Locating Errors in Faulty Formulas

SAMPATH KANNAN and KEVIN T. TIAN, University of Pennsylvania, USA

Given a drawing of a read-once formula (called the blueprint), and a blackbox implementation with the same topology as the blueprint that purports to compute the formula, can we tell if it does? Under a fault model, where the only faults in the implementation are gates that complement their outputs, we show that there is an efficient algorithm that makes a linear number of probes to the blackbox implementation and determines if the blueprint and implementation are identical. We also show a matching lower bound. We further ask whether we can diagnose where the faults are, using blackbox testing. We prove that if the implementation has a property called *polynomial balance*, then it is possible to do this efficiently. To complement this result, we show that even if the *blueprint* is polynomially balanced and there are only logarithmically many errors in the implementation, the implementation could be unbalanced and the diagnosis problem provably requires super-polynomially many tests. We point out that this problem is one instance of a general class of problems of learning deviations from a blueprint, which we call *conformance learning*. Conformance learning seems worthy of further investigation in a broader context.

CCS Concepts: • **Theory of computation** → *Graph algorithms analysis*; • **Hardware** → *Error detection and error correction*;

Additional Key Words and Phrases: Fault-tolerant computing, fault diagnosis, learning

ACM Reference format:
Sampath Kannan and Kevin T. Tian. 2019. Locating Errors in Faulty Formulas. *ACM Trans. Algorithms* 15, 3, Article 34 (May 2019), 13 pages.
https://doi.org/10.1145/3313776

1 INTRODUCTION

The problem of reliably computing functions using circuits with unreliable components was first studied by von Neumann [16]. Since then, a number of researchers have considered this problem and variants (for example, References [2, 4, 6, 8, 10–12]). Generally all these papers assume a probabilistic failure model, where each gate fails independently with probability p. In von Neumann's model a failing gate produces an output that is the complement of the correct output. Since then, other models have been considered. The *stuck-at* model assumes that the output of failing gates is stuck at one of the constants—0 or 1 [7]; another model assumes that the failing gate produces an output that is always one of its inputs. There are models where failures do not cancel—the failure of one gate is enough to ensure that the circuit produces the wrong answer. In all these models, the goal has been to find upper bounds on the probability of failure of a gate such that the circuit computes the right answer with probability strictly greater than (the trivial value of) 1/2.

Research supported in part by NSF Grant 1547360.
Authors' addresses: S. Kannan and K. T. Tian, University of Pennsylvania, Department of Computer and Information Science, Levine Hall, 3330 Walnut St. Philadelphia, PA, 19104; emails: {kannan, ktian}@cis.upenn.edu.

1549-6325/2019/05-ART34 $15.00
https://doi.org/10.1145/3313776

The question we ask in this article is different. We don't seek to *design* fault-tolerant formulas, but instead to *diagnose* faults in a given formula. We will draw from von Neumann's model—i.e., we assume that we are given a paper drawing of the formula (called the *blueprint*) and an implementation with the same topology. When gates in the implementation fail, they do so by complementing their output. However, an adversary chooses which gates are faulty. We are given only blackbox access to the implementation. We can provide it with any n-bit Boolean input (called a probe) and observe the 1-bit output it produces. We seek necessary and sufficient conditions on the formula that allow us to determine if the blueprint and implementation are the same, and if not, identify the faulty gates using only polynomially many probes.

Formally, we are given a *blueprint* read-once formula $f(x_1, x_2, \ldots, x_n)$ with n inputs as a tree structure with the leaves labeled by the variables. We are also given an *implementation* $\hat{f}$ as a blackbox, where we know that $\hat{f}$ has the same topology as f, but in which some of the gates produce the complement of the output they should produce. While the restriction to read-once formulas is indeed a serious limitation, it is already challenging enough that there is a rich literature focusing on read-once formulas; for example, the works cited near the end of the introduction in the line of von Neumann's work and papers in learning theory, such as References [1, 9, 14]. We show the following results.

- $O(n)$ probes are sufficient to determine whether $\hat{f} \equiv f$.
- If for any partial setting of the inputs we can find two settings of the remaining inputs (if they exist) that cause the implementation to evaluate to 0 and 1, respectively, then we can find all the faults in the implementation. This is our main diagnosis tool.
- Given this theorem, it is natural to focus on the *balance* of a gate—the lesser of the two probabilities of the gate outputting 0 and outputting 1, if each input is chosen uniformly at random. We show that if every gate in the *implementation* has at least inverse polynomial balance, then we can find the faulty gates with polynomially many probes.

The problem we consider could be thought of as one in a new framework for machine learning, which we call *conformance learning*. In conformance learning, a blueprint concept, chosen out of some concept class, is given. The target concept (the implementation) is a "local" variant of the blueprint concept, defined by defining a set of allowed edit operations. Any of the formalisms used for machine learning could be modified to define conformance learning problems. For example, in PAC-conformance learning, following PAC learning [15], we are given labeled examples drawn from an unknown distribution and want to produce a concept in the local neighborhood of the blueprint concept that with high probability has only a small error, measured according to the unknown distribution. Our interest here is in the conformance learning of read-once formulas using queries, along the lines of Angluin's query learning model [1]. Their paper showed exact learning is possible on *monotone* read-once formulas using only membership queries, and exact learning is possible for all read-once formulas given equivalence queries. In contrast, we do not need equivalence queries—they can be simulated using membership queries and efficient computation. Thus, we will learn a "local" read-once formula using membership queries. We would like to minimize the number of such queries we use. Our results use parameters of the blueprint and of the implementation to tightly characterize the number of queries needed.

One could also look at the scenario we describe in the property-testing framework. We define the distance between two Boolean functions on the same n inputs to be the fraction of the input settings on which they differ. The strict property-testing question is to design a (probe) efficient algorithm to distinguish an implementation that completely agrees with the blueprint from an implementation that is ϵ-far, for a given ϵ. As stated above, one of our results is an $O(n)$-probe

algorithm to decide if the implementation is identical to the blueprint. In the looser, property-testing framework, a trivial algorithm is to sample $O(1/\epsilon)$ probes and return "far" if the implementation disagrees on any of the sampled probes. A similar idea easily extends to tolerant property testing, where we are given $\epsilon_1 < \epsilon_2$ and have to output "close" when the implementation is at most ϵ_1-far from the blueprint and "far" when it is at least ϵ_2-far.

Von Neumann considered the problem of computing a function using a Boolean formula (i.e., a circuit where all gates have fan-out 1) [16]. Each gate is allowed to fail independently with probability ϵ, and the failure model is that when a gate fails, it produces the complement of its correct output. The input variables are allowed to be duplicated without error. By alternating computation stages with error-correction stages, von Neumann showed that reliable computation was possible as long as $\epsilon < .0073$. (The formula is called reliable if its probability of producing the correct bit as output is a constant greater than .5, where the probability is over the random faults.) Hajek and Weller [10] showed reliable computation was possible even for ϵ up to 1/6 by increasing redundancy and using a tree of three-way majorities for error correction, but their result depends on assuming that the failure probability of each gate is precisely ϵ. Evans and Schulman [5] generalize this to higher odd arity gates.

In a broader view, we may interpret the gates constituting our read-once Boolean formulas as components in a system, and our goal in this case is to diagnose which components are faulty. This is a special case of the general theory of fault diagnosis in such systems as defined by Reference [13] for which we are able to find rigorous and efficient algorithms. While in this article we only consider faulty components that always produce a complement of their true answer, we can generalize this slightly and still preserve the property that there is enough information to identify the faulty components.

2 PRELIMINARIES

In this article, we will deal with read-once formulas—that is, gates and inputs in the formula will have fan-out 1. We will refer to the tree underlying a read-once formula as its *topology*. The classical model of read-once formulas uses a basis consisting of AND, OR, and NOT gates, leading to the use of the term "AND-OR trees." Our results apply to this basis. They also apply to more general bases, consisting of a set of "identifiable" gates, a notion we make more precise in Section 3. The intuition is that the allowable gates are those for which we can solve our problem on a depth-1 formula consisting only of that gate.

Our error model will be deterministic: a "faulty" or "bad" gate will always output precisely the wrong answer, relative to the inputs received from its children. One may think of the fault as adding a NOT gate to the output of the bad gate. However, a "good" gate will always output the correct answer relative to the inputs received from its children. Note that the output of a gate is dependent upon errors occurring in its descendants, and errors themselves may cancel out, as in References [10, 16]. In addition, we will permit the inputs themselves to be faulty—they are negated consistently in the same manner.

Although our errors are deterministic, the positive results extend to the stochastic case, where we instead repeat each probe a polylogarithmic number of times to determine with high probability whether that probe can result in a correct answer, an incorrect answer, or both. This extension to the probabilistic setting is a reason for considering our fault model instead of one where we are concerned about the overall truth tables, rather than the correctness of individual gates; in a probabilistic setting, cancellation of errors does not allow an incorrect circuit to produce correct results.

Our model and problem can be stated as follows: Suppose we are given a read-once formula, which we will refer to as the *blueprint*. This corresponds to an intended design for the formula.

In addition, we are given an *implementation* of the formula, where some of the gates of the blueprint are replaced with faulty versions as described above. We are given only blackbox access to the implementation—if we set all the inputs, we are allowed to see the output, but we are given full access to the blueprint; we can see exactly its topology and its gates. Our goal is to use our knowledge of the blueprint to determine first if the implementation is faithful to the blueprint, and if it is not, diagnose or locate the faults in the implementation.

In Section 3, we will prove our two positive results. The first is that we can always determine whether the implementation is identical to the blueprint—that is, every gate in the implementation behaves identically to its corresponding gate in the blueprint.

Theorem 2.1. *Given a blueprint read-once formula f and an implementation $\hat{f}$, it is always possible to test equality, $f = \hat{f}$, in a number of probes linear in the sizes of the formulas and exponential in the maximum arity.*

We will also show a matching lower bound.

Proposition 2.2. *There exists a blueprint read-once formula and an implementation of this blueprint, such that testing equality requires a number of probes linear in the size of the formulas.*

The second goal is to diagnose the errors. Suppose that we are promised there is some constant bound c on the number of errors. Then there are only $O(n^c)$ possible implementations. Treating each of these as a "blueprint," we can compare it to the actual implementation using the same method as testing equality. Thus, in $O(n^{c+1})$ queries it is possible to diagnose the formulas if there is a constant bound c on the number of errors.

However, if the number of errors is not bounded by a constant, our positive result revolves around the existence of a certain $\{0, 1\}$-oracle for the implementation. This oracle, given a partial setting of the inputs, produces two complete inputs, each extending the given partial setting: one that results in the implementation outputting 0 and the other that results in the implementation outputting 1, if such inputs exist.

Theorem 2.3. *Given a blueprint read-once formula f, an implementation $\hat{f}$, and a $\{0, 1\}$-oracle for $\hat{f}$, we can exactly identify the erroneous gates in $\hat{f}$ in a number of probes and oracle queries linear in the sizes of the formulas and exponential in the maximum arity.*

To actually make use of this result, we provide a probabilistic method for implementing this oracle under a "balance" condition whose formal definition is given later in this article. Informally, an implementation is balanced if at every gate the probability of computing a 0 and the probability of computing a 1 are lower bounded by an inverse polynomial, where the probability is over uniformly random choices for the input bits. Using the theorem above, we provide a probabilistic method of exact diagnosis when the balance *of the implementation* is at least 1/poly.

We will show a matching negative result in Section 4—that we essentially require 1/poly balance. One could hope first that a 1/poly balance of the blueprint is sufficient or even that no balance condition is necessary, as we have no effective control over the balance of the implementation. However, in Section 4, we provide a blueprint that has 1/poly balance, for which we can get an implementation with just $O(\log n)$ errors that has worse than 1/poly balance and in fact requires $2^{\Omega(n)}$ probes to diagnose exactly.

Theorem 2.4. *There exists a blueprint read-once formula with 1/poly balance and an implementation that has only $O(\log n)$ errors that require $2^{\Omega(n)}$ queries to locate all the errors.*

In Section 5, we will view this problem as a learning problem and adapt our results as learning problems.

3 TESTING EQUALITY AND DIAGNOSIS OF ERRORS

Let g be any gate in a formula and let $x_1, x_2, \ldots, x_k$ be a partial assignment to exactly those variables that lie in the subtree rooted at g. We use the notation $g(x_1, x_2, \ldots, x_k)$ to denote the output of g on this partial assignment.

Next, we define a notion of identifiability for a family of read-once formulas.

Definition 3.1. Let $\mathcal{F}$ be a family of read-once formulas with the same topology. $\mathcal{F}$ is *identifiable* if for every two formulas $f_1, f_2 \in \mathcal{F}$ that are different anywhere, there exists an input x such that $f_1(x) \neq f_2(x)$.

Definition 3.2. For a formula f, define the neighborhood $\mathcal{E}(f)$ to be the set of read-once formulas that are constructed by modifying f by doing an arbitrary subset of the modification operations listed below:

- For an input x to f, replace x with $\overline{x}$
- For a gate g in f, replace g with NOT-g.

We can only expect to diagnose the errors of a formula f exactly if $\mathcal{E}(f)$ is identifiable. We now define the same notion of identifiability for gates.

Definition 3.3. Let g be an arbitrary gate with input arity r. Let f_g be the depth-one read-once formula on r inputs, containing just the gate g. We will say g is *identifiable* if $\mathcal{E}(f_g)$ is identifiable.

We note a few facts about identifiable gates. First, every identifiable gate depends on all of its inputs—if it does not depend on some input, then there is no way to distinguish between negating that input and not. This is not a strong restriction: a gate that does not depend on one of its inputs may as well not have that input.

Second, for any even arity r, all threshold gates are identifiable (a threshold gate is one that counts the number of inputs that are 1, and outputs 1 if it exceeds some fixed threshold). For any odd arity r, all non-trivial threshold gates, except the $(r+1)/2$-threshold gate, are identifiable. In particular, AND and OR gates of any arity are identifiable.

Finally, if g is an identifiable gate, then $\hat{g}$ given by negating one or more of the inputs to g and possibly the output to g, is also identifiable. NOT-gates are not identifiable, but if the output of a NOT-gate f feeds into an identifiable gate g, then the formula with gates f and g is identifiable.

However, XOR gates are not identifiable. The set $\mathcal{E}(XOR)$ realizes just two truth tables, corresponding essentially to the parity of the number of faults surrounding the XOR gate, while there are 2^{r+1} possible formulas in $\mathcal{E}(XOR)$.

We note that we have given a definition of identifiability of a gate g in terms of its neighborhood $\mathcal{E}(g)$, and that this neighborhood is linked to our error model, where faulty gates are wrong by negation. However, the results we give below are dependent on the notion of identifiability. If we choose a different error model and a definition of neighborhood appropriate to that error model (that is, so that the neighborhood of a gate captures all of the local errors it or its children may see), then our results below will still apply to identifiable gates in that error model.

A basis consists of a set of allowed types of gates. For example, a standard basis consists of AND, OR, and NOT gates; another consists of just NAND gates.

Definition 3.4. A basis $\mathcal{G}$ of gates will be called *identifiable* if every read-once formula f over the basis $\mathcal{G}$ is identifiable.

We observe that an easy consequence of this definition is that if a basis $\mathcal{G}$ is identifiable, then each gate $g \in \mathcal{G}$ must be identifiable, as there is a formula consisting of just the gate g over that basis. Conversely, we will show (in the proof to Theorem 3.8) if a read-once formula f consists of

only identifiable gates, then $\mathcal{E}(f)$ is identifiable. Thus, a basis $\mathcal{G}$ is identifiable if and only if each gate in $\mathcal{G}$ is identifiable—that identifiability of a set of gates means that any formula constructed from those gates is also identifiable.

While our results hold for an arbitrary basis, it may be helpful for intuition to regard the gates in the formula as AND, OR, NAND, and NOR. In the remainder of this article, we will fix a finite basis consisting of identifiable gates $\mathcal{G}$.

3.1 Testing Equality of the Implementation

Let r^* be the maximum input arity of a gate in $\mathcal{G}$. We begin by showing that for any blueprint read-once formula f over an identifiable basis $\mathcal{G}$, and an implementation $\hat{f} \in \mathcal{E}(f)$, it is always possible to test equality of f and $\hat{f}$ in $O(2^{r^*}|f|)$ probes to the blackbox. Recall that equality here means that every gate of $\hat{f}$ behaves identically to the corresponding gate in $\hat{f}$[1].

Theorem 3.5 (Restatement of Theorem 2.1). *Let f be a blueprint read-once formula and $\hat{f} \in \mathcal{E}(f)$ an implementation. Then there exists an algorithm to determine the equality of f and $\hat{f}$ in $O(2^{r^*}|f|)$ probes.*

Proof. The algorithm begins at the root of f, verifies that the root is correct, and then recurses on each subformula.

Let g be the gate at the root of f, and suppose g has input arity r. Let $f_1, \ldots, f_r$ be the subformulas of the children of g in the blueprint, and let $\hat{f}_1, \ldots, \hat{f}_r$ be the subformulas of the children of g in the implementation.

Choose vectors $\vec{x}_{i,b}$ for $1 \le i \le r$ and $b \in \{0,1\}$ such that $f_i(\vec{x}_{i,b}) = b$, where $\vec{x}_{i,b}$ is a setting of exactly those variables in the subformula f_i. Since the blueprint f is known, we can work backwards from each f_i to find such inputs. Then perform all 2^r probes of $\hat{f}(\vec{x}_{1,b_1}, \ldots, \vec{x}_{r,b_r})$ for $b_1, \ldots, b_r \in \{0,1\}$.

We will now show that if all 2^r probes of $\hat{f}$ agree with their corresponding evaluations on f, then the gate g at the root is correct. First, based on the fact that the probes agree with f, we infer some conditions on the values at the $\hat{f}_i$'s, which we cannot observe directly.

First, for every i, $\hat{f}_i(\vec{x}_{i,0}) \neq \hat{f}_i(\vec{x}_{i,1})$. Suppose otherwise. Since g is identifiable, it depends on each input. Thus, there is some setting of the inputs other than its ith input such that g is either the identity or negation on its ith input. So, if $\hat{f}_i(\vec{x}_{i,0}) = \hat{f}_i(\vec{x}_{i,1})$, then g will appear to not depend on its ith input in $\hat{f}$, where we know it does in f—proving that f and $\hat{f}$ are not equal, but our assumption here is that all 2^r probes of $\hat{f}$ agree with the corresponding evaluation on f.

Now we know that for each i, $\hat{f}_i(\vec{x}_{i,b})$ is equal to either b or $1-b$, which is effectively saying the ith input to g is negated or not. And g itself is possibly negated. However, by hypothesis, g is identifiable, and so we can determine now if g has been replaced by its negation in the implementation. If it has, we can stop and say they are not equal. If it has not been negated, then we can also apply the identifiability of g to claim that for all i and b, $\hat{f}_i(\vec{x}_{i,b}) = b$.

Now, we can recurse on each subformula $\hat{f}_i$. Since we know that g is correct and we have inputs $\vec{x}_{j,b}$ for $j \neq i$ to control the other inputs to g, we can effectively isolate and look at the subformula $\hat{f}_i$ on its own.

It is clear that for each gate g of the read-once formula f, we perform one set of 2^r probes where r is the input arity of g. Thus, the total number of probes is linear in the size of f and 2^{r^*} where r^* is the maximum arity of a gate in $\mathcal{G}$. □

[1]It is possible to modify the algorithm in this result to test equality of f and $\hat{f}$ as Boolean formulas instead and drop the identifiability requirement.

We can pick all the probes we will perform ahead of time, and so our algorithm is in fact non-adaptive. In addition, we note that testing a node g in the formula actually tests both g and its children, so if our basis contains both high and low arity gates, then we may not need to pay the full cost for isolated high-arity gates.

We now state a proposition, showing that we cannot in general solve this problem in a sublinear number of probes.

Proposition 3.6 (Restatement of Proposition 2.2). *There exists a read-once formula f using only arity-2 OR gates such that given an unknown implementation $\hat{f}$, $\Omega(|f|)$ probes are required to test equality of $\hat{f}$ with f.*

Proof. Define a "caterpillar" as follows: a caterpillar on 1 gate is simply a formula of one gate. A caterpillar on n gates is a tree where the left subtree of the root is a caterpillar on $n-1$ gates, and the right child of the root is an input. Let f be a caterpillar on n arity-2 OR gates.

Consider the subset $\mathcal{F} \subset \mathcal{E}(f)$ of possible implementations, consisting of read-once formulas with exactly two errors on successive gates in the caterpillar. Label the nodes from the root down the caterpillar as $v_1, \ldots, v_n$. For $1 \leq i < n$, denote by F^i the implementation that contains errors exactly at v_i and v_{i+1}. Let $\mathcal{F} = \{F^i | i = 1, \ldots, n-1\}$.

We may assume that any algorithm for testing equality is non-adaptive, since on any probe, if the implementation produces a different response from the blueprint, we reject right away; and if not, then we continue (or accept). Thus, we may consider an algorithm for equality as a set of probes S.

We note that the only input on which f outputs 0 is the all 0's input—each $F^i \in \mathcal{F}$ also outputs 0 on this input.

Thus, to find a difference, for each $F^i \in \mathcal{F}$, S must contain an input on which F^i outputs 0 and f outputs 1. Fix an F^i. The errors in F^i are at v_i and v_{i+1}. We know that an input distinguishing F^i from f must contain a 1, otherwise both of these will output 0. If an input attached to a gate above v_i is set to 1, then both formulas will output 1. Thus, some input below v_i must be set to 1. In addition, if the right child input of v_i is set to 0, then some input below v_{i+1} is set to 1. In this case, v_{i+1} (being in error) will output 0, leading to v_i (again being in error) outputting 1. Thus, in order for an input to cause F^i to output 1, the probe must have a 1 at the input that is the right child of v_i and have no 1's at inputs above v_i.

Thus, each $f_i \in \mathcal{F}$ requires a different input to distinguish it from f, and so the set S contains at least $|\mathcal{F}|$ probes. □

3.2 Exact Diagnosis of Errors in the Implementation

Now, we approach the main question of exact diagnosis of the errors in the implementation. Here, we are not only interested in whether there *are* any errors, but also identifying exactly where the errors occur. To be precise, we are again given a blueprint f and an implementation $\hat{f}$. Our aim is to identify exactly which gates in $\hat{f}$ are faulty compared to their counterparts from f.

We define a $\{0,1\}$-*oracle* for the implemented read-once formula $\hat{f}$. The rough idea is that the oracle is given a partial assignment to the inputs of $\hat{f}$ and returns two inputs both matching this partial assignment—one causing the implementation to output 0 and the other causing it to output 1. The definition below restricts the class of partial assignments that are allowed.

Definition 3.7. A $\{0,1\}$-*oracle* for a read-once formula $\hat{f}$ is an oracle that does the following: It takes as input a node g in $\hat{f}$ and a subset C of the children of g. In addition, it receives a partial assignment fixing exactly those inputs that are not in the subtrees rooted at nodes in C. Given these, it outputs two settings of the inputs under C such that, combined with the given partial setting:

one setting causes the formula $\hat{f}$ to output 0, and the other causes the formula $\hat{f}$ to output 1. If one or the other such setting does not exist, then it outputs the one that does not exist.

We now show how to use such an oracle to exactly diagnose an implementation $\hat{f}$.

Theorem 3.8 (Restatement of Theorem 2.3). *Let f be a blueprint read-once formula and $\hat{f}$ be an implementation. Let A be a {0,1}-oracle for $\hat{f}$. Then there exists an algorithm to exactly diagnose the errors of $\hat{f}$, using $O(n2^{r^*})$ probes and $O(nr^*2^{r^*})$ oracle calls.*

Proof. The key point to our algorithm will be that the identifiability of a node allows us to determine the faultiness of each of its children.

Suppose that $\vec{x}$ and $\vec{y}$ are partial assignments to the inputs of f such that each input is given an assignment in at most one of $\vec{x}$ or $\vec{y}$. Then let $\vec{x} \circ \vec{y}$ denote the partial assignment that assigns the values of $\vec{x}$ and the values of $\vec{y}$. If $\vec{x}$ and $\vec{y}$ exactly partition the inputs, then $\vec{x} \circ \vec{y}$ is a full assignment.

To apply identifiability at a node g in $\hat{f}$, we will need the following partial assignments:

(1) A partial assignment $\vec{y}_g$ for all inputs not under g such that either, for every partial assignment $\vec{x}$ for exactly the inputs under g, $f(\vec{y}_g \circ \vec{x}) = g(\vec{x})$, or for every partial assignment $\vec{x}$ for exactly the inputs under g, $f(\vec{y}_g \circ \vec{x}) = 1 - g(\vec{x})$. In other words, $\vec{y}_g$ is such that the overall output of the formula is determined by the output of gate g.
(2) For each child c_i of g, two partial assignments to the inputs under c_i, $\vec{z}_{c_i,0}$ and $\vec{z}_{c_i,1}$, such that $c_i(\vec{z}_{c_i,0}) = 1 - c_i(\vec{z}_{c_i,1})$ (that is, one will lead to c_i outputting 0 and the other to outputting 1, though we will not need to know which is which at this point).

We will implement a recursive approach involving both a top-down phase and a bottom-up phase. In the top-down phase, we will maintain the following *entry condition*: when the recursion reaches a node g, we will have a partial assignment $\vec{y}_g$ as described above. And in the bottom-up phase, we will maintain the following *exit condition*: when we finish at g, we will have diagnosed the errors at every node in the subtree of g, except for g itself.

Note that if we have diagnosed a subtree, then we know exactly where the errors occur and thus have full knowledge of the subtree. As a result, we can generate inputs under those subtrees as needed.

The base case for the entry condition is clear: the first node we reach is the root, and every input is under the root, so an empty "partial assignment" suffices.

Now, we show how we satisfy the entry condition for nodes other than the root. Let g be the current node for which we already have $\vec{y}_g$ as a partial assignment to satisfy the entry condition to g. Let $c_1, \ldots, c_r$ be the children of g. Suppose we have already completed the exit conditions at some (possibly empty subset) of the children of g. For notational convenience, let us say the completed children are $c_1, \ldots, c_k$. For each completed child c_i, we can generate $\vec{z}_{c_i,0}$ and $\vec{z}_{c_i,1}$ as described above, because the exit condition applies. Note that we can only guarantee that $c_i(\vec{z}_{c_i,0})$ and $c_i(\vec{z}_{c_i,1})$ are different but not exactly their values, since the exit condition at c_i does not require us to know whether c_i itself is faulty.

For each choice of $b_1, \ldots, b_k \in \{0, 1\}$, we provide A, the oracle, with the following arguments: the node g, a subset of its children $\{c_{k+1}, \ldots, c_r\}$, and the partial assignment $\vec{y}_g \circ \vec{z}_{c_1,b_1} \circ \cdots \vec{z}_{c_k,b_k}$, which assigns to all inputs not under $c_{k+1}, \ldots, c_r$. Note that when we explore the first child under a node g, the partial assignment will just be $\vec{y}_g$.

Since g is identifiable and thus depends on all of its inputs, there is some choice of $b_1, \ldots, b_k$ for which the oracle succeeds and returns two inputs $\vec{x}_0$ and $\vec{x}_1$ compatible with the given partial assignment. In fact, we can find two such inputs that have Hamming distance 1 using the oracle. If $\vec{x}_0$ and $\vec{x}_1$ are at distance greater than 1, then simply query the oracle on every input on a shortest

path between them on the hypercube and we will find two successive points where the answers change. For notational convenience, assume this input on which $\vec{x}_0$ and $\vec{x}_1$ differ occurs under c_{k+1}. Then let $\vec{y}_{c_{k+1}}$ be a partial assignment agreeing with both $\vec{x}_0$ and $\vec{x}_1$, assigning to all inputs not under c_{k+1}. $\vec{y}_{c_{k+1}}$ must satisfy the entry condition for c_{k+1}, since $f(\vec{x}_0)$ and $f(\vec{x}_1)$ differ. (Note, therefore, that the oracle's answers control the order in which the children of g are explored in the top-down phase.)

The base case for the exit condition is also simple: if g is an input (a leaf), then the exit condition is vacuous and is thus satisfied automatically.

In general, suppose again that we are at a node g, with children $c_1, \ldots, c_r$, where we have now satisfied the exit condition for all the children. Then, we are able to construct the partial assignments $\vec{z}_{c_i,b_i}$ for each $1 \le i \le r$ and $b_i \in \{0,1\}$. We also have the partial assignment $\vec{y}_g$ from the entry condition to g. For each choice of $b_1, \ldots, b_r \in \{0,1\}$, we query the implementation $\hat{f}(\vec{y}_g \circ \vec{z}_{c_1,b_1} \circ \cdots \circ \vec{z}_{c_r,b_r})$ to construct a truth table for g, on the b_i's and the output value f. Identifiability then applies and tells us exactly which of the children are faulty. Since the exit conditions at the children were already satisfied, the nodes in each of their subtrees were already diagnosed. We have now diagnosed the children of g, which completes the exit condition for g.

Observe that if g is not the root, then we cannot distinguish between a fault at g and a fault between g and the root, so we do not yet diagnose g. However, at the root, we can exactly diagnose g, since any error appearing to occur at g must occur exactly there, because there is nothing above g. This allows us to complete the diagnosis at the root.

To analyze oracle and query complexity, we see that we use at most $2r^*$ oracle calls for the entry condition to each node, and at most 2^{r^*} probes for the exit condition at each node. □

Thus, we have given an explicit procedure proving that the formula is identifiable if the gates in the formula are identifiable.

The $O(2^{r^*})$ oracle calls per gate is required in general but is not necessary for AND and OR. In the latter case, $O(r^*)$ calls suffice: at a gate g with a set of children C', we do not need to find the inputs for each child in C' afresh each time, and we do not need to iterate over all $2^{|C'|}$ possibilities each time. If c was the last child we recursed on, then we have a setting that already works for $C' \setminus \{c\}$, and thus only need to test the two values of c. This reduces the number of oracle calls to $2r^*$ per node, from $r^*2^{r^*}$.

To see that this optimization does not work in general, take for example a gate with input arity 3, where its output is equal to its second input if the first input is 0, and equal to its third input if the first input is 1. A setting for the first input, which allows us to read the second input, is not a usable setting to read the third input.

Now, we provide a probabilistic implementation of this oracle, under an additional assumption on the *implementation* read-once formula $\hat{f}$. To state this assumption, we define a notion of balance.

Definition 3.9. The *balance* of a gate g in a read-once formula is the lesser of the probabilities that the value of the gate is 0 or 1 when the inputs below that gate are chosen uniformly at random. The *balance* of a read-once formula is the minimum of the balances of all of its gates.

If the implementation $\hat{f}$ has balance that is lower bounded by an inverse polynomial $1/p(n)$, then the following lemma will provide a probabilistic algorithm for a $\{0,1\}$-oracle for $\hat{f}$:

Lemma 3.10. *If $\hat{f}$ is an implementation with balance lower bounded by an inverse polynomial $1/p(n)$, then there is a probabilistic implementation of a $\{0,1\}$-oracle for $\hat{f}$ using $(p(n))^{r^*} \log(2/\varepsilon)$ probes, which fails with probability at most ε.*

Proof. The implementation is simple. On input g, a subset of its children C, and a setting $\vec{x}$ of the inputs not under C, we will randomly choose values for the inputs under C until we have a

setting that has $\hat{f}$ output 0 and a setting that has $\hat{f}$ output 1, or until we have tried $(p(n))^r \log(2/\varepsilon)$ inputs.

To prove correctness, it suffices to show that when such settings of the inputs under C exist for both 0 and 1 outputs, we find them with high probability. We will lower bound the probability of finding each such setting in a single probe. For simplicity, let us speak of finding a setting of the inputs under C that causes $\hat{f}$ to output 0 (the other case is obviously identical). Since such a setting exists, there exists at least one output of each gate in C such that $\hat{f}$ would output 0.

Then the probability when choosing the inputs under C at random that we produce the desired outputs at the gates in C is at least $1/(p(n))^{r^*}$. Thus, the probability of success of one probe is at least $1/(p(n))^{r^*}$, and the probability of failure after $(p(n))^{r^*} \log(2/\varepsilon)$ probes is at most $\varepsilon/2$. A union bound over failing to find the inputs to produce 0 and the inputs to produce 1 yields an error probability of at most ε. □

Finally, we note that a modification of this algorithm may also be used to test equality of the blueprint and implementation. The oracle simply uses the blueprint to pick inputs that yield 0 and 1, and if either disagrees in the implementation, then we stop and reject.

4 LOWER BOUNDS FOR DIAGNOSIS

In this section, we show that it is not always possible to locate all errors, or even one error, in polynomially many queries if the balance of the implementation is not polynomially bounded, even if there are only $\log n$ errors in the implementation and even if we assume that we start from a blueprint that is balanced. In addition, we will use only the arity-2 AND and OR gates. In particular, we now construct a blueprint of polynomial balance and show a corresponding class of implementations with $\log n$ errors that cannot be distinguished using polynomially many queries.

The tree for our formula f on n nodes will be a complete binary tree. The top $\log n - \log\log n$ layers will be all AND nodes, and the lower $\log\log n$ layers will be all OR nodes.

Proposition 4.1. *f has polynomial balance.*

Proof. Let r be a node in the lower $\log\log n$ layers. Then r outputs 0 if and only if all inputs under it are 0. Since it is in the lower $\log\log n$ layers, it has at most $2^{\log\log n} = \log n$ inputs under it, and thus $\frac{1}{n}$ of its inputs output 0 and all other inputs output 1. Thus, it has polynomial balance.

Now let r be a node in the upper $\log n - \log\log n$ layers. Let $s_1, \ldots, s_k$ be its descendants at level $\log\log n$. Then r is 0 if any of $s_1, \ldots, s_k$ is 0. This is thus with probability at least $1/n$. So, we need to guarantee the probability r is 1 is high enough. r is 1 if and only if all of $s_1, \ldots, s_k$ are 1. Thus, the probability that r is 1 is $(1 - 1/n)^k$. $k < n$, so the probability r is 1 is at least $\frac{1}{e}$. □

Let $r_1, \ldots, r_k$ for $k = n/\log n$ be the nodes at height $\log\log n$ (the maximal height OR nodes). The class of implementations will be those where $\log n$ errors are located in the set $\{r_1, \ldots, r_k\}$. Note there are $\binom{\frac{n}{\log n}}{\log n}$ such formulas, which is super-polynomial.

Proposition 4.2. *The class $\mathcal{E}(f)$ of implementations cannot be distinguished in polynomially many queries.*

Proof. We allow the distinguishing algorithm more power. In particular, we also permit the algorithm to set the values of $r_1, \ldots, r_k$ (before being modified by errors), instead of the inputs. The only way the algorithm can gain information is if it can set each of the faulty r_i's to 0 (so that after fault, they output 1) and the non-faulty r_i's to 1. Any other values at the r_i will cause the implementation to output 0 and offer no information. Then the best that an algorithm can do to try to diagnose the implementation is to try every possibility—but there are super-polynomially many. □

5 ERROR DIAGNOSIS AS LEARNING

Work by Angluin et al. [1] and Bshouty et al. [3] provide algorithms for learning read-once formulas in a general sense. However, their results rely on the use of equivalence queries. This allows them to compare to a constant circuit to produce an input that causes the implementation to output a specific value. In particular, this allows them to seek inputs that yield an output of 0 and inputs that yield an output of 1.

In our model, equivalence queries would have to be between the implementation and another read-once formula in $\mathcal{E}(f)$, since our concept class is no longer any read-once formula, but a read-once formula in $\mathcal{E}(f)$. This would appear to be much less powerful. In fact, we can simulate such equivalence queries using a linear number of membership queries in our model. The algorithm is simple—run the algorithm of Theorem 3.5 on the two formulas. Note that if q is the formula that we want to query for equivalence, then q is available to us explicitly, and $\hat{f}$, as always, is available as a black box. Either the algorithm will complete, affirming equivalence, or it will halt at some point where we have an input that causes the formulas to have differing outputs. Thus, the equivalence queries that are permissible in our model do not add any extra power to the learner.

Instead of equivalence queries, we rely on a $\{0,1\}$-oracle that gives us inputs that cause the implementation to output specific outputs, and an efficient probabilistic implementation of this oracle in turn depends on the balance condition.

5.1 PAC Learning under a Product Distribution

Schapire [14] shows that it is possible to learn read-once (arithmetic) formulas under a PAC model if the inputs are drawn from a product distribution. Our result can be adapted to provide a PAC-like-learning algorithm for inputs drawn from a product distribution, under our model. This yields a more combinatorial alternative to the algorithm given by Schapire in our setting.

The intuitive idea to our adaptation is that we can approximately learn the product distribution over the inputs, and then we can implement a probabilistic oracle like the one in Lemma 3.10. Now the balance condition needs to be reformulated in the obvious way for the particular product distribution at hand, rather than for the uniform distribution. If there are no unbalanced nodes, then this oracle and the algorithm in Theorem 3.8 suffice. If there are unbalanced nodes, however, then we will take advantage of the fact that we only need to learn approximately. So, instead of diagnosing under unbalanced nodes, we will simply guess an unbalanced implementation, which will be approximately close to the actual implementation, because both will have low balance.

Proposition 5.1. *Suppose we are given an error parameter δ and an accuracy parameter ϵ, along with blackbox access to a formula $\hat{f}$ as before, and $\mathcal{P}$ a product distribution on the inputs. Then with probability at least $1-\delta$, we can output an explicit formula $\tilde{f} \in \mathcal{E}(f)$ for which $\Pr_{x\sim\mathcal{P}}[\tilde{f}(x) \neq \hat{f}(x)] \leq \epsilon$.*

Proof. First, we draw from $\mathcal{P}$ to learn the product distribution using standard techniques until our hypothesis $\tilde{\mathcal{P}}$ is with probability at least $1-\delta/2$, within variational distance $\epsilon/(8n^2/\epsilon \cdot \log(2n/\delta) \cdot 2^{r^*})$ of the $\mathcal{P}$.

Define a node to be balanced if its balance is at least $\epsilon/4n$, and unbalanced otherwise. Each time we need to use the $\{0,1\}$-oracle, we will use the parameters of $\tilde{\mathcal{P}}$ to generate samples as needed.

We define a conditional balance of a node to be the balance of the node when we have fixed some of its inputs. We will say a node is conditionally balanced under some setting of some of its inputs if its conditional balance when those inputs are fixed is at least $\epsilon/4n$, and conditionally unbalanced otherwise.

We now describe the changes to the algorithm. Suppose we are at a gate g with children C and have diagnosed some of its children $C' \subset C$. Instead of querying the oracle $2^{|C'|}$ times, we will skip the role of the oracle. We will sample $4n/\epsilon \cdot \log(2n/\delta)$ inputs for the inputs under $C \setminus C'$. Fix one of the settings of the inputs under C', and see if the sampled inputs under $C \setminus C'$ will produce both a 0 and a 1. If yes, then we proceed as before. Otherwise, we simply pick one of the sampled inputs arbitrarily and fix those as the inputs under $C \setminus C'$. Now note that we can still pick inputs setting g to 0 and to 1 by using the fact that C' is non-empty and is diagnosed (so we must have been able to find inputs setting g to 0 and to 1). If C' were empty, then we would never have recursed to g.

Note that we have not diagnosed $C \setminus C'$ and their descendants at this point. Instead, at the end, we will simply pick a distribution of faults under each $C \setminus C'$ to minimize the balance at each node of $C \setminus C'$ and by deciding whether the root is in error or not, we can determine which input is biased against.

We note that the balance of a node can be written as the minimum of the probability a node is 0 and the probability a node is 1. Each is linear in the probabilities of each of its children being 0 or 1. Thus, a dynamic program which for each node remembers an arrangement of errors under that node which minimizes the probability of the node being 0, and an arrangement that minimizes the probability of the node being 1, will also allow us to minimize the balance of each node.

First, we analyze the probability of failure. The first source of failure is a probability $\delta/2$ of failure from learning the product distribution. The other source is if some node g with children C, and some subset $C' \subset C$ of diagnosed children, was conditionally balanced but we decided it was not. Since it is conditionally balanced, we should find an output for 0 and for 1 in each trial with probability at least $\epsilon/4n$. Since we run $4n/\epsilon \cdot \log(2n/\delta)$ trials, the probability of failure is at most $\delta/2n$. Since we can fail at most n gates, the total probability of this type of failure is $\delta/2$. A union bound gives us an overall probability of failure of at most δ.

Second, we analyze the quality of approximation in the absence of failure. Note that the approximation comes from two sources: our approximating the product distribution $\mathcal{P}$, and the replacing of subformulas with computed minimum-balance formulas. Since our distribution is variational distance $\epsilon/(8n^2/\epsilon \cdot \log(2n/\delta))$ and we use the learned distribution at most $4n^2/\epsilon \cdot \log(2n/\delta)$ times (each set of $4n/\epsilon \cdot \log(2n/\delta)$, trials either let us diagnose a node or let us skip diagnosing a node, since the conditional balance was bad and there are n nodes. The total error caused by approximating $\mathcal{P}$ is at most $\epsilon/2$.

Now at each node g for which we could not diagnose some of its children $C \setminus C'$, the conditional balance for all settings of C' was at most $\epsilon/4n$, so the distribution on $C \setminus C'$ is $\epsilon/4n$-close to constant. Since our choice of a guess for the nodes under $C \setminus C'$ minimizes balance, it must also be $\epsilon/4n$-close to constant, so the distance between the implementation and our guess is at most $\epsilon/2n$. Since there are at most n nodes, the probability that for some input we are wrong at one or more of them is at most $\epsilon/2$, giving us an overall probability of ϵ of being incorrect. □

6 CONCLUSIONS AND OPEN PROBLEMS

We have taken the first steps in diagnosing faults in an implementation given a blueprint in a learning-theoretic framework. It would be interesting to extend these ideas to a richer class of models, but this seems difficult given only the blackbox access to the implementation that we assume. A natural generalization would be to remove the "read once" restriction and extend it to all formulas. But in the blackbox testing framework, we are limited by being unable to test subcircuits independently, because there might be inputs they share with the complement. Extension beyond formulas to circuits seems even more difficult because of the dependence between errors in different subcircuits. The other generalizations could be in the fault model. A potentially feasible extension is to show that our lower bounds also hold in the stochastic fault model. This is

not obvious, because error cancelation effects that contribute to the lower bound in the case of deterministic faults might be low-probability events in the case of stochastic faults. Extensions to "stuck at" faults are also challenging because of the lack of identifiability for such faults. Models that allow us to examine values in the interior of the implementation (at a cost) might provide us a way of obtaining positive results for these generalizations.

REFERENCES

[1] Dana Angluin, Lisa Hellerstein, and Marek Karpinski. 1993. Learning read-once formulas with queries. *J. ACM* 40, 1 (1993), 185–210.

[2] S. Assaf and E. Upfal. 1991. Fault-tolerant sorting networks. *SIAM J. Disc. Math.* 4 (1991), 472–480.

[3] Nader H. Bshouty, Thomas R. Hancock, and Lisa Hellerstein. 1995. Learning Boolean read-once formulas over generalized bases. *J. Comput. System Sci.* 50, 3 (1995), 521–542.

[4] W. Evans and L. Schulman. 1993. Signal propagation, with application to a lower bound on the depth of noisy formulas. In *Proceedings of the 34th IEEE Symposium on Foundations of Computer Science*. IEEE, 594–603.

[5] William S. Evans and Leonard J. Schulman. 2003. On the maximum tolerable noise of k-input gates for reliable computation by formulas. *IEEE Trans. Inform. Theor.* 49, 11 (2003), 3094–3098.

[6] T. Feder. 1989. Reliable computation by networks in the presence of noise. *IEEE Trans. Inform. Theor.* 35, 3 (1989), 569–571.

[7] Hideo Fujiwara. 1985. *Logic Testing and Design for Testability*. MIT Press, Cambridge, MA.

[8] P. Gács and A. Gál. 1994. Lower bounds for the complexity of reliable Boolean circuits with noisy gates. *IEEE Trans. Inform. Theor.* 40 (1994), 579–583.

[9] Sally A. Goldman, Michael J. Kearns, and Robert E. Schapire. 1993. Exact identification of read-once formulas using fixed points of amplification functions. *SIAM J. Comput.* 22, 4 (1993), 705–726.

[10] B. Hajek and T. Weller. 1991. On the maximum tolerable noise for reliable computation by formulas. *IEEE Trans. Inform. Theor.* 37, 2 (1991), 388–391.

[11] N. Pippenger. 1985. On networks of noisy gates. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science*. IEEE, 30–36.

[12] N. Pippenger. 1988. Reliable computation by formulae in the presence of noise. *IEEE Trans. Inform. Theor.* 34 (1988), 194–197.

[13] Raymond Reiter. 1987. A theory of diagnosis from first principles. *Artificial Intelligence* 32, 1 (1987), 57–95.

[14] Robert E. Schapire. 1994. Learning probabilistic read-once formulas on product distributions. *Mach. Learn.* 14, 1 (1994), 47–81.

[15] Leslie G. Valiant. 1984. A theory of the learnable. *Commun. ACM* 27, 11 (1984), 1134–1142.

[16] J. von Neumann. 1956. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies,* C. Shannon (Ed.), Vol. 34. Princeton Univ. Press, 43–98.

Received March 2018; revised February 2019; accepted February 2019