Automatically Solving Deduction Games via Symbolic Execution, Model Counting, and Entropy Maximization

Mara Downing, Chris Thompson, Lucas Bang
{mdowning, cbthompson, lbang} @hmc.edu
Harvey Mudd College
301 Platt Blvd.
Claremont, California 91711

Abstract

We present a technique for automatically solving deduction games in which a player makes repeated queries to a running implementation of the game and receives a game outcome, with the goal of discovering an unknown secret value. By making multiple queries, a player iteratively reduces the uncertainty about the secret until it is known. We show how to synthesize player queries using static program analysis, model-counting, and information theory. The system we describe automatically solves deduction games implemented in a Python-based game specification language.

1 Introduction

Deduction games are a form of puzzle in which a player attempts to discover a game solution using logical reasoning. We consider deduction games that proceed in a series of game rounds in which a player makes a query and is provided with an outcome corresponding to that query which reveals some information about an unknown secret value. The player's goal is to discover the secret. Popular games that fall into this category are Mastermind (crack a secret code using information about how similar a query is to the code) and Battleship (find the locations of ships by querying coordinates and learning if they are a 'hit' or 'miss').

In this paper, we present a method of automatically synthesizing queries to solve deduction games. Our approach uses static code analysis, namely *symbolic execution*, to analyze the implementation of a game in order to extract a set of constraints that model the behavior of the game. These constraints are used in a process called '*model counting*', which is leveraged to compute probability distributions relating player queries to game outcomes. The probability distribution functions determine an *information gain objective function* based on Shannon entropy, which, when maximized, yields the optimal play for the current game round. We implemented a domain specific language in which to write deduction games, enabling our static analysis phase. Our experiments demonstrate the effectiveness of our approach on a set of deduction games.

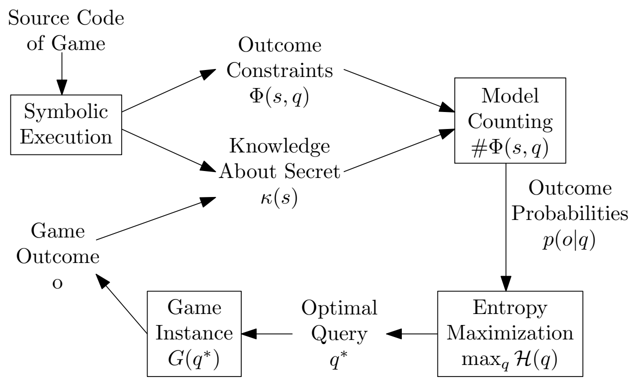
Source Code of Game
Symbolic Execution
Outcome Constraints $\Phi(s,q)$
Knowledge About Secret $\kappa(s)$
Model Counting $\#\Phi(s,q)$
Outcome Probabilities $p(o|q)$
Game Outcome o
Game Instance $G(q^*)$
Optimal Query q^*
Entropy Maximization $\max_q \mathcal{H}(q)$

Figure 1: Overall strategy synthesis approach.

2 Automatically Solving Deduction Games

We give an overview of our approach, including the definitions for our model of deduction games, the steps of automatically solving a game, and the corresponding algorithms. We follow with an example that illuminates each step.

2.1 Components of a Deduction Game

A deduction game G is a tuple: a secret set, a query set, an outcome set, and game rules, respectively, denoted $\langle S, Q, O, R\rangle$.

Secret. The goal of a deduction game is to discover a *secret* value s among a set of all potential secret values S.

Player Queries. A player makes repeated queries q over the course of a game from among a set of possible queries Q.

Outcomes. After each query, the outcome o, from among a set of possible outcomes O, of that game round is revealed.

Game Rules. The outcome of a game round is made according to a deterministic game rule depending on the player query and the secret. We can think of the game rules as a function $R : Q \times S \to O$. We assume that the player knows the rules (code) of the game and the only unknown is s.

2.2 Solution Synthesis Steps and Components

We now describe our solution technique. The reader may find it helpful to refer to Figure 1 along with this discussion.

1. System Input. Source code for a game is provided that specifies (a) the possible secret values and (b) the relationship between player queries, the secret value, and the game outcome. We assume this relationship is specified by a deterministic program. (Our domain specific language for describing games is given in Section 3.1.)

2. Static Code Analysis. We implemented symbolic execution, a form of static program analysis (King 1976), for our game language (see Section 3.2). Symbolic execution returns a set of constraints $\Phi(s, q)$, one per game outcome, encode the relationship between input queries and secret values corresponding to that outcome. Since the game encodes which possible values the secret may take on, this symbolic execution phase also results in a logical constraint capturing the initial "knowledge" about the secret $\kappa(s)$.

3. Probabilities via Model Counting. We seek to compute probability distributions, $p(o|q)$, the probability that a play will result in outcome o given query q. It is sufficient to compute the number of secrets s that satisfy both the player's current knowledge about the secret $\kappa(s)$ and each outcome constraint ϕ_o, all as a function of q, denoted $\#[\phi_o \wedge \kappa(s)](q)$.

Counting solutions to constraints is known as *model counting* and various tools exist for accomplishing this task (Aydin, Bang, and Bultan 2015; Loera et al. 2004; Luu et al. 2014). Here, we used the BARVINOK library, an implementation of Barvinok's polynomial-time lattice point enumeration algorithm. BARVINOK performs model counting by representing a constraint C on variables V as symbolic polytopes $\mathcal{P} \subseteq \mathbb{R}^m$. Barvinok's algorithm generates a multivariate piecewise polynomial that evaluates to the number of assignments of integer values to V that lie in the interior of Q (Barvinok 1994; Verdoolaege 2017).

Now, the probability of an outcome given a player query is easily computed using these model counts: $p(o|q) = \#[\phi_o \wedge \kappa(s)](q)/\#\kappa(s)$, where $\#\kappa(s)$ is the number of secrets consistent with knowledge $\kappa(s)$.

4. Entropy Maximization. Using $p(o|q)$ from the previous step, we compute the conditional Shannon entropy (Shannon 1948; Cover and Thomas 2006) of game outcome O given player query Q. We interpret the conditional Shannon entropy as the amount of information that the player expects to gain by receiving outcome o after making query q:

$$\mathcal{H}(o|q) = -\sum_{o \in O} p(o|q) \log p(o|q)$$

We then maximize $\mathcal{H}(o|q)$ to find a query q^* with the highest expected information gain about the secret s:

$$q^* = \arg\max_{q \in Q} \mathcal{H}(o|q)$$

Note that, in general, $\mathcal{H}(o|q)$ is a multi-dimensional non-convex objective function; locating the true maximum value poses a challenge. Thus, we settle for an approach that searches the query space for the best q^* it can find. Recall that Barvinok represents a constraint C on variables (s, q) as symbolic polytopes $\mathcal{P} \subseteq \mathbb{R}^m$. The result of model counting is a piecewise polynomial function whose domain is a disjunction of polytope chambers $\mathcal{Q} \subseteq \mathbb{R}^m$ represented by linear half-spaces in $\mathbb{R}^m$. We then randomly sample from $\mathcal{Q}$ and evaluate $\mathcal{H}(o|q)$, taking q^* to the maximizing sample.

5. Play Query and Update Knowledge. Query q^* is played producing outcome o according to the game rules. Since each o is associated with a constraint on q and s, the player can update the knowledge about the secret as the conjunction of the current knowledge $\kappa(s)$ with the corresponding observation constraint, replacing the query variable q with the query that was actually played, q^*, denoted $\phi_o[q \mapsto q^*]$:

$$\kappa(s) \leftarrow \kappa(s) \wedge \phi_o[q \mapsto q^*]$$

7. Repeat Until No Information Gain Is Possible. This process repeats in a loop until there are no more queries that can result in positive entropy (information gain).

2.3 Algorithms

The algorithms that specify the behavior of the game and solution synthesis are given here. The process by which entropy is computed is factored into its own procedure.

Algorithm 1 RUNGAME
Input: game $G = \langle S, Q, O, R \rangle$

```
1: procedure RunGame($P$)
2:     $s \leftarrow$ ChooseSecret($S$)
3:     while true do
4:         $q \leftarrow$ PlayerQuery
5:         if WinningCondition($q, s$) then
6:             PlayerWins
7:         else
8:             $o \leftarrow R(q, s)$
9:             RevealToPlayer($o$)
```

Algorithm 2 SOLVEGAME
Input: game program, P.
P is the code for game $G = \langle S, Q, O, R \rangle$.
Output: queries q^* to solve game.

```
1: procedure SolveGame($P$)
2:     $\langle \Phi, \kappa(s) \rangle \leftarrow$ SymbolicExecution($P$)
3:     $\mathcal{H}(o|q) \leftarrow$ ComputeEntropy($\Phi, \kappa(s)$)
4:     while $\exists q[\mathcal{H}(o|q) > 0]$ do
5:         $q^* \leftarrow \arg\max_{q \in Q} \mathcal{H}(o|q)$
6:         $o \leftarrow$ PlayQuery($q^*$)
7:         $\kappa(s) \leftarrow \kappa(s) \wedge \phi_o[q \mapsto q^*]$
8:         $\mathcal{H}(o|q) \leftarrow$ ComputeEntropy($\Phi, \kappa(s)$)
```

Algorithm 3 COMPUTEENTROPY
Input: outcome constraints Φ, knowledge $\kappa(s)$.
Output: Entropy function $\mathcal{H}(o|q)$.

```
1: procedure ComputeEntropy($\Phi, \kappa(s)$)
2:     for each $\phi_o \in \Phi$ do
3:         $\#[\phi_o \wedge \kappa(s)](q) \leftarrow$ Barvinok($\phi_o \wedge \kappa(s)$)
4:     $\#[\kappa(s)] \leftarrow$ Barvinok($\kappa(s)$)
5:     for each $o \in O$ do
6:         $p(o|q) = \#[\phi_o \wedge \kappa(s)](q)/\#\kappa(s)$
7:     $\mathcal{H}(o|q) = -\sum_{o \in O} p(o|q) \log p(o|q)$
8:     return $\mathcal{H}(o|q)$
```

```
function rules(q, s)
  if s < q[0]
    return "Low"
  else if q[0] <= s <= q[1]
    return "Middle"
  else
    return "High"

function chooseSecret(min, max)
  return uniformRandInt(min, max)

function Low-Middle-High(min, max)
  s = chooseSecret(min, max)
  while(true)
    q = playerInput
    if q[0] == s == q[1]
      return "You win!"
    else
      o = rules(q,s)
      print o

main:
  Low-Middle-High(1,27)
```

Figure 2: Pseudocode for the LOW-MIDDLE-HIGH game.

2.4 Example Game: Low-Middle-High

We now walk through the steps just outlined for a simple game that touches on the main points of our approach.

Game Description. Consider a game called LOW-MIDDLE-HIGH in which the secret is an integer within a known range: $1 \leq s \leq 27$. A player proposes pairs of integers $q = (q_0, q_1)$, which we interpret as a lower and upper bound of an integer interval. The player is informed of the value of the secret s relative to the query. That is, the game responds with outcomes according to the `rules` function in the pseudocode of Figure 2: "Low" if $s < q_0$, "Middle" if $q_0 \leq s \leq q_1$, and "High" if neither of those cases apply.

Intuitive Solution Strategy. One might reason that the best strategy for playing this game is to choose (q_0, q_1) at each round such that the search space is cut into equal thirds every time. Indeed, this is the optimal strategy. We will walk through how our approach synthesizes this solution.

Game Parameters. For this game, the secret set is $S = \{s : 1 \leq s \leq 27\}$, the set of potential queries is $Q = \mathbb{Z} \times \mathbb{Z}$, and the set of outcomes is $O = \{$ 'Low', 'Middle', 'High' $\}$, which we abbreviate to $O = \{L, M, H\}$.

Static Code Analysis. Performing symbolic execution (detailed in Section 3.2) of the `rules` function results in the following *outcome constraints*. For this simple example, one can easily see how the constraints correspond to the function that implements the rules of the game.

$$\phi_L = s < q_0, \quad \phi_M = q_0 \leq s \leq q_1, \quad \phi_H = s > q_1$$

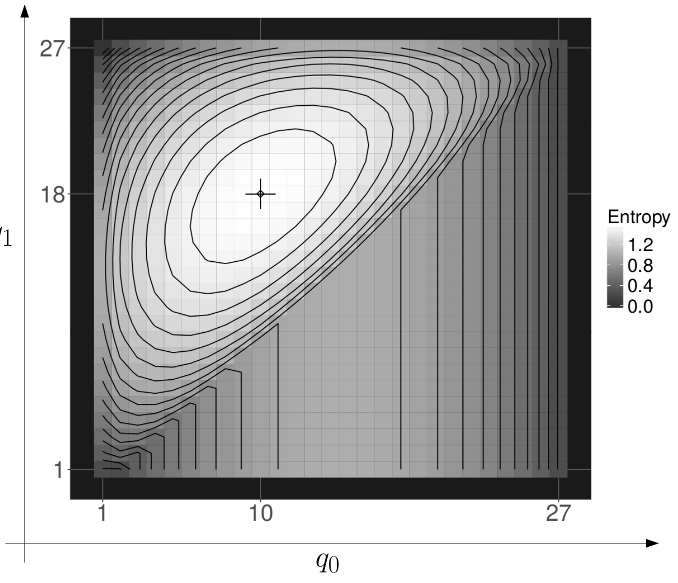

Figure 3: Entropy function (contour plot), $\mathcal{H}(o|(q_0, q_1))$, for the interval game where $1 \leq s \leq 27$. Maximum occurs at $(q_0, q_1) = (10, 18)$, indicated by the cross-hair.

Conditional Outcome Probabilities via Model Counting. We now wish to compute the probabilities of each outcome as a function of the player query. Consider constraint ϕ_L. There are two non-trivial cases. If $q_0 \geq 27$, then there are 27 secret values consistent with the knowledge $\kappa(s) = 1 \leq s \leq 27$. On the other hand, if $1 \leq q_0 < 27$, then there are $q_0 - 1$ solutions for $\kappa(s)$, namely, $s \in \{1, \ldots, q_0 - 1\}$. For any other values of q_0, $\kappa(s)$ is unsatisfiable, so there 0 solutions. Reasoning about $\#\phi_H(q_0, q_1)$ is symmetrically similar, and $\#\phi_M(q_0, q_1)$ is slightly more complicated, giving us the three piecewise counting functions shown here.

$$\#\phi_L(q_0, q_1) = \begin{cases} 27 & q_0 \geq 27 \\ q_0 - 1 & 1 \leq q_0 < 27 \\ 0 & \text{otherwise} \end{cases}$$

$$\#\phi_H(q_0, q_1) = \begin{cases} 27 & q_1 < 0 \\ 27 - q_1 & 0 \leq q_1 < 27 \\ 0 & \text{otherwise} \end{cases}$$

$$\#\phi_M(q_0, q_1) = \begin{cases} 27 & q_0 \leq 1 \wedge q_1 > 27 \\ 28 - q_0 & 1 < q_0 \leq 27 \wedge q_1 > 27 \\ q_1 - q_0 + 1 & q_0 < 1 \wedge q_0 \leq q_1 \leq 27 \\ q_1 & q_0 \leq 1 \wedge 0 < q_1 \leq 27 \\ 0 & \text{otherwise} \end{cases}$$

These counting functions are produced automatically using the Barvinok library. With these counting functions, we compute the probability of each outcome, o, conditioned on the player's query: $p(o|q) = \#[\phi_o \wedge \kappa(s)](q)/\#\kappa(s)$.

Optimal Query via Entropy Maximization. From the outcome probabilities we compute the Shannon entropy using

$p(L|q_0, q_1)$, $p(M|q_0, q_1)$, and $p(H|q_0, q_1)$ by plugging in the appropriate expressions:

$$\mathcal{H}(o|q_0, q_1) = -\sum_{o \in \{L,M,H\}} p(o|q_0, q_1) \log p(o|q_0, q_1)$$

A contour plot of the LOW-MIDDLE-HIGH entropy objective function $\mathcal{H}$ as a function of (q_0, q_1) is given in Figure 3. The point in this space that maximizes $\mathcal{H}$ is the query

$$q^* = \arg\max \mathcal{H}(o|(q_0, q_1)) = (10, 18).$$

This aligns with our earlier intuition that the best strategy is to split the search space in thirds during each round.

Knowledge Update Based on Game Outcome. The player then learns the outcome o of the game when played with query q^* according to the rules: $o = R(q^*, s)$. In our example, a player can learn that $1 \leq s \leq 9$ if the outcome is "Low", $10 \leq s \leq 18$ if the outcome is "Middle", or $19 \leq s \leq 27$ if the outcome is "High".

In our running example, if the player sees outcome $o =$ "Low" after playing query $q^* = (10, 18)$, the update is:

$$\begin{aligned} \kappa(s) &\leftarrow 1 \leq s \leq 27 \wedge \phi_L[(q_0, q_1) \mapsto (10, 18)] \\ &\equiv 1 \leq s \leq 27 \wedge s < 10 \\ &\equiv 1 \leq s \leq 9 \end{aligned}$$

We have demonstrated that entropy maximization based on the constraints generated by static analysis of the game code produces the first step of the optimal *ternary search*. This is a more general principle that applies to more complex games, as seen in our experimental results (Section 4).

Query Until Information Is Exhausted. Given updated $\kappa(s)$, the process repeats, starting with model counting. Note that the static code analysis phase is not repeated, as the symbolic execution constraints sufficiently capture the behavior of the game. Supposing that the knowledge is updated as described in the previous step, the next round of query synthesis using model counting and entropy maximization results in $q^* = (4, 6)$. This continues until there are no queries with positive information gain, i.e. $\forall q[\mathcal{H} > 0]$.

3 Implementation

We implemented the approach of Section 2 by implementing Algorithms 1, 2, and 3 in Python and interfacing with Z3 for constraint satisfiability checking during symbolic execution and Barvinok for model counting. There are two implementation aspects that deserve further explication, described in this section: our domain specific language for encoding deduction games and how we perform symbolic execution.

3.1 Domain Specific Language

Our approach relies on extracting a logical representation of the game from the source code of the game. In order to facilitate this, we designed a small language for encoding games that strikes a balance between analyzability and expressiveness. The language has features that we found necessary to express deduction games but is simple enough that writing static analysis routines and interfacing with the Z3 constraint solver under the hood is straightforward. The interpreter of our language is written in Python and supports basic imperative programming features including Boolean and integer operations, control and iteration structures, functions, and arrays. See Figure 4 for the abstract grammar.

```
 Program  :=  List(Stmt)
    Stmt  :=  Stmt; Stmt
           |  If(Exp, Stmt)
           |  While(Exp, Stmt)
           |  Assign(Id, Exp)
           |  ArrayStore(Exp, Stmt)
           |  Return(Exp)
     Exp  :=  BoolExp
           |  IntExp
           |  ArrayDeclare(Id, IntExp)
           |  ArrayAccess(Id, IntExp)
           |  Function(List(Id), Stmt)
           |  FunctionCall(Id, List(Exp))
 BoolExp  :=  true | false
           |  And(BoolExp, BoolExp)
           |  Or(BoolExp, BoolExp)
           |  Not(BoolExp)
           |  Less(IntExp, IntExp)
           |  Equal(IntExp, IntExp)
  IntExp  :=  IntConst
           |  Plus(IntExp, IntExp)
           |  Times(IntConst, IntExp)
IntConst  :=  c ∈ ℤ
```

Figure 4: Domain specific language abstract grammar for specifying deduction games, supporting basic imperative constructs, Boolean and integer operations, arrays, and functions with the standard expected semantics.

3.2 Symbolic Execution

We extract a symbolic model of the game in the form of a set of outcome constraints using symbolic execution. Symbolic execution (King 1976) is a popular static code analysis technique by which a program is executed on *symbolic* (as opposed to concrete) input values that represent all possible concrete values. In the limit, symbolic execution explores all feasible paths of program execution.

Symbolically executing a program yields a set of *path constraints* $\Psi = \{\psi_1, \psi_2, \ldots, \psi_n\}$. Each ψ_i is a conjunction of constraints on the symbolic inputs that characterize all concrete inputs that would cause a path to be followed. All the ψ_i are disjoint. Whenever symbolic execution hits a branch condition c, both branches are explored and the constraint is updated: $\psi \leftarrow \psi \wedge c$ in the *true* branch and $\psi \leftarrow \psi \wedge \neg c$ in the *false* branch. Path constraint satisfiability is checked using constraint solvers such as Z3 (De Moura and Bjørner 2008). If a path constraint is found to be unsatisfiable, that path pruned from the symbolic exploration.

We treat the secret value s and the input query q as symbolic, and we associate each path constraint with the corresponding game outcome. Thus, during symbolic execution, we track the return values of functions that implement the rules of the game, and each ψ_i is associated with a concrete

```
function rules(q,s)
  for(i = 0; i < 3; i++)
    if(q[i] != s[i])
      return i
  return i
```

Figure 5: Code snippet for example of symbolic execution.

game outcome o_i. Path constraints that result in the same outcome are combined using disjunction to produce the *outcome constraints*:

$$\phi_o = \bigvee_{o=o_i} \psi_i$$

To deal with loops in symbolic execution, a bound is typically enforced on exploration depth. In our setting, we found that we did not have to artificially bound symbolic execution, and the path constraints we generate correspond to a complete characterization of the programmed game rules.

Symbolic Execution Example. Consider a simple game in which a player attempts to learn a secret integer represented as an array of digits between 0 and 9. The player proposes an integer query and it is compared digit by digit to the secret. The outcome of the game is the length of the matching prefix. The logic of the game outcome is modeled by the pseudocode of Figure 5. For this simple example, we assume the query and secret are 3 digits each.

We model the arrays q and s as arrays of symbolic integers: $[q_0, q_1, q_2]$ and $[s_0, s_1, s_2]$. Symbolically executing the code results in the symbolic execution tree of Figure 6. Path constraints ϕ are maintained during symbolic execution, but for simplicity we only show the path constraints that are obtained at the leaves of the tree. First, i is set to 0, and then we symbolically explore both branches of the condition $i < 3$: the constraints $i = 0 \wedge \neg(i < 3)$ and $i = 0 \wedge i < 3$ are both sent to the Z3 theorem prover for satisfiability checking. The false branch is determined to be unsatisfiable and so is not explored. The true branch is satisfiable and so exploration continues into the loop. Next, constraints for both possible branches of execution of the if-statement are sent to Z3: $i = 0 \wedge i < 3 \wedge \neg(q_0 \neq s_0)$ and $i = 0 \wedge i < 3 \wedge q_0 \neq s_0$. Both branches are determined to be satisfiable. The true branch executes a return statement, which returns outcome $o = 0$, so we have $\phi_0 : q_0 \neq s_0$. In the other branch, i is incremented and exploration continued, resulting in the constraints illustrated in the tree.

4 Experiments

We ran our implementation on several variants of well-known logical deduction games. We give a brief description of each game, followed by the experimental results.

4.1 Logical Deduction Games

We coded each of the following games in the domain specific language of Section 3.1.

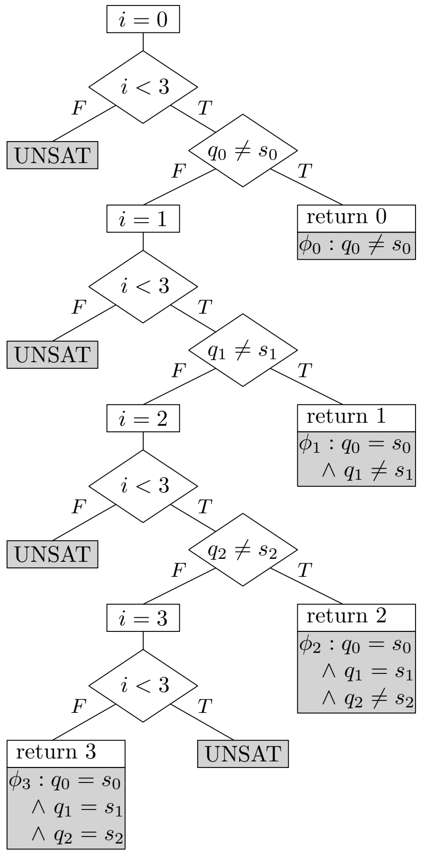

Figure 6: Symbolic Execution Tree of Figure 5.

Low-Middle-High. This is the example game of Section 2.

Mastermind. The player tries to find a secret code consisting of 4 colored pegs, where each peg can be one of 6 colors, by proposing their own 4-color code. The game responds by giving a number of red flags (the number of pegs in the correct positions with the correct color) and a number of white flags (the number of pegs with correct colors but in the incorrect positions) (Kooi 2005).

Simple Mastermind. This is a simplified version of Mastermind where only red flags are revealed.

Password Cracker. The player guesses a list of integers from 0 to 9 to try to discover a secret password consisting of digits. The game responds by giving the length of the prefix of the user query which matches the secret. This game models a hacker attempting to learn a password or key using a segment oracle attack (Bang et al. 2016).

Counterfeit Coin. This is the classic counterfeit coin problem. There are n coins that look identical, one of which is either slightly lighter or heavier than the others. The player can place any number of coins on either side of a scale, which tilts left, tilts right, or balances (Smith 1947).

Sushi. A player is attempting to determine the sushi pref-

Table 1: Experimental results showing solving time, number of rounds, symbolic execution time, number of path constraints $|\Psi|$, number of observation constraints $|\phi|$, size of the query space $|Q|$, and size of the secret search space $|S|$.

Game	Details	Average Solve Time (s)	Average # Rounds	Symbolic Exec. Time (s)	$\|\Psi\|$	$\|\Phi\|$	$\|Q\|$	$\|S\|$
Counterfeit Coin	6 coins	33.182	2.44	3.278	36	3	729	12
Counterfeit Coin	9 coins	2072.795	3	7.146	54	3	8952	18
Mastermind	6 colors 4 pegs	8185.924	3.8	38.414	209	14	1296	1296
Simple Mastermind	6 colors 4 pegs	248.278	6.2	0.77	16	5	1296	1296
Low-Middle-High	from 1 to 10	2.727	5	0.033	3	3	81	45
Low-Middle-High	from 1 to 50	24.643	10.1	0.032	3	3	2401	1225
Low-Middle-High	from 1 to 100	89.158	11.9	0.032	3	3	9801	4950
Sushi	5 options	188.955	6.933	2.381	20	2	25	120
Password Cracker	4 digits	27.235	15.75	0.15	5	5	10000	10000
Password Cracker	6 digits	501.127	28.55	0.246	7	7	1000000	1000000
Horse Race	5 horses 3 lanes	599.923	3.5	68.132	60	6	125	120
Battleship	8x8 grid	17.984	11.333	0.115	4	2	64	96
Battleship	12x12 grid	123.38	27.4	0.116	4	2	144	240

erences of a customer out of 5 possible types. The player offers two sushi dishes at a time. The customer tries each of them and says which of the two they prefer. The player hopes to discover the complete ranking of sushi dishes for the customer. This is a gamified version of the ranking via pairwise comparison problem (Jamieson and Nowak 2011) and is similar to a machine learning task in the existing literature (Pu, Kaelbling, and Solar-Lezama 2017).

Horse Race. A player is attempting to discover the fastest 3 horses from among 5 horses. The player is allowed to race 3 horses at a time and finds out the race winner. This is a variation of a Google interview question (Talwalkar 2017).

Simple Battleship. The player guesses two integers, coordinates in a grid of cells, attempting to sink a ship that takes up 3 vertically or horizontally adjacent cells. The game responds by saying whether the player has hit the hidden ship at those coordinates. This is a simplified version of the popular Battleship game, in which there are 5 ships of different sizes (J. M. Meuffels and den Hertog 2010).

4.2 Experimental Results

The results of running our automatic game solving approach are summarized in Table 1. We solved each game 5 times for 5 different, randomly chosen secret values. We report the number of path conditions $|\Psi|$, number of possible game outcomes per round, or equivalently, the number of observation constraints $|\Phi|$ after disjunctive merging (Section 3.2), size of the query space $|Q|$, and size of the secret search space $|S|$. We also report averages of the symbolic execution time, game solving time, and number of rounds required to solve the game.

The three most challenging games were Mastermind, Counter Coin, and Horse Race, taking approximately 140 minutes, 34 minutes, and 10 minutes respectively. We observe that the time required for static analysis is always under 1 minute, except in the case of Horse Race, which is barely over a minute. The most expensive operations of the game solving phase are the model counting done by BARVINOK and then maximizing the resulting entropy function.

5 Related Work

Our approach to solving deduction games is strongly influenced by work in the intersection of computer security, software verification, and quantitative information flow, where the goal is to show that a software systems is potentially vulnerable to attackers by synthesizing inputs that reveal sensitive information. Often, these scenarios are framed as a game between an adversary and a computer system. These works typically make use of static code analysis, constraint solving, model counting, and information theory (Klebanov 2014; Phan et al. 2017; Saha et al. 2018; Clarkson, Myers, and Schneider 2005). This paper is an attempt to port those techniques to the world of games.

There is substantial work focused individually on two of the games we consider, Mastermind and variants (Goodrich 2012; Kooi 2005; Maestro-Montojo, Salcedo-Sanz, and Merelo Guervs 2014) and scale and coin problems (Khovanova 2013; M. Chudnov 2015). A 2017 paper discusses synthesizing human-interpretable strategies for the Nonogram Game (Butler, Torlak, and Popovic 2017). There is an interesting approach that uses training data and neural networks along with entropy maximization (Pu, Kaelbling, and Solar-Lezama 2017).

The most closely related work is that of COde-BReaking game Analyzer (COBRA), which relies on a logical specification of a game and uses constraint solvers and entropy maximization to sol ve code-breaking games (Klimos and Kucera 2015). The main difference with our approach is that we analyze the source code of a game rather than an encoding in logic, and we generate symbolic model counting expressions, whereas COBRA enumerates all constraint solutions in order to count models.

6 Conclusions and Future Work

We demonstrated that our approach is effective at automatically solving deduction games using static code analysis, model counting, and entropy maximization. Efficiency and scalability of the approach can be improved. Because evalautions of the entropy function for different queries are independent, we intend to investigate approaches that leverage parallel objective function evaluation. In addition, because metaheuristic techniques such as genetic algorithms have shown success in solving specific games, we hope to explore their use within our framework. The main takeaway of this work is that given only the code that implements a deduction game, we are able to automatically solve it.

Acknowledgments. This research was supported by NSF award #1659805: Harvey Mudd Research Experience for Undergraduates Site in Computer Systems.

References

Aydin, A.; Bang, L.; and Bultan, T. 2015. Automata-based model counting for string constraints. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*, 255–272.

Bang, L.; Aydin, A.; Phan, Q.-S.; Pasareanu, C. S.; and Bultan, T. 2016. String analysis for side channels with segmented oracles. In *Proc. of the 24th ACM SIGSOFT International Symp. on the Foundations of Software Engineering*.

Barvinok, A. I. 1994. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.* 19(4):769–779.

Butler, E.; Torlak, E.; and Popovic, Z. 2017. Synthesizing interpretable strategies for solving puzzle games. In *Proceedings of the International Conference on the Foundations of Digital Games, FDG 2017, Hyannis, MA, USA, August 14-17, 2017*, 10:1–10:10.

Clarkson, M. R.; Myers, A. C.; and Schneider, F. B. 2005. Belief in information flow. In *Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, CSFW '05, 31–45. Washington, DC, USA: IEEE Computer Society.

Cover, T. M., and Thomas, J. A. 2006. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience.

De Moura, L., and Bjørner, N. 2008. Z3: an efficient SMT solver. In *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, 337–340.

Goodrich, M. T. 2012. Learning character strings via mastermind queries, with a case study involving mtdna. *IEEE Trans. Information Theory* 58(11):6726–6736.

J. M. Meuffels, W., and den Hertog, D. 2010. Puzzle solving the battleship puzzle as an integer programming problem. *Journal of Financial Stability* 10:156–162.

Jamieson, K. G., and Nowak, R. D. 2011. Active ranking using pairwise comparisons. In *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain.*

Khovanova, T. 2013. Parallel weighings. *arXiv preprint arXiv:1310.7268.*

King, J. C. 1976. Symbolic execution and program testing. *Commun. ACM* 19(7):385–394.

Klebanov, V. 2014. Precise quantitative information flow: a symbolic approach. *Theor. Comput. Sci.* 538:124–139.

Klimos, M., and Kucera, A. 2015. Cobra: A tool for solving general deductive games. In *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, November 24-28*, 31–47.

Kooi, B. P. 2005. Yet another mastermind strategy. *ICGA Journal* 28(1):13–20.

Loera, J. A. D.; Hemmecke, R.; Tauzer, J.; and Yoshida, R. 2004. Effective lattice point counting in rational convex polytopes. *Journal of Symbolic Computation* 38(4):1273 – 1302. Symbolic Computation in Algebra and Geometry.

Luu, L.; Shinde, S.; Saxena, P.; and Demsky, B. 2014. A model counter for constraints over unbounded strings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 57.

M. Chudnov, A. 2015. Weighing algorithms of classification and identification of situations. *Discrete Mathematics and Applications* 25.

Maestro-Montojo, J.; Salcedo-Sanz, S.; and Merelo Guervs, J. 2014. New solver and optimal anticipation strategies design based on evolutionary computation for the game of mastermind. *Evolutionary Intelligence* 6.

Phan, Q.; Bang, L.; Pasareanu, C. S.; Malacaria, P.; and Bultan, T. 2017. Synthesis of adaptive side-channel attacks. *IACR Cryptology ePrint Archive* 2017:401.

Pu, Y.; Kaelbling, L. P.; and Solar-Lezama, A. 2017. Learning to acquire information. In *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence, UAI 2017, Sydney, Australia, August 11-15, 2017.*

Saha, S.; Kadron, I. B.; Eiers, W.; Bang, L.; and Bultan, T. 2018. Attack synthesis for strings using meta-heuristics. *ACM SIGSOFT Software Engineering Notes* 43(4):56.

Shannon, C. 1948. A mathematical theory of communication. *Bell System Technical Journal* 27:379–423, 623–656.

Smith, C. A. B. 1947. The counterfeit coin problem. *The Mathematical Gazette* 31(293):31–39.

Talwalkar, P. 2017. Can you solve the 25 horses puzzle? google interview question.

Verdoolaege, S. 2017. The Barvinok model counter.