In [5]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import GradientBoostingClassifier
import random
from scipy.ndimage.filters import gaussian_filter1d
from scipy.interpolate import UnivariateSpline
import warnings
warnings.filterwarnings('ignore')
```

1. Изучите классическую работу У. Рурмаира о криптографических атаках с помощью машинного обучения на ФНФ.

2. Сформулируйте задачу в терминах машинного обучения.

При применении к ФНФ с однобитовыми выходами каждому вызову $C = b^k \cdots b^k$ назначается вероятность $p(C, t|\vec{w})$ такая, что он генерирует выходной сигнал $t \in \{-1, 1\}$. Таким образом, вектор $\vec{w}$ кодирует соответствующие внутренние параметры, например, конкретные задержки времени выполнения отдельной ФНФ. Вероятность определяется логистической сигмоидой, действующей на функцию $f(\vec{w})$, параметризованную вектором $\vec{w}$ как $p(C, t|\vec{w}) = \sigma(tf) = (1 + e^{-tf})^{-1}$. Таким образом, f через $f = 0$ определяет границу решения с равными выходными вероятностями. Для данного обучающего набора M из CRP необходимо определить границу путем выбора вектора параметров $\vec{w}$ таким образом, чтобы вероятность наблюдения этого набора была максимальной, соответственно, отрицательная логарифмическая вероятность была минимальной:
$w = argmin_{\vec{w}} l(M, \vec{w}) = argmin_{\vec{w}} X(C, t) \in M - ln(\sigma(tf(\vec{w}, C)))$

3. Обучите модель, которая могла бы предсказывать ответы по запросам, которых нет в обучающей выборке.

In [6]:
```
def into_features_vect(chall):
    "Transforms a challenge into a feature vector"
    phi = []
    for i in range(1,len(chall)):
        s = sum(chall[i:])
        if s % 2 == 0:
            phi.append(1)
        else:
            phi.append(-1)
    phi.append(1)
    return phi
```

In [7]:
```
class Stage:
    _delay_out_a = 0.
    _delay_out_b = 0.
    _selector = 0

    def __init__(self,delay_a,delay_b):
        self._delay_out_a = delay_a
        self._delay_out_b = delay_b

    def set_selector(self,s):
        self._selector = s

    def get_output(self,delay_in_a, delay_in_b):
        if self._selector == 0:
            return (delay_in_a  + self._delay_out_a,
                    delay_in_b  + self._delay_out_b)
        else:
            return (delay_in_b  + self._delay_out_a,
                    delay_in_a  + self._delay_out_b)
```

In [8]:
```
class ArbiterPUF:

    def __init__(self,n):
        self._stages = []

        for _ in range(n):
            d1 = random.random()
            d2 = random.random()
            self._stages.append(Stage(d1,d2))

    def get_output(self,chall):
        # Set challenge
        for stage,bit in zip(self._stages,chall):
            stage.set_selector(bit)

        # Compute output
        delay = (0,0)
        for s in self._stages:
            delay = s.get_output(delay[0],delay[1])

        if delay[0] < delay[1]:
            return 0
        else:
            return 1
```

In [9]:

```
class Stage:
    _delay_out_a = 0.
    _delay_out_b = 0.
    _selector = 0

    def __init__(self,delay_a,delay_b):
        self._delay_out_a = delay_a
        self._delay_out_b = delay_b

    def set_selector(self,s):
        self._selector = s

    def get_output(self,delay_in_a, delay_in_b):
        if self._selector == 0:
            return (delay_in_a  + self._delay_out_a,
                    delay_in_b  + self._delay_out_b)
        else:
            return (delay_in_b  + self._delay_out_a,
                    delay_in_a  + self._delay_out_b)

class ArbiterPUF:

    def __init__(self,n):
        self._stages = []

        for _ in range(n):
            d1 = random.random()
            d2 = random.random()
            self._stages.append(Stage(d1,d2))

    def get_output(self,chall):
        # Set challenge
        for stage,bit in zip(self._stages,chall):
            stage.set_selector(bit)

        # Compute output
        delay = (0,0)
        for s in self._stages:
            delay = s.get_output(delay[0],delay[1])

        if delay[0] < delay[1]:
            return 0
        else:
            return 1

N  = 32     # Size of the PUF
LS = 600    # Size learning set
TS = 10000  # Size testing set
apuf = ArbiterPUF(N)

# Creating training suite
learningX = [[random.choice([0,1]) for _ in range(N)] for _ in range(LS)] # Ch
allenges
learningY = [apuf.get_output(chall) for chall in learningX] # Outputs PUF

# Creating testing suite
```

```
testingX = [[random.choice([0,1]) for _ in range(N)] for _ in range(TS)]
testingY = [apuf.get_output(chall) for chall in testingX]

# Convert challenges into feature vectors
learningX = [into_features_vect(c) for c in learningX]
testingX = [into_features_vect(c) for c in testingX]

# Prediction
lr = LogisticRegression()
lr.fit(learningX, learningY)
print("Score arbiter PUF (%d stages): %f" % (N,lr.score(testingX,testingY)))
```

```
Score arbiter PUF (32 stages): 0.963800
```

4. Применить как минимум 3 различных алгоритма (например, метод опорных векторов, логистическая регрессия и градиентный бустинг).

In [10]:
```
svc = SVC()
svc.fit(learningX, learningY)
print("Score arbiter PUF (%d stages): %f" % (N, svc.score(testingX, testingY
)))
```

```
Score arbiter PUF (32 stages): 0.925700
```

In [11]:
```
gb = GradientBoostingClassifier()
gb.fit(learningX, learningY)
print("Score arbiter PUF (%d stages): %f" % (N, gb.score(testingX, testingY)))
```

```
Score arbiter PUF (32 stages): 0.857900
```

5. Какая метрика наиболее подходит для оценки качества алгоритма?

Количество правильных ответов обученной модели на тестовом наборе, деленное на количество CRP в тестовом наборе.

6. Какой наибольшей доли правильных ответов (Accuracy) удалось достичь?

Наибольшей доли правильных ответов удалось достичь используя алгоритм линейной регрессии. Его доля правильных ответов (на 32-битном арбитре) составила 0.876 Для сравнения: SVM = 0.797, GB = 0.749

7. Какой размер обучающей выборки необходим, чтобы достигнуть доли правильных ответов минимум 0.95?

Чтобы достичь 95% доли правильных ответов, необходимо примерно 640 входных значений в случае с 64-битным Арбитром и примерно 1350 входных значений в случае с 128-битным Арбитром.

8. Как зависит доля правильных ответов от N?

Доля правильных ответов прямопропорционально зависит от количества входных пар запрос-ответ.

In [12]:
```
learning_set_numbers = np.arange(100, 1000, 50)
scores = []

for i in learning_set_numbers:
    N  = 32     # Size of the PUF
    LS = i    # Size learning set
    TS = 10000  # Size testing set
    apuf = ArbiterPUF(N)

    # Creating training suite
    learningX = [[random.choice([0,1]) for _ in range(N)] for _ in range(LS)]
# Challenges
    learningY = [apuf.get_output(chall) for chall in learningX] # Outputs PUF

    # Creating testing suite
    testingX = [[random.choice([0,1]) for _ in range(N)] for _ in range(TS)]
    testingY = [apuf.get_output(chall) for chall in testingX]

    # Convert challenges into feature vectors
    learningX = [into_features_vect(c) for c in learningX]
    testingX = [into_features_vect(c) for c in testingX]

    # Prediction
    lr = LogisticRegression()
    lr.fit(learningX, learningY)
    scores.append(lr.score(testingX, testingY))

plt.plot(learning_set_numbers, scores, LineWidth=2)
plt.ylabel('Accuracy')
plt.xlabel('N')
```

Out[12]:
```
Text(0.5, 0, 'N')
```