$.

Observe that $H(\mathcal{E}) = 0$ if $\Pr(\texttt{TYPE}(x) = t) = 100\%$, meaning *purity* or all examples $x \in \mathcal{E}$ share the same type $\texttt{TYPE}(x) = t$. The difference between $H(\mathcal{E})$ and $H(\mathcal{E} \mid f)$ is also referred to as the *information gain*. If the learner cannot find a feature with strictly positive information gain (Line 4), it will invoke the feature synthesis algorithm on Line 5.

Algorithm 2 FEATURESYN($\mathcal{E}$).

Input: Examples, $\mathcal{E} = \{(x_1, \texttt{TYPE}(x_1)), \ldots, (x_n, \texttt{TYPE}(x_n))\}$
Output: Feature f with positive information gain, or $\perp$ to indicate failure

```
1: Let S be the meta-rules defined in Figure 7, i.e. the hypothesis space
2: for each relation schema r defined in S do
3:     for each subset S of meta-rules corresponding to the schema do
4:         for each choice p_in, q_in, and nested relational predicates do
5:             Let f be the corresponding instantiation of the meta-rules in S
6:             if h(E | f) ≤ h(E) then
7:                 return f
8:             end if
9:         end for
10:    end for
11: end for
12: return ⊥
```

$$R_f = \left\{ \begin{array}{lll} f(x) & \leftarrow & p_{in}(x), \\ f(x) & \leftarrow & q_{in}(x,y), \\ f(x) & \leftarrow & p_{in}(x,y) \wedge q_{in}(x,y), \\ f(x) & \leftarrow & q_{in}(x,y) \wedge f(y), \\ f(x) & \leftarrow & q_{in}(x,y) \wedge p_{in}(x) \wedge f(y) \end{array} \right\}$$

$$R_g = \left\{ \begin{array}{lll} g(x,y) & \leftarrow & q_{in}(x,y), \\ g(x,y) & \leftarrow & p_{in}(x) \wedge q_{in}(x,y), \\ g(x,y) & \leftarrow & q_{in}(x,z) \wedge g(z,y), \\ g(x,y) & \leftarrow & q_{in}(x,z) \wedge p_{in}(x) \wedge g(z,y) \end{array} \right\}$$

$$R_h = \left\{ \begin{array}{lll} h(x) & \leftarrow & f(x) \wedge p_{in}(x) \wedge q_{in}(x,y), \\ h(x) & \leftarrow & g(x,y) \wedge p_{in}(x) \wedge q_{in}(x,y), \\ h(x) & \leftarrow & f(x) \wedge g(x,y) \wedge p_{in}(x) \wedge q_{in}(x,y) \end{array} \right\}$$

$$\begin{array}{rcl} p_{in}(x) & ::= & \texttt{AND}(x) \mid \texttt{OR}(x) \mid \texttt{NOT}(x) \mid \texttt{XOR}(x) \mid \texttt{MUL}(x) \mid \texttt{LEAF}(x) \\ & \mid & \texttt{INRAND}(x) \mid \texttt{INKEY}(x) \mid \texttt{INPUB}(x) \\ & \mid & p_{in} \wedge p_{in} \mid p_{in} \vee p_{in} \mid \neg p_{in} \\ q_{in}(x,y) & ::= & \texttt{LC}(x,y) \mid \texttt{RC}(x,y) \mid x = y \\ & \mid & q_{in}(x,y) \wedge q_{in}(x,y) \mid q_{in}(x,y) \vee q_{in}(x,y) \\ & \mid & \neg q_{in}(x,y) \end{array}$$

Fig. 7. Syntax of the DSL for synthesizing new features.

B. The On-Demand Feature Synthesis Algorithm

We represent newly synthesized features as Datalog programs. Datalog is an increasingly popular medium to express static analyses [32]–[35], and its recursive nature enables the newly learned features to represent arbitrarily deep properties of AST nodes. A Datalog rule is a constraint of the form

$$h(\boldsymbol{x}) \leftarrow b_1(\boldsymbol{y}_1) \wedge b_2(\boldsymbol{y}_2) \wedge \cdots \wedge b_n(\boldsymbol{y}_n), \tag{1}$$

where h, $b_1 \ldots b_n$ are relations with pre-specified arities and schemas, and where $\boldsymbol{x}$, $\boldsymbol{y}_1 \ldots \boldsymbol{y}_n$ are vectors of typed variables. Each rule can be interpreted as a logical implication: if $b_1 \ldots b_n$ are true, then so is h. The semantics of a Datalog program is defined as the *least fixed-point* of rule application [36]: the solver starts with empty output relations, and repeatedly derives new output tuples until no new tuples can be derived.

Program synthesis commonly restricts the space of target concepts and biases the search to speed up computation and improve generalization. One form of bias has been to constrain the syntax: this has been formalized as the SyGuS problem [37] and as meta-rules in inductive logic programming [38], [39]. A meta-rule is construct of this form

$$X_h(\boldsymbol{x}) \leftarrow X_1(\boldsymbol{y}_1) \wedge X_2(\boldsymbol{y}_2) \wedge \cdots \wedge X_n(\boldsymbol{y}_n) \tag{2}$$

Here, X_h, X_1, X_2, $\ldots$, X_n are *relation variables* whose instantiation yields a concrete rule. Fig. 7 shows the meta-rules used in our work. For example, instantiating the meta-rule $f(x) \leftarrow q_{in}(x,y) \wedge p_{in}(x) \wedge f(y)$ with $q_{in}(x,y) = \texttt{RC}(x,y)$ and $p_{in}(x) = \texttt{XOR}(x)$ yields $f(x) \leftarrow \texttt{RC}(x,y) \wedge \texttt{XOR}(x) \wedge f(y)$. There are three variations of the final target relation schema, $f(x)$, $g(x,y)$ and $h(x)$, where x and y denote AST nodes.

We formalize the synthesis problem as that of choosing a relation $R \in \{f(x), g(x,y), h(x)\}$ and finding a subset P_D of its instantiated meta-rules from Fig. 7 such that the resulting Datalog program P_D has strictly positive information gain on the provided training examples $\mathcal{E}$.

Algorithm 2 shows the procedure, which repeatedly instantiates the meta-rules from Fig. 7 and computes their information gain. It successfully terminates when it discovers a feature that can improve classification. Otherwise, it returns failure (upon timeout) and invokes $\text{DTL}(\mathcal{E}, \mathcal{F})$ to conservatively classify the decision tree node as being of type UKD.

Example IV.1. Given $\mathcal{E} = \{(\texttt{b4}, \texttt{RUD}), (\texttt{n1}, \texttt{UKD})\}$ shown in Fig. 2a, the synthesizer may alternatively learn the rules in Equations 3, 4 and 5.

$$\begin{aligned} f(x) &\leftarrow \texttt{INRAND}(x), \\ f(y) &\leftarrow \texttt{LC}(y,x) \wedge f(x), \\ f(y) &\leftarrow \texttt{RC}(y,x) \wedge f(x), \\ \texttt{RUD}(x) &\leftarrow \texttt{XOR}(x) \wedge \texttt{LC}(x,L) \wedge \texttt{RC}(x,R) \wedge \texttt{RUD}(L) \wedge f(R). \end{aligned} \tag{3}$$

$$\begin{aligned} g(x,x) &\leftarrow \texttt{INRAND}(x), \\ g(y,z) &\leftarrow \texttt{LC}(y,x) \wedge g(x,z), \\ g(y,z) &\leftarrow \texttt{RC}(y,x) \wedge g(x,z), \\ h(x) &\leftarrow \texttt{LC}(x,L) \wedge \texttt{RC}(x,R) \wedge g(L,x_L) \wedge g(R,x_R) \wedge x_L = x_R, \\ \texttt{RUD}(x) &\leftarrow \texttt{XOR}(x) \wedge \texttt{RUD}(L) \wedge \texttt{RUD}(R) \wedge \texttt{LC}(x,L) \wedge \texttt{RC}(x,R) \wedge \neg h(x). \end{aligned} \tag{4}$$

$$\begin{aligned} g(x,x) &\leftarrow \texttt{INKEY}(x), \\ g(y,z) &\leftarrow \texttt{LC}(y,x) \wedge g(x,z), \\ g(y,z) &\leftarrow \texttt{RC}(y,x) \wedge g(x,z). \\ h(x) &\leftarrow \texttt{LC}(x,L) \wedge \texttt{RC}(x,R) \wedge g(L,x_L) \wedge g(R,x_R) \wedge x_L = x_R, \\ \texttt{RUD}(x) &\leftarrow \texttt{XOR}(x) \wedge \texttt{RUD}(L) \wedge \texttt{RUD}(R) \wedge \neg h(x). \end{aligned} \tag{5}$$

Since the information gain of Rule 3 applying to $\{\texttt{b4}, \texttt{n1}\}$ is zero, it gets discarded (Line 6 in Algorithm 2). In contrast, the information gains of Rules 4 and 5 are both positive. Rule 4 intuitively requires that both the left and right operands of x are of type RUD, and that they do not share any random inputs in computing $\neg h(x)$. Rule 5 requires that the same secret key be used in the computation of both operands. While Rule 4 is sound when applied to arbitrary programs, Rule 5 is unsound. In the next section, we will present an algorithm that can check the soundness of these learned rules.

V. Proving the Inference Rules

We wish to prove that a learned rule, denoted α, never reaches unsound conclusions when applied to any program, by showing that it can be deduced from a *known-to-be-correct* knowledge base (KB). More specifically, we wish to confirm that every AST node x marked as RUD (or SID) by α can be certified to be RUD (or SID) by KB. When both α and KB are expressed in Datalog, the problem reduces to one of determining query containment, e.g., for every valuation of the input relations, $\texttt{RUD}_\alpha \subseteq \texttt{RUD}_{KB}$ (or $\texttt{SID}_\alpha \subseteq \texttt{SID}_{KB}$).

$b \lor \neg b \equiv true\ (B_1)$ $b \land \neg b \equiv false\ (B_2)$ $\neg\neg b \equiv b\ (B_3)$ $\neg a \lor \neg b \equiv \neg(a \land b)\ (B_4)$

$\neg a \land \neg b \equiv \neg(a \lor b)\ (B_5)$ $b \lor false \equiv b\ (B_6)$ $b \lor true \equiv true\ (B_7)$ $b \land true \equiv b\ (B_8)$

$b \land b \equiv b\ (B_9)$ $b \land false \equiv false\ (B_a)$ $b \lor b \equiv b\ (B_b)$ $a \land (a \lor b) \equiv a\ (B_c)$

$a \lor (a \land b) \equiv a\ (B_d)$ $a \oplus b \equiv (a \land \neg b) \lor (\neg a \land b)\ (B_e)$ $(a \lor b) \lor c \equiv a \lor c \lor b\ (B_f)$

$(a \land b) \land c \equiv a \land c \land b\ (B_{10})$ $a \lor (b \lor c) \equiv a \lor b \lor c\ (B_{11})$ $a \land (b \land c) \equiv a \land b \land c\ (B_{12})$

Fig. 8. Proof rules for propositional logic, to simplify the logic formula and deduce Boolean constants ($true$ and $false$).

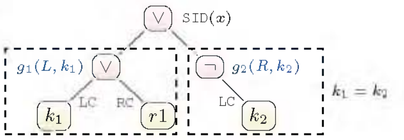


Fig. 9. Example AST from which α is learned.

We will now describe a semi-decision procedure to verify the soundness of the learned rules α, which forms the second phase of the synthesis loop in GPS.

A. Representation of the Learned Rule (α)

Let α be a set of Datalog rules, each of which has a head relation ϕ_α and a body of the following form:

$$\phi_\alpha(\boldsymbol{x}) \leftarrow \phi_1(\boldsymbol{x}_1) \land \phi_2(\boldsymbol{x}_2) \land \cdots \land \phi_n(\boldsymbol{x}_n) \quad (6)$$

It means ϕ_α holds only when all of $\phi_1, \ldots, \phi_n$ hold. Here, ϕ_α may be a distribution type, e.g., $\texttt{SID}(x)$, or a recursive feature $g(x, y)$, e.g., representing that x depends on y.

B. Representation of the Knowledge Base (KB)

Our KB consists of two sets of proof rules, one for propositional logic and the other for distribution types.

Proof Rules for Propositional Logic. Fig. 8 shows the proof rules that represent axioms of propositional logic [40]; they can be used to reduce any valid (resp. invalid) Boolean formula to constant $true$ (resp. $false$). Thus, they are useful in showing results such as $true \lor P$ and $false \land Q$ are secret-independent (SID), for arbitrary logical sentences P and Q.

Consider the example rule α below, where g_1 and g_2 are synthesized features shown as dashed boxes in Fig. 9:

$$\begin{aligned} \texttt{SID}(x) \ &\leftarrow \texttt{OR}(x) \land \texttt{LC}(x, L) \land \texttt{RC}(x, R) \land \texttt{OR}(L) \land \texttt{NOT}(R) \land \\ &\quad g_1(L, k_1) \land g_2(R, k_2) \land \texttt{EQ}(k_1, k_2) \\ g_1(L, k_1) &\leftarrow \texttt{INKEY}(k_1) \land \texttt{INRAND}(r_1) \land \texttt{LC}(L, k_1) \land \texttt{RC}(L, r_1) \\ g_2(R, k_2) &\leftarrow \texttt{INKEY}(k_2) \land \texttt{LC}(R, k_2) \end{aligned}$$

Since $k_1 = k_2$, we transform α into an equivalent logic formula:

$$\texttt{SID}(x) \leftarrow \texttt{EQ}(x, (k_1 \lor r_1) \lor (\neg k_1))$$

Rules $B1$, $B7$ and Bf in Fig. 8 show that $(k_1 \lor r_1) \lor (\neg k_1)$ is always $true$. Thus, x is always $true$. Since x is a constant, we have $\texttt{SID}(x)$, meaning x is secret-independent.

Such SID rules, learned by our method automatically, and yet overlooked by state-of-the-art, hand-crafted analyzers [1], [2], can significantly improve the accuracy of side-channel analysis on many programs.

Proof Rules for Distribution Types. Fig. 10 shows the proof rules that represent properties of the distribution types. They were collected from published papers [2], [16], [24] that focus on verifying masking countermeasures, which also provided the soundness proofs of these rules. For brevity, we omit the detailed explanation. Instead, we use Rule $D_{2.1}$ as an example to illustrate the rationale behind these proof rules.

In Rule $D_{2.1}$, the $dom(x, S)$ relation means that variable x is masked by some input from the set S of random inputs. For example, in $y = x_1 \oplus x_2$, where $x_1 = k \oplus r_1 \oplus r_2$ and $x_2 = b \oplus r_2$, we say that x_2 is masked by r_2, and x_1 is masked by both r_1 and r_2. However, since $r_2 \oplus r_2 = false$, y is masked only by r_1. Thus, $dom(y, \{r_1\})$ holds, but $dom(y, \{r_2\})$ does not hold. In this sense, Rule $D_{2.4}$ defines a *masking set*. For y, it is $S_y = (\{r_1, r_2\} \cup \{r_2\}) \setminus (\{r_1, r_2\} \cap \{r_2\}) = \{r_1\}$, which contains r_1 only. The masking set defined by $D_{2.4}$ is useful in that, as long as the set is not empty, the corresponding variable is guaranteed to be of the RUD type.

C. Proving the Soundness of α Using KB

To prove that for every AST node x marked as $\texttt{RUD}_\alpha(x)$ (resp. $\texttt{SID}_\alpha(x)$) by α, it is also marked as $\texttt{RUD}_{KB}(x)$ (resp. $\texttt{RUD}_{KB}(x)$) by KB, we show that the following relation $Ind(x)$ is empty for any valuation of the input relations:

$$Ind(x) \leftarrow \phi_\alpha(x) \land \neg\phi_{KB}(x) \ , \quad (7)$$

where the relation ϕ may be instantiated to either RUD or SID. In theory, this amounts to proving *query containment*, which is undecidable for Datalog in general [41], [42], but there is a decidable Datalog fragment [41], [43], [44], and our meta-rules in Fig. 7 produce rules in this fragment.

First, we observe that every tuple $t = \phi(x)$ produced by a Datalog program is associated with one or more *derivation trees*. The heights of these derivation trees correspond to the depth of rule inlining at which the program discovers t. In particular, for each inlining depth $k \in \mathbb{N}$, each rule $\phi_h(\boldsymbol{x}_h) \leftarrow \phi_1(\boldsymbol{x}_1) \land \phi_2(\boldsymbol{x}_2) \land \cdots \land \phi_n(\boldsymbol{x}_n)$ is transformed into the rule:

$$\phi^{(k+1)}(\boldsymbol{x}_h) \leftarrow \phi_1^{(k)}(\boldsymbol{x}_1) \land \phi_2^{(k)}(\boldsymbol{x}_2) \land \cdots \land \phi_n^{(k)}(\boldsymbol{x}_n), \quad (8)$$

Our insight is to prove that at each unrolling depth k, we have $\phi_\alpha^{(k)} \subseteq \phi_{KB}^{(k)}$. Thus, we define the relation $Ind^{(k)}$ as follows:

$$Ind^{(k)}(x) \leftarrow \phi_\alpha^{(k)}(x) \land \neg\phi_{KB}^{(k)}(x), \quad (9)$$

and prove the emptiness of $Ind(x)$ by k-induction [45]–[47].

Proposition V.1. *If $Ind^{(k)}(x)$ is an empty relation for each depth $k \in \mathbb{N}$, then $Ind(x)$ is an empty relation.*

$$\frac{\Gamma \vdash x : \texttt{INRAND}}{supp(x, \{x\})}\ (D_{1.1}) \qquad \frac{\Gamma \vdash x : \texttt{INKEY}}{supp(x, \{x\})}\ (D_{1.2}) \qquad \frac{\Gamma \vdash x : \texttt{INPUB}}{supp(x, \{x\})}\ (D_{1.3}) \qquad \frac{\Gamma \vdash x : \texttt{INRAND}}{dom(x, \{x\})}\ (D_{2.1}) \qquad \frac{\Gamma \vdash x : \texttt{INKEY}}{dom(x, \emptyset)}\ (D_{2.2})$$

$$\frac{\Gamma \vdash x, y : \mathsf{v},\ \Gamma \vdash S : \mathsf{Set\ v}, \texttt{RC}(y, x_1) \wedge \texttt{LC}(y, x_2) \wedge supp(x_1, S_1) \wedge supp(x_2, S_2)}{supp(y, S_1 \cup S_2)}\ (D_{1.4}) \qquad \frac{\Gamma \vdash x : \texttt{INPUB}}{dom(x, \emptyset)}\ (D_{2.3})$$

$$\frac{\Gamma \vdash x, y : \mathsf{v}, \Gamma \vdash S : \mathsf{Set\ v}, \texttt{RC}(y, x_1) \wedge \texttt{LC}(y, x_2) \wedge \texttt{XOR}(y) \wedge dom(x_1, S_1) \wedge dom(x_2, S_2)}{dom(x, (S_1 \cup S_2)/(S_1 \cap S_2))}\ (D_{2.4})$$

$$\frac{\Gamma \vdash x : \mathsf{v}, \Gamma \vdash S : \mathsf{Set\ v}, dom(x, S_x) \wedge S_x \neq \emptyset}{\Gamma \vdash x : \texttt{RUD}}\ (D_3) \qquad \frac{\Gamma \vdash x : \texttt{INKEY}, \Gamma \vdash S : \mathsf{Set}\ \texttt{INKEY}}{\Gamma \vdash x :: S : \mathsf{Set}\ \texttt{INKEY}}\ (D_4)$$

$$\frac{\begin{array}{c}\Gamma \vdash x : \mathsf{v},\ \Gamma \vdash S_k : \mathsf{Set}\ \texttt{INKEY},\\ \Gamma \vdash S_d : \mathsf{Set}\ \texttt{RUD},\ \Gamma \vdash S_s : \mathsf{Set\ v},\\ dom(x, S_d) \wedge S_d{=}\emptyset \wedge supp(x, S_s) \wedge S_s \cap S_k{=}\emptyset\end{array}}{\Gamma \vdash x : \texttt{SID}}\ (D_5) \qquad \frac{\begin{array}{c}\Gamma \vdash x_1 : \texttt{SID},\ \Gamma \vdash x_2 : \texttt{RUD},\ \Gamma \vdash S_1, S_2 : \mathsf{Set\ v}\\ \texttt{LC}(y, x_1) \wedge \texttt{RC}(y, x_2) \wedge \texttt{AND}(y)\\ \wedge supp(x_1, S_1) \wedge supp(x_2, S_2) \wedge S_1 \cap S_2{=}\emptyset\end{array}}{\Gamma \vdash y : \texttt{SID}}\ (D_6)$$

$$\frac{\begin{array}{c}\Gamma \vdash x_1 : \texttt{SID},\ \Gamma \vdash x_2 : \texttt{RUD},\ \Gamma \vdash S_1, S_2 : \mathsf{Set\ v}\\ \texttt{LC}(y, x_1) \wedge \texttt{RC}(y, x_2) \wedge \texttt{OR}(y) \wedge\\ supp(x_1, S_1) \wedge supp(x_2, S_2) \wedge S_1 \cap S_2{=}\emptyset\end{array}}{\Gamma \vdash y : \texttt{SID}}\ (D_7) \qquad \frac{\begin{array}{c}\Gamma \vdash x_1 : \texttt{SID},\ \Gamma \vdash x_2 : \texttt{SID},\ \Gamma \vdash S_1, S_2 : \mathsf{Set\ v}\\ \texttt{LC}(y, x_1) \wedge \texttt{RC}(y, x_2) \wedge\\ supp(x_1, S_1) \wedge supp(x_2, S_2) \wedge S_1 \cap S_2{=}\emptyset\end{array}}{\Gamma \vdash y : \texttt{SID}}\ (D_8)$$

$$\frac{\begin{array}{c}\Gamma \vdash x_1 : \texttt{SID},\ \Gamma \vdash x_2 : \texttt{SID},\ \Gamma \vdash S_1 : \mathsf{Set}\ \texttt{RUD},\ \Gamma \vdash S_2 : \mathsf{Set\ v},\\ \texttt{AND}(y) \wedge \texttt{LC}(y, x_1) \wedge \texttt{RC}(y, x_2) \wedge dom(x_1, S_1) \wedge supp(x_2, S_2) \wedge S_1 \cap S_2 \neq \emptyset\end{array}}{\Gamma \vdash y : \texttt{SID}}\ (D_9)$$

$$\frac{\begin{array}{c}\Gamma \vdash x_1 : \texttt{SID},\ \Gamma \vdash x_2 : \texttt{SID},\ \Gamma \vdash S_1 : \mathsf{Set}\ \texttt{RUD},\\ \Gamma \vdash S_2 : \mathsf{Set\ v},\ \texttt{OR}(y) \wedge \texttt{LC}(y, x_1) \wedge \texttt{RC}(y, x_2) \wedge\\ dom(x_1, S_1) \wedge supp(x_2, S_2) \wedge S_1/S_2 \neq \emptyset\end{array}}{\Gamma \vdash y : \texttt{SID}}\ (D_a) \qquad \frac{\begin{array}{c}\Gamma \vdash x_1 : \texttt{RUD},\ \Gamma \vdash x_2 : \texttt{RUD},\ \Gamma \vdash S_1 : \mathsf{Set}\ \texttt{RUD},\\ \Gamma \vdash S_2 : \mathsf{Set\ v},\ \texttt{AND}(y) \wedge \texttt{LC}(y, x_1) \wedge \texttt{RC}(y, x_2) \wedge\\ dom(x_1, S_1) \wedge supp(x_2, S_2) \wedge S_2/S_1 \neq \emptyset\end{array}}{\Gamma \vdash y : \texttt{SID}}\ (D_b)$$

$$\frac{\begin{array}{c}\Gamma \vdash x_1 : \texttt{RUD},\ \Gamma \vdash x_2 : \texttt{RUD},\ \Gamma \vdash S_1 : \mathsf{Set}\ \texttt{RUD},\ \Gamma \vdash S_2 : \mathsf{Set\ v},\\ \texttt{OR}(y) \wedge\ \texttt{LC}(y, x_1) \wedge \texttt{RC}(y, x_2) \wedge dom(x_1, S_1) \wedge supp(x_2, S_2) \wedge S_2/S_1 \neq \emptyset\end{array}}{\Gamma \vdash y : \texttt{SID}}\ (D_c)$$

$$\frac{\Gamma \vdash x : \texttt{RUD}}{\Gamma \vdash x : \texttt{NOUKD}}\ (D_d) \qquad \frac{\Gamma \vdash x : \texttt{SID}}{\Gamma \vdash x : \texttt{NOUKD}}\ (D_e) \qquad \frac{\Gamma \vdash x : v,\ \texttt{NOT}(y) \wedge \texttt{LC}(y, x)}{\Gamma \vdash y : v}\ (D_f) \qquad \frac{\Gamma \vdash x : \mathsf{bool},\ x{=}\mathsf{true}}{\Gamma \vdash x : \texttt{SID}}\ (D_{10})$$

$$\frac{\Gamma \vdash x : \mathsf{bool},\ x{=}\mathsf{false}}{\Gamma \vdash x : \texttt{SID}}\ (D_{11}) \qquad \frac{\Gamma \vdash x : \mathsf{v},\ \Gamma \vdash S_k : \mathsf{Set}\ \texttt{INKEY}, \Gamma \vdash S : \mathsf{Set\ v},\ supp(x, S_s) \wedge S_s \cap S_k{=}\emptyset}{\Gamma \vdash x : \texttt{NOUKD}}\ (D_{12})$$

$$\frac{\begin{array}{c}\Gamma \vdash x_1 : \texttt{RUD},\ \Gamma \vdash x_2 : \texttt{RUD},\ \Gamma \vdash S_1, S_2 : \mathsf{Set}\ \texttt{RUD},\\ \texttt{LC}(y, x_1) \wedge \texttt{RC}(y, x_2) \wedge \texttt{MUL}(y) \wedge\\ (y) \wedge dom(x_1, S_1) \wedge dom(x_2, S_2) \wedge S_2/S_1 \neq \emptyset\end{array}}{\Gamma \vdash y : \texttt{SID}}\ (D_{13}) \qquad \frac{\begin{array}{c}\Gamma \vdash x_1 : \texttt{RUD},\ \Gamma \vdash x_2 : \texttt{SID},\ \Gamma \vdash S_1 : \mathsf{Set}\ \texttt{RUD}\\ \Gamma \vdash\ S_2 : \mathsf{Set\ v},\ \texttt{LC}(y, x_1) \wedge \texttt{RC}(y, x_2)\ \texttt{MUL}(y) \wedge\\ dom(x_1, S_1) \wedge supp(x_2, S_2) \wedge S_1/S_2 \neq \emptyset\end{array}}{\Gamma \vdash y : \texttt{SID}}\ (D_{14})$$

$$\frac{\begin{array}{c}\Gamma \vdash x_1 : \texttt{SID},\ \Gamma \vdash x_2 : \texttt{RUD},\ \Gamma \vdash S_1 : \mathsf{Set}\ \texttt{RUD},\ S_2 : \mathsf{Set\ v},\\ \texttt{LC}(y, x_1) \wedge \texttt{RC}(y, x_2) \wedge \texttt{MUL}(y) \wedge dom(x_1, S_1) \wedge supp(x_2, S_2) \wedge S_2/S_1 \neq \emptyset\end{array}}{\Gamma \vdash y : \texttt{SID}}\ (D_{15})$$

Fig. 10. Proof rules for distribution types, gathered from prior works [2], [16], [24]. Here, v denotes the type of variable x,and is of the following types: UKD, SID and RUD. NOUKD denotes the secure type (either RUD or SID). All the predefined relations in KB are the same as in α.

Observe that unrolling the rules of a Datalog program to any specific depth yields a formula which can be interpreted within propositional logic. For example, unrolling $f(x)$ from Equation 3 at depths 1 and 2 gives us

$$f^{(1)}(x) = \texttt{INRAND}(x), \text{ and}$$
$$f^{(2)}(y) = (\texttt{LC}(y, x) \wedge f^1(x)) \vee (\texttt{RC}(y, x) \wedge f^1(x)).$$

For any specific value of k, we can therefore use an SMT solver to verify the emptiness of $Ind^{(k)}$.

For the induction step, in particular, we ask the SMT solver to check if $Ind^{(k)}$ can be non-empty while the i preceding relations $Ind^{(k-1)}, \ldots, Ind^{(k-i)}$ are assumed to be empty. Here, $\phi_\alpha^{(k)}$ is expressed recursively using $\phi_\alpha^{(k-1)}, \ldots, \phi_\alpha^{(k-i)}$ and induction succeeds if there exists such a value for $i \in \mathbb{N}$.

Let $V^{(k)}$ be free variables introduced by unrolling the rules at depth k. We assert the non-emptiness of $Ind^{(k)}$ below:

$$\Phi^{(k)} = \bigvee_{x \in V^{(k)}} Ind^{(k)}(x). \tag{10}$$

Thus, we formalize the induction step of the proof by constructing the following formula:

$$\Psi^{(k)} = \neg\Phi^{(k-i)} \wedge \cdots \wedge \neg\Phi^{(k-1)} \wedge \Phi^{(k)} \tag{11}$$

Proposition V.2. *If for some $i \in \mathbb{N}$, the relations $Ind^{(1)}, \ldots, Ind^{(i)}$ are all empty (the base case), and the formula $\Psi^{(k)}$ as defined above is unsatisfiable (the induction step), then $Ind^{(k)}$ is empty for all $k \in \mathbb{N}$.*

Starting from $i = 1$, we use the SMT solver to check Proposition V.2 for increasingly larger i until a timeout is reached. If the SMT solver is ever successful in proving the proposition, it follows that the learned rule α is sound.

D. Generating Abstract Counter-Examples

When the proof fails, however, we need to prevent the same rule from being learned again to guarantee progress. Let $\pi = \{f_1 = v_1, f_2 = v_2, \dots, f_k = v_k\}$ be the feature valuation in the failing rule R_π. We then construct the counter-example,

$$CE_\pi = \{f \mapsto v \mid (f, v) \in \pi\} \cup \{f \mapsto -1 \mid f \in \mathcal{F} \setminus \pi\}$$

with label $\texttt{UKD}(CE_\pi)$. Recall that $\mathcal{F}$ is the set of all features currently under consideration. Therefore, the feedback CE_π provided to $\text{DTL}(\mathcal{E}, \mathcal{F})$ is an *abstract* counter-example, with all missing features $f \in \mathcal{F} \setminus \pi$ set to the unknown value -1.

Consider the subsequent iteration of the decision tree learner, $\text{DTL}(\mathcal{E} \cup \{CE_\pi\}, \mathcal{F})$. Observe that whenever it is in a decision context which is also a prefix π_{pre} of the counter-example CE_π, the information gain of each feature $f \in \pi$ is strictly less than that encountered in the previous invocation. Therefore, at some level of the decision tree, it will either choose a different feature, or invoke the feature synthesis algorithm to grow $\mathcal{F}$. By formalizing this argument, we say that:

Proposition V.3. *Given a counter-example CE_π to a learned rule R_π, the subsequent invocation of the learner* $\text{DTL}(\mathcal{E} \cup \{CE_\pi\}, \mathcal{F})$ *is guaranteed to no longer produce R_π.*

Before ending this section, we stress that the *proof rules* in KB should not be confused with *analysis rules* used in the learned analyzer, since they are way more computationally expensive. Consider Rule $D_{1.4}$, whose Datalog encoding size for $supp(x, S)$ would be $|V| \times 2^{|IN|}$. For the benchmark named B19 in Table I, it owns 1250 input variables and thereby causing the exponential explosion with 2^{1250}. The learned rule α, in contrast, is much cheaper since it does not rely on these expensive set (union and intersection) operations.

VI. Experiments

Our experiments were designed to answer the following research questions (RQs):

- RQ1: How effective is our learned analyzer in terms of the analysis speed and accuracy?
- RQ2: How effective is our GPS method for learning inference rules from training data?
- RQ3: How effective is our GPS method for proving the learned inference rules?

We implemented GPS in LLVM 3.6. GPS relies on LLVM to parse the C programs and construct the internal representation (IR). Then, it learns a static analyzer in two steps. The first step, which is SyGuS-guided decision tree learning, is implemented in 4,603 lines of C++ code. The second step, which proves the learned inference rules, is implemented using the Z3 [48] SMT solver. Furthermore, the learned analyzer (for detecting power side channels in cryptographic software) is implemented in LLVM as an optimization (*opt*) pass. We ran all experiments on a computer with 2.9 GHz Intel Core i5 CPU and 8 GB RAM.

TABLE I
STATISTICS OF THE BENCHMARK PROGRAMS IN D_{test}.

Name	LoC	I_{pub}	I_{priv}	I_{rand}	Name	LoC	I_{pub}	I_{priv}	I_{rand}
B1	11	0	2	2	B2	12	0	2	2
B3	12	0	1	2	B4	25	1	1	3
B5	25	1	1	3	B6	32	1	1	3
B7	81	1	1	7	B8	84	1	1	7
B9	104	1	1	7	B10	964	1	16	32
B11	1,130	1	16	32	B12	1,256	0	25	75
B13	2,506	0	25	125	B14	3,764	0	25	175
B15	8,810	0	25	349	B16	13,810	0	25	575
B17	18,858	0	25	775	B18	23,912	0	25	975
B19	30,228	0	25	1,225	B20	34,359	16	16	1,232
B21	79	0	16	16	B22	67	0	8	16
B23	21	0	2	2	B24	23	0	2	2
B25	27	0	1	2	B26	32	0	2	2
B27	40	0	2	3	B28	59	0	3	4
B29	60	0	3	4	B30	66	0	3	4
B31	66	0	3	4	B32	426k	288	288	3205
B33	426k	288	288	3205	B34	426k	288	288	3205
B35	429k	288	288	3205	B36	426k	288	288	3205
B37	442k	288	288	3205					

A. Benchmarks

Our benchmarks are 568 programs with 2,691K lines of C code in total. They implement well-known cryptographic algorithms such as AES and SHA-3. Some of these programs are hardened by countermeasures, such as reordered MAC-Keccak computation [23], masked AES [16], [17], masked S-box calculation [49] and masked multiplication [50], to eliminate power side-channel leaks.

We partition the benchmarks into two sets: D_{train} for GPS, and D_{test} for the learned analyzer. The training set D_{train} consists of 531 small programs gathered from various public sources, including byte-masked AES [51], random reduction of S-box [52], common shares [53], and leak examples [24]. Each benchmark is a pair, consisting of a program AST and its distribution type, i.e, the ground truth annotated by developers. The testing set D_{test} consists of 37 large programs, whose statistics (the number of lines of code and inputs labeled public, private, and random) are shown in Table I. Since these programs are large, it is no longer practical to manually annotate the ground truth; instead, we relies on the results of published tools: a (manually-crafted) static analyzer [1] for B1-B20 and a formal verification tool [2] for B21-B37.

B. Performance and Accuracy of the Learned Analyzer

To demonstrate the advantage of our learned analyzer (answer RQ1), we compared our learned analyzer with the two existing tools [1], [2] on the programs in D_{test}. Only our analyzer can handle all of the 37 programs. Therefore, we compared the results of our analyzer with the tool from [1] on B1-B20, and with the tool from [2] on B21-B37. The results are shown in Table II and Table III, respectively.

In both tables, Columns 1-2 show the benchmark name and the number of AST nodes. Columns 3-6 show the existing tool's analysis time and result, including a breakdown in three types. Similarly, Columns 7-10 show our learned analyzer's time and result. Note that in [1], the UKD/SID/RUD numbers were the number of variables of the LLVM IR, and thus larger than the number of variables in the original programs. To be consistent, we compared with their results in the same manner in Table II.

The results in Table II and Table III show that our learned analyzer is much faster, especially on larger programs such as B20

TABLE II
COMPARING THE LEARNED ANALYZER WITH THE TOOL FROM [1].

Name	# AST	Manually Designed Analyzer [1]				Our Learned Analyzer			
		Time (s)	UKD	SID	RUD	Time (s)	UKD	SID	RUD
B1	7	0.061	4	0	22	**0.001**	4	0	22
B2	6	0.105	**7**	**0**	20	**0.001**	**6**	**1**	20
B3	8	0.099	1	3	31	**0.001**	1	3	31
B4	11	0.208	**6**	12	31	**0.001**	**17**	12	20
B5	11	0.216	**1**	10	29	**0.001**	**11**	10	19
B6	14	0.276	**1**	15	48	**0.001**	**8**	15	41
B7	39	0.213	2	25	151	**0.002**	2	25	151
B8	39	0.147	4	42	249	**0.002**	4	42	249
B9	47	0.266	2	61	153	**0.001**	2	61	153
B10	522	0.550	31	12	2334	**0.008**	31	12	2334
B11	522	0.447	31	0	2334	**0.029**	31	0	2334
B12	426	0.619	52	300	2062	**0.001**	52	300	2062
B13	827	1.102	49	600	4030	**0.006**	49	600	4030
B14	1,228	1.998	49	900	5995	**0.065**	49	900	5995
B15	2,832	16.999	49	2,100	13861	**0.107**	49	2,100	13861
B16	4,436	24.801	49	3,300	21,723	**2.663**	49	3,300	21,723
B17	6,040	59.120	49	4,500	29,587	**1.956**	49	4,500	29,587
B18	7,644	121.000	47	5,700	37,449	**3.258**	47	5,700	37,449
B19	9,649	202.000	49	7200	47,280	**5.381**	49	7200	47,280
B20	13,826	972.000	127	26,330	38,070	**3.650**	127	26,330	38,070

TABLE III
COMPARING THE LEARNED ANALYZER WITH SCINFER [2].

Name	# AST	The SCInfer Verification Tool [2]				Our Learned Analyzer			
		Time (s)	UKD	SID	RUD	Time (s)	UKD	SID	RUD
B21	32	0.390	16	0	16	**0.005**	16	0	16
B22	24	0.570	8	0	16	**0.002**	8	0	16
B23	6	0.010	0	0	6	**0.001**	0	0	6
B24	6	0.060	0	0	6	**0.001**	0	0	6
B25	8	0.250	0	2	6	**0.001**	0	2	6
B26	9	0.160	2	3	4	**0.002**	2	3	4
B27	11	0.260	1	5	5	**0.001**	1	5	5
B28	18	0.290	3	4	11	**0.003**	3	4	11
B29	18	0.230	2	4	11	**0.002**	2	4	12
B30	28	0.340	2	6	20	**0.001**	**8**	0	20
B31	28	0.500	2	7	19	**0.001**	2	7	19
B32	197k	3.800	0	6.4k	190.4k	**3.180**	0	6.4k	190.4k
B33	197k	2,828.000	4.8k	6.4k	185.6k	**3.260**	4.8k	6.4k	185.6k
B34	197k	2,828.000	3.2k	6.4k	187.2k	**3.170**	3.2k	6.4k	187.2k
B35	198k	2,828.000	1.6k	8k	188.8k	**3.140**	**3.2k**	8k	187.2k
B36	197k	2,828.000	4.8k	6.4k	185.6k	**3.150**	4.8k	6.4k	185.6k
B37	205k	2,828.000	17.6k	1.6k	185.6k	**3.820**	17.6k	1.6k	185.6k

(3.6 seconds versus 16 minutes). The reason why our analyzer is faster is because the manually-crafted analyses [1], [2] rely on evaluating set-relations (e.g. difference and intersection of sets of random variables), whereas our DSL syntax is designed without set operations to infer the same types, thus leading to faster analyses. Although in general the set operation-based algorithm is more accurate, it has excessive computational overhead. Moreover, it does not always improve precision in practice. Furthermore, the method in [2] uses an SMT solver-based model counting technique to infer leak-free variables, which is significantly more expensive than type inference.

As shown in Table II and Table III, by learning inference rules from data, we can achieve almost the same accuracy as manually-crafted analysis [1], [2] while avoiding the huge overhead. Given the same definitions of distribution types (`UKD`, `SID` and `RUD`), both our learned rules and manually-crafted analysis rules [1], [2] can infer the non-leaky patterns, thus recognizing the variable types correctly under most benchmarks in Table II and Table III, except for B4-B6 and B30, where set operations are required to prove the leak-freedom of some variables. Recall that losing accuracy here indicates that our learned rules infer the types more conservatively, without losing soundness. Nevertheless, our analyzer also increased accuracy in some other cases (e.g., B2), due to its deeper constant propagation (which led to the proof of more `SID` variables) while the existing tool [1] failed to do so, and conservatively marked them as `UKD` variables.

C. Effectiveness of Rule Induction and Soundness Verification

To answer RQ2 and RQ3, we collected statistics while applying GPS to the 531 small programs in D_{test}, as shown in Table IV. In total, GPS took 30 iterations to complete the entire learning process. Column 1 shows the iteration number and Column 2 shows the time taken by the *learner* and the *prover* together. Columns 3-6 show the number of inference rules learned during each iteration, together with their types (`UKD`, `SID`, and `RUD`). Similarly, Columns 7-10 show the number of verified inference rules and their types.

The next two columns show the following statistics: (1) the size of the learned decision tree (# Tree_{learn}) in terms of the number of decision nodes; (2) the number of counter-examples (CEX) added by the prover (# AST_{CEX}), which are added to the 531 original training programs before the next iteration starts. The last column shows the number of features generated by SyGuS; these features are also added to the original feature set and then used by the learner during the next iteration.

Results in Table IV demonstrate the efficiency of both the learner and the prover. Within the *learner*, the number of rules produced in each iteration remains modest (8 on average), indicating it has successfully avoided overfitting. This is because the SyGuS solver is biased toward producing small features which, by *Occam's razor*, are likely to generalize well. Furthermore, any learned analysis rules have to pass the soundness check, and this provides additional assurance against overfitting to the training data. The *prover* either quickly verifies a rule, or quickly drops it after adding a counter-example to prevent it from being learned again. In early iterations, about half of all learned rules can be proved, but as more counter-examples are added, the quality of the learned rules improves, and thus the percentage of proved rules also increases.

D. Threats to Validity

Our experimental evaluation focused on cryptographic software, which is structurally simple and, unlike general-purpose software, does not exercise complicated language constructs. It is an interesting direction of future work to extend our techniques to these more general classes of software code.

A notable limitation in our work is the assumption of the knowledge base (KB). While KB is readily available for our application (side-channel analysis), for other applications, it might be non-trivial to construct. Furthermore, an incorrect KB might compromise the soundness of the learned rules, although in this work, we have carefully mitigated this threat by curating the proof rules from previous papers [2], [16], [24] that have themselves formally verified the validity of these proof rules.

VII. RELATED WORK

Generating Analyzers from Examples. While there are prior works on learning static analyzers [5], [54], they do not guarantee soundness. For example, the analyzer learned by Bielik et al. [5] is sound with respect to programs in the

TABLE IV

DECISION TREE LEARNING WITH FEATURE SYNTHESIS (DIFFERENT ITERATIONS WITH #AST = 531).

Iteration	Time (s)	# Rules Learned				# Rules Verified				# $Tree_{learn}$	# AST_{CEX}	# $Feature_{syn}$
		Total	*UKD*	*SID*	*RUD*	Total	*UKD*	*SID*	*RUD*			
1	1.316	9	2	2	5	5	2	1	2	23	4	5
2	0.775	8	2	2	4	4	2	1	1	17	9	7
3	1.115	8	2	2	4	5	2	2	1	24	13	9
4	0.511	8	2	2	4	5	2	1	2	18	18	10
5	0.513	8	2	2	4	7	2	2	3	27	21	11
6	0.537	8	2	2	4	6	2	2	2	24	25	12
7	0.510	8	2	2	4	6	2	2	2	26	29	13
8	0.512	8	2	2	4	6	2	2	2	28	33	14
9	0.511	8	2	2	4	6	2	2	2	30	37	15
10	0.524	8	2	2	4	5	2	2	1	32	41	16
11	0.546	8	2	2	4	4	2	2	0	34	45	17
12	0.556	8	2	2	4	4	2	2	0	36	49	18
13	0.550	8	2	2	4	5	2	2	1	38	53	19
14	0.540	8	2	2	4	6	2	2	2	40	57	20
15	0.542	8	2	2	4	4	2	2	0	42	61	21
16	0.552	8	2	2	4	6	2	2	2	44	65	22
17	0.577	8	2	2	4	5	2	2	1	46	69	23
18	0.598	8	2	2	4	6	2	2	2	48	73	24
19	0.571	8	2	2	4	6	2	2	2	50	77	25
20	0.673	8	2	2	4	5	1	2	2	52	82	26
21	0.526	8	2	2	4	3	1	2	0	54	87	27
22	0.525	8	3	2	3	6	3	2	1	35	91	27
23	0.697	9	3	2	4	7	2	2	3	37	93	27
24	0.700	9	3	2	4	8	2	2	4	38	95	28
25	0.691	7	2	2	3	6	1	2	3	36	97	29
26	0.707	7	2	2	3	6	1	2	3	37	99	30
27	0.716	7	2	2	3	6	1	2	3	38	101	31
28	0.540	7	2	2	3	6	1	2	3	39	102	32
29	0.534	7	2	2	3	6	1	2	3	39	103	32
30	0.528	7	2	2	3	7	2	2	3	39	104	32
TOTAL	18.693	237	63	60	114	167	54	57	56	1071	1833	622

training set, not all programs written in the same programming language (JavaScript). They also need to manually modify the training programs to generate counter-examples, while our method generates counter-examples automatically.

Formal Specifications. There are also works on synthesizing static analyzers from formal specifications, e.g., proof rules or second-order logic formulas [29], [55], [56] as opposed to training data. However, they restrict the logic used to write the specification, and as a result, may not be expressive enough to synthesize practical analyzers. Users are also expected to write correct specifications, which is a non-trivial task. In addition, they cannot exploit the information provided by data.

Learning-based Techniques. There are several prior techniques using machine learning to conduct static program analyses [57]–[60]. Such techniques focus on finding a suitable program-to-feature embedding. However, they require the user to perform feature engineering, which is known to be laborious. Some of these techniques [58], [61]–[63] do not take advantage of new features that may be learned from data; instead, they build classifiers based solely on existing features. In contrast, our method not only learn new analysis rules from data, but also use SyGuS to synthesize new features automatically.

Optimizing an Analyzer. It is possible to optimize an existing static analyzer [57], [64]–[68], which can be achieved by adjusting the level of abstraction [64], [65], [69], learn heuristics and parameters [66], make soundness-accuracy trade-offs [67], or select sound transformers [68]. However, such techniques fundamentally differ from our method because they assume the analyzer is already given, and focus on optimizing its performance, whereas we focus on synthesizing a new analyzer.

Syntax-Guided Synthesis. Since we automatically generate new features, our method is related to the large and growing body of work on SyGuS. While SyGuS has been used in various applications [70]–[80], none of them aims to synthesize a provably sound static analyzer from data. While some of these existing techniques can synthesize Datalog rules [39], [81], [82], the focus has been on efficiency, e.g., pruning the search space based on syntactic structures, instead of guaranteeing the soundness of the analyzer.

Power Side-Channel Analysis. In this work, we use power side-channel analysis as the application to evaluate our method. In this sense, it is related to the body of work on side-channel leak detection [2]–[4], [83]–[85] as well as mitigation [1], [24], [28], [86]–[88]. While static analysis engines used in these existing works are all hand-crafted by domain experts, our method aims to synthesize the static analysis from data automatically.

VIII. CONCLUSIONS

We have presented a data-driven method for learning a *provably sound* static analyzer to detect power side channels in cryptographic software. It relies on SyGuS to generate features and DTL to generate analysis rules based on the synthesized features. It verifies the soundness of these learned analysis rules by solving a query containment checking problem using an SMT solver. We have evaluated our method on C programs that implement well-known cryptographic protocols and algorithms. Our experimental results show that the learning algorithm is efficient and the learned analyzer can achieve the same empirical accuracy as state-of-the-art analysis tools while being several orders-of-magnitudes faster.

ACKNOWLEDGMENTS

This research was supported in part by the U.S. National Science Foundation (NSF) under grant CNS-1617203 and Office of Naval Research (ONR) under grant N00014-17-1-2896. We thank the anonymous reviewers for their helpful feedback.

REFERENCES

[1] J. Wang, C. Sung, and C. Wang, "Mitigating power side channels during compilation," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 590–601.

[2] J. Zhang, P. Gao, F. Song, and C. Wang, "SC Infer: refinement-based verification of software countermeasures against side-channel attacks," in *International Conference on Computer Aided Verification*, 2018, pp. 157–177.

[3] J. Chen, Y. Feng, and I. Dillig, "Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic," in *ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 875–890.

[4] T. Brennan, S. Saha, T. Bultan, and C. S. Păsăreanu, "Symbolic path cost analysis for side-channel detection," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 27–37.

[5] P. Bielik, V. Raychev, and M. Vechev, "Learning a static analyzer from data," in *International Conference on Computer Aided Verification*, 2017, pp. 233–253.

[6] C. Mendis, C. Yang, Y. Pu, S. Amarasinghe, and M. Carbin, "Compiler auto-vectorization with imitation learning," in *Advances in Neural Information Processing Systems*, 2019, pp. 14 598–14 609.

[7] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Communications of the ACM*, vol. 55, no. 8, pp. 97–105, 2012.

[8] A. Leung, J. Sarracino, and S. Lerner, "Interactive parser synthesis by example," *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 565–574, 2015.

[9] R. Singh and S. Gulwani, "Synthesizing number transformations from input-output examples," in *International Conference on Computer Aided Verification*, 2012, pp. 634–651.

[10] C. Smith and A. Albarghouthi, "Mapreduce program synthesis," *Acm Sigplan Notices*, vol. 51, no. 6, pp. 326–340, 2016.

[11] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 404–415.

[12] S. An, R. Singh, S. Misailovic, and R. Samanta, "Augmented example-based synthesis using relational perturbation properties," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–24, 2019.

[13] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani, "User interaction models for disambiguation in programming by example," in *ACM Symposium on User Interface Software & Technology*, 2015, pp. 291–301.

[14] R. Singh and S. Gulwani, "Predicting a correct program in programming by example," in *International Conference on Computer Aided Verification*, 2015, pp. 398–414.

[15] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli, "Robustfill: Neural program learning under noisy i/o," in *International Conference on Machine Learning*, 2017, pp. 990–998.

[16] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, and P.-Y. Strub, "Verified proofs of higher-order masking," in *International Conference on the Theory and Applications of Cryptographic Techniques*, 2015, pp. 457–485.

[17] J. Blömer, J. Guajardo, and V. Krummel, "Provably secure masking of AES," in *International workshop on selected areas in cryptography*, 2004, pp. 69–83.

[18] A. G. Bayrak, F. Regazzoni, D. Novo, and P. Ienne, "Sleuth: Automated verification of software power analysis countermeasures," in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2013, pp. 293–310.

[19] Y. Ishai, A. Sahai, and D. Wagner, "Private circuits: Securing hardware against probing attacks," in *Annual International Cryptology Conference*, 2003, pp. 463–481.

[20] J. Balasch, B. Gierlichs, V. Grosso, O. Reparaz, and F.-X. Standaert, "On the cost of lazy engineering for masked software implementations," in *International Conference on Smart Card Research and Advanced Applications*, 2014, pp. 64–81.

[21] J.-S. Coron, C. Giraud, E. Prouff, S. Renner, M. Rivain, and P. K. Vadnala, "Conversion of security proofs from one leakage model to another: A new issue," in *International Workshop on Constructive Side-Channel Analysis and Secure Design*, 2012, pp. 69–81.

[22] NIST, "NIST selects winner of the Secure Hash Algorithm (SHA-3) competition," *https://www.nist.gov/news-events/news/2012/10/nist-selects-winner-secure-hash-algorithm-sha-3-competition*, 2012.

[23] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, "Keccak implementation overview," *URL: http://keccak. noekeon. org/Keccak-implementation-3.2. pdf*, 2012.

[24] H. Eldib and C. Wang, "Synthesis of masking countermeasures against side channel attacks," in *International Conference on Computer Aided Verification*, 2014, pp. 114–130.

[25] G. Barthe, S. Belaïd, P.-A. Fouque, and B. Grégoire, "maskverif: a formal tool for analyzing software and hardware masked implementations." *IACR Cryptology ePrint Archive*, vol. 2018, p. 562, 2018.

[26] C. Wang and P. Schaumont, "Security by compilation: an automated approach to comprehensive side-channel resistance," *ACM SIGLOG News*, vol. 4, no. 2, pp. 76–89, 2017.

[27] G. Barthe, S. Belaïd, F. Dupressoir, P.-A. Fouque, B. Grégoire, P.-Y. Strub, and R. Zucchini, "Strong non-interference and type-directed higher-order masking," in *ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 116–129.

[28] S. Tizpaz-Niari, P. Černý, and A. Trivedi, "Quantitative mitigation of timing side channels," in *International Conference on Computer Aided Verification*, 2019, pp. 140–160.

[29] C. David, D. Kroening, and M. Lewis, "Using program synthesis for program analysis," in *Logic for programming, artificial intelligence, and reasoning*, 2015, pp. 483–498.

[30] L. Rokach and O. Maimon, "Top-down induction of decision trees classifiers-a survey," *Transactions on Systems, Man, and Cybernetics*, vol. 35, no. 4, pp. 476–487, 2005.

[31] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2017, pp. 319–336.

[32] T. W. Reps, *Demand Interprocedural Program Analysis Using Logic Databases*, 1995, pp. 163–196.

[33] J. Whaley and M. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004, pp. 131–144.

[34] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *International Conference on Computer Aided Verification*, 2016, pp. 422–430.

[35] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, 2009, p. 243–262.

[36] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases: The Logical Level*. Pearson, 1994.

[37] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *Formal Methods in Computer-Aided Design*, 2013, pp. 1–8.

[38] S. Muggleton, D. Lin, and A. Tamaddoni-Nezhad, "Meta-interpretive learning of higher-order dyadic Datalog: Predicate invention revisited," *Machine Learning*, vol. 100, no. 1, pp. 49–73, 2015.

[39] X. Si, W. Lee, R. Zhang, A. Albarghouthi, P. Koutris, and M. Naik, "Syntax-guided synthesis of datalog programs," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 515–527.

[40] S. Russell and P. Norvig, "Artificial intelligence: a modern approach," 2002.

[41] D. Calvanese, G. De Giacomo, and M. Y. Vardi, "Decidable containment of recursive queries," *Theoretical Computer Science*, vol. 336, no. 1, pp. 33–56, 2005.

[42] D. Calvanese, G. De Giacomo, and M. Lenzerini, "On the decidability of query containment under constraints," in *PODS*, vol. 98, 1998, pp. 149–158.

[43] D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi, "Reasoning on regular path queries," *ACM SIGMOD Record*, vol. 32, no. 4, pp. 83–92, 2003.

[44] P. Barceló, M. Romero, and M. Y. Vardi, "Does query evaluation tractability help query containment?" in *ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2014, pp. 188–199.

[45] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *International conference on formal methods in computer-aided design*, 2000, pp. 127–144.
[46] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer, "Software verification using k-induction," in *International Static Analysis Symposium*, 2011, pp. 351–368.
[47] M. Y. Gadelha, H. I. Ismail, and L. C. Cordeiro, "Handling loops in bounded model checking of c programs via k-induction," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 1, pp. 97–114, 2017.
[48] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340.
[49] J.-S. Coron, E. Prouff, M. Rivain, and T. Roche, "Higher-order side channel security and mask refreshing," in *International Workshop on Fast Software Encryption*, 2013, pp. 410–424.
[50] M. Rivain and E. Prouff, "Provably secure higher-order masking of AES," in *International Workshop on Cryptographic Hardware and Embedded Systems*, 2010, pp. 413–427.
[51] Y. Yao, M. Yang, C. Patrick, B. Yuce, and P. Schaumont, "Fault-assisted side-channel analysis of masked implementations," in *International Symposium on Hardware Oriented Security and Trust*, 2018, pp. 57–64.
[52] R. Zhang, S. Qiu, and Y. Zhou, "Further improving efficiency of higher order masking schemes by decreasing randomness complexity," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 11, pp. 2590–2598, 2017.
[53] J.-S. Coron, A. Greuet, E. Prouff, and R. Zeitoun, "Faster evaluation of sboxes via common shares," in *International Conference on Cryptographic Hardware and Embedded Systems*, 2016, pp. 498–514.
[54] M. Zaheer, J.-B. Tristan, M. L. Wick, and G. L. Steele Jr, "Learning a static analyzer: A case study on a toy language," 2016.
[55] S. Grebenshchikov, N. P. Lopes, C. Popeea, and A. Rybalchenko, "Synthesizing software verifiers from proof rules," in *ACM SIGPLAN Notices*, vol. 47, no. 6, 2012, pp. 405–416.
[56] G. Chen, Y. Wang, M. Zhou, and J. Sun, "VFQL: combinational static analysis as query language," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 378–381.
[57] O. Katz, R. El-Yaniv, and E. Yahav, "Estimating types in binaries using predictive modeling," in *ACM SIGPLAN Notices*, vol. 51, no. 1, 2016, pp. 313–326.
[58] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from big code," in *ACM SIGPLAN Notices*, vol. 50, no. 1, 2015, pp. 111–124.
[59] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin, "Aletheia: Improving the usability of static security analysis," in *ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 762–774.
[60] T. Gvero and V. Kuncak, "Synthesizing java expressions from free-form queries," in *Acm Sigplan Notices*, vol. 50, no. 10, 2015, pp. 416–432.
[61] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *Joint Meeting on Foundations of Software Engineering*, 2015, pp. 462–473.
[62] K. Heo, H. Oh, H. Yang, and K. Yi, "Adaptive static analysis via learning with bayesian optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 40, no. 4, p. 14, 2018.
[63] K. Heo, H. Oh, and H. Yang, "Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis," in *International Static Analysis Symposium*, 2016, pp. 237–256.
[64] R. Grigore and H. Yang, "Abstraction refinement guided by a learnt probabilistic model," in *ACM SIGPLAN Notices*, vol. 51, no. 1, 2016, pp. 485–498.
[65] K. Heo, H. Oh, and H. Yang, "Resource-aware program analysis via online abstraction coarsening," in *International Conference on Software Engineering*, 2019, pp. 94–104.
[66] H. Oh, H. Yang, and K. Yi, "Learning a strategy for adapting a program analysis via bayesian optimisation," in *ACM SIGPLAN Notices*, vol. 50, no. 10, 2015, pp. 572–588.
[67] K. Heo, H. Oh, and K. Yi, "Machine-learning-guided selectively unsound static analysis," in *International Conference on Software Engineering*, 2017, pp. 519–529.
[68] G. Singh, M. Püschel, and M. Vechev, "Fast numerical program analysis with reinforcement learning," in *International Conference on Computer Aided Verification*, 2018, pp. 211–229.
[69] C. Wang, Z. Yang, A. Gupta, and F. Ivančić, "Using counterexamples for improving the precision of reachability computation with polyhedra," in *International Conference on Computer Aided Verification*, 2007, pp. 352–265.
[70] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *International Conference on Software Engineering*, 2017, pp. 404–415.
[71] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *International Conference on Software Engineering*, 2016, pp. 691–701.
[72] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," in *ACM SIGPLAN Notices*, vol. 52, no. 6, 2017, pp. 422–436.
[73] N. Polikarpova, I. Kuraj, and A. Solar-Lezama, "Program synthesis from polymorphic refinement types," in *ACM SIGPLAN Notices*, vol. 51, no. 6, 2016, pp. 522–538.
[74] A. Abate, I. Bessa, D. Cattaruzza, L. Cordeiro, C. David, P. Kesseli, D. Kroening, and E. Polgreen, "Automated formal synthesis of digital controllers for state-space physical plants," in *International Conference on Computer Aided Verification*, 2017, pp. 462–482.
[75] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: automated data-driven synthesis of repairs for static analysis violations," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 613–624.
[76] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *International Conference on Software Engineering*, 2017, pp. 416–426.
[77] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Joint Meeting on Foundations of Software Engineering*, 2017, pp. 593–604.
[78] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *International Conference on Software Engineering*, 2018, pp. 12–23.
[79] T. Knoth, D. Wang, N. Polikarpova, and J. Hoffmann, "Resource-guided program synthesis," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 253–268.
[80] P. Li, P. Zhang, L.-N. Pouchet, and J. Cong, "Resource-aware throughput optimization for high-level synthesis," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 200–209.
[81] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang, "On abstraction refinement for program analyses in datalog," in *ACM SIGPLAN Notices*, vol. 49, no. 6, 2014, pp. 239–248.
[82] X. Si, M. Raghothaman, K. Heo, and M. Naik, "Synthesizing datalog programs using numerical relaxation," *arXiv preprint arXiv:1906.00163*, 2019.
[83] L. Bang, A. Aydin, Q.-S. Phan, C. S. Păsăreanu, and T. Bultan, "String analysis for side channels with segmented oracles," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 193–204.
[84] S. Guo, M. Wu, and C. Wang, "Adversarial symbolic execution for detecting concurrency-related cache timing leaks," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 377–388.
[85] C. Sung, B. Paulsen, and C. Wang, "CANAL: a cache timing analysis framework via LLVM transformation," in *IEEE/ACM International Conference On Automated Software Engineering*, 2018, pp. 904–907.
[86] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "FaCT: a dsl for timing-sensitive computation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 174–189.
[87] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 15–26.
[88] B. Paulsen, C. Sung, P. A. H. Peterson, and C. Wang, "Debreach: Mitigating compression side channels via static analysis and transformation," in *IEEE/ACM International Conference On Automated Software Engineering*, 2019, pp. 899–911.