26th USENIX Security Symposium

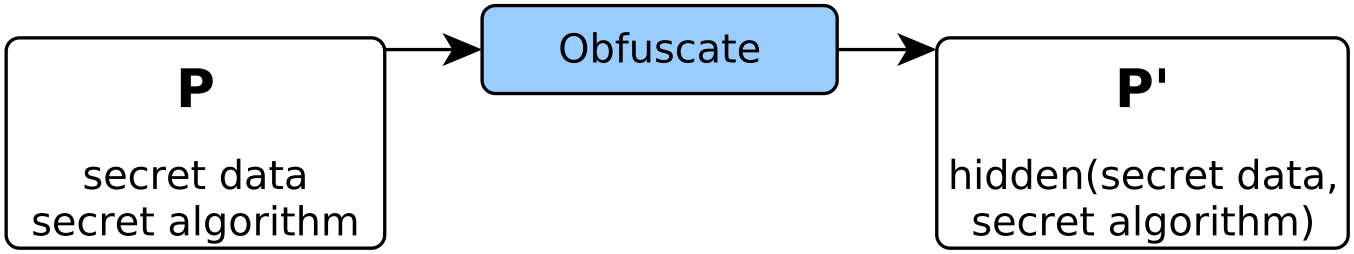
Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning

Sebastian Banescu
Christian Collberg
Alexander Pretschner

Introduction

Informal Definition of Obfuscation

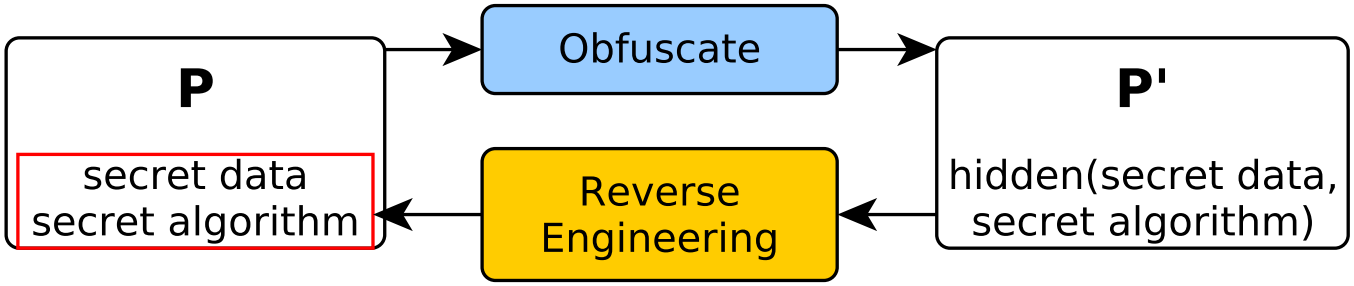
To obfuscate a program P means to transform it into an equivalent program P' from which it is harder to extract information than from P.



Introduction

Informal Definition of Obfuscation

To obfuscate a program P means to transform it into an equivalent program P' from which it is harder to extract information than from P.

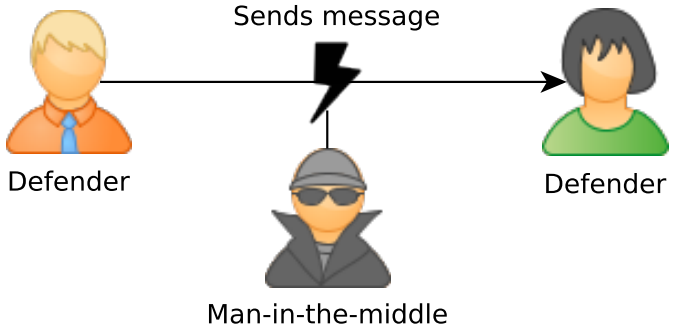


Informal Definition of Reverse Engineering

The process of extracting data or a model of the system by inspecting its lower level description and/or behavior.

Attacker Models in IT Security

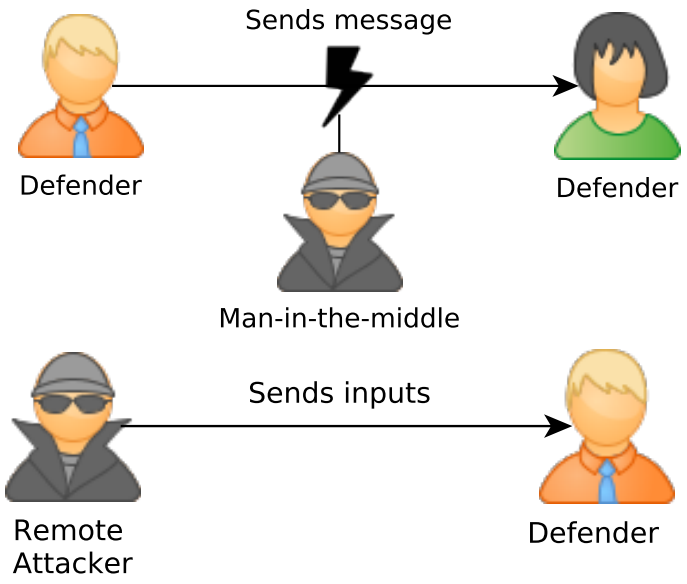
1. **Man-in-the-middle (MITM)** attacks communication channels



Attacker Models in IT Security

1. **Man-in-the-middle (MITM)** attacks communication channels

2. **Remote attacker** exploits vulnerabilities (e.g. buffer overflows)

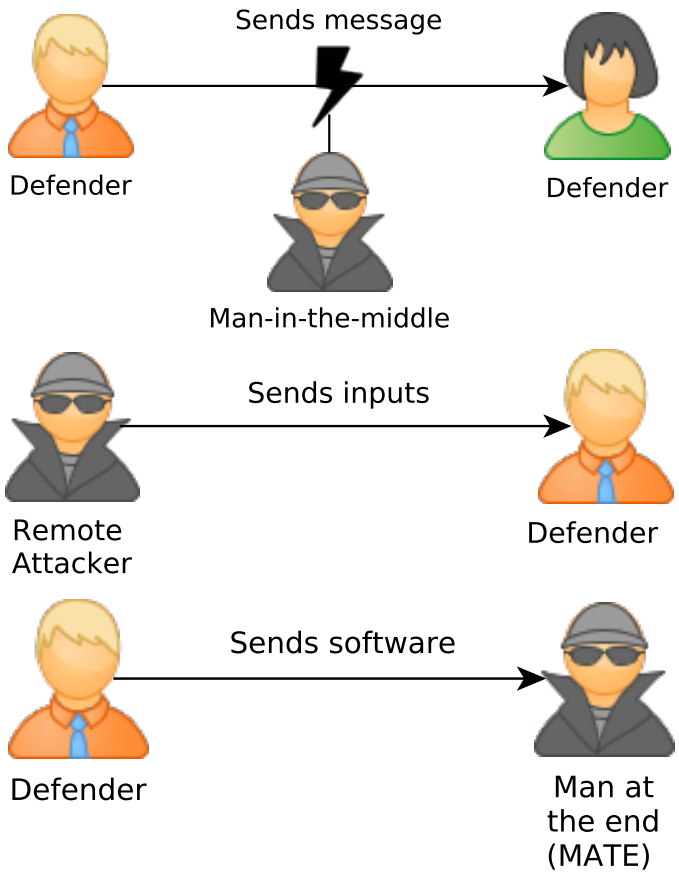


Attacker Models in IT Security

1. **Man-in-the-middle (MITM)** attacks communication channels

2. **Remote attacker** exploits vulnerabilities (e.g. buffer overflows)

3. **Man-at-the-end (MATE)** reverse engineers software

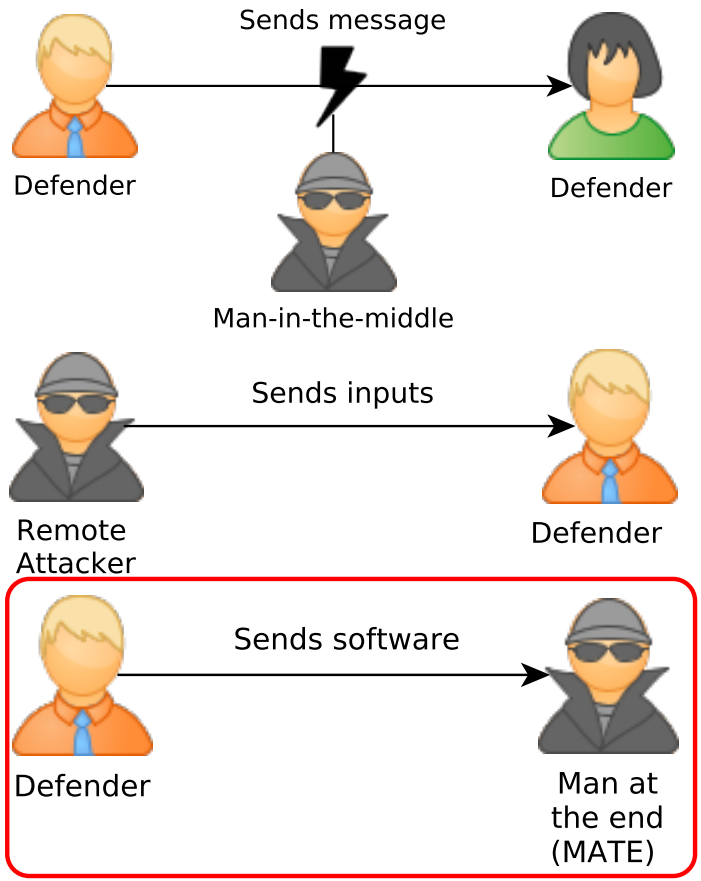


Attacker Models in IT Security

1. **Man-in-the-middle (MITM)** attacks communication channels

2. **Remote attacker** exploits vulnerabilities (e.g. buffer overflows)

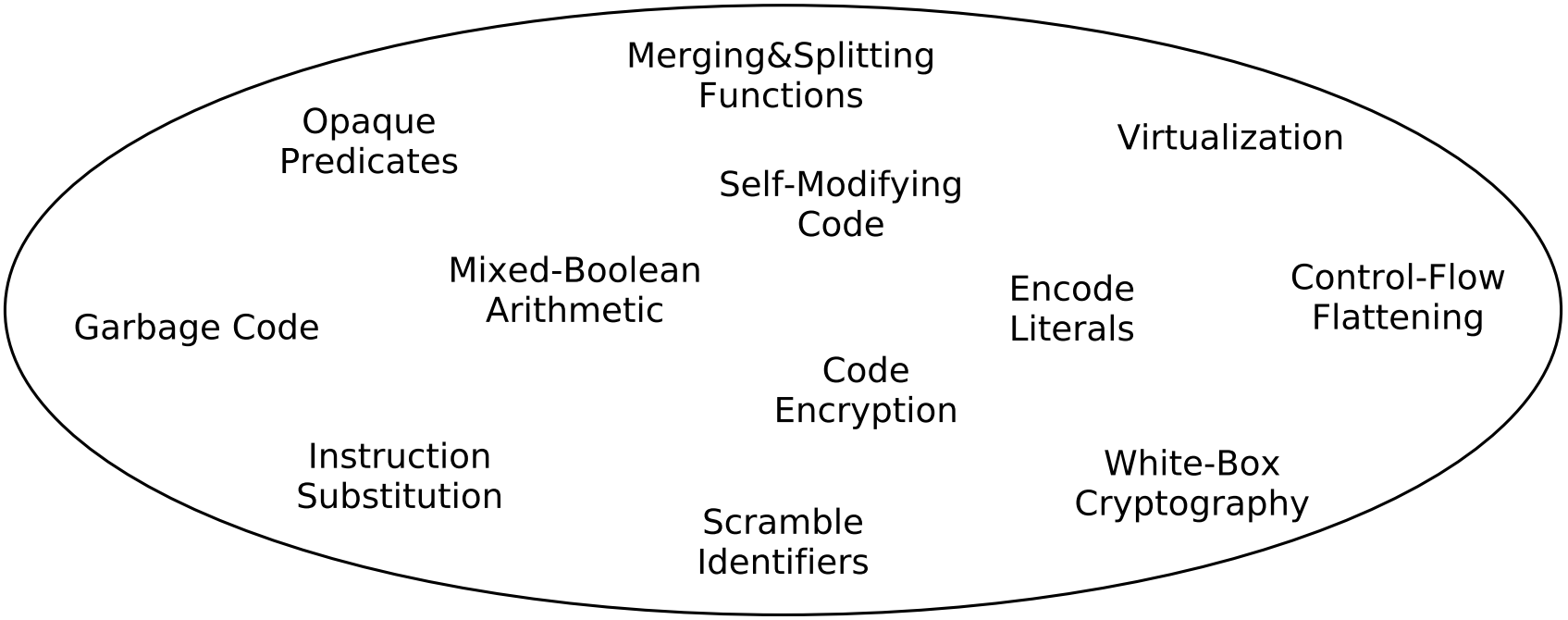
3. **Man-at-the-end (MATE)** reverse engineers software



Motivation

Motivational Example

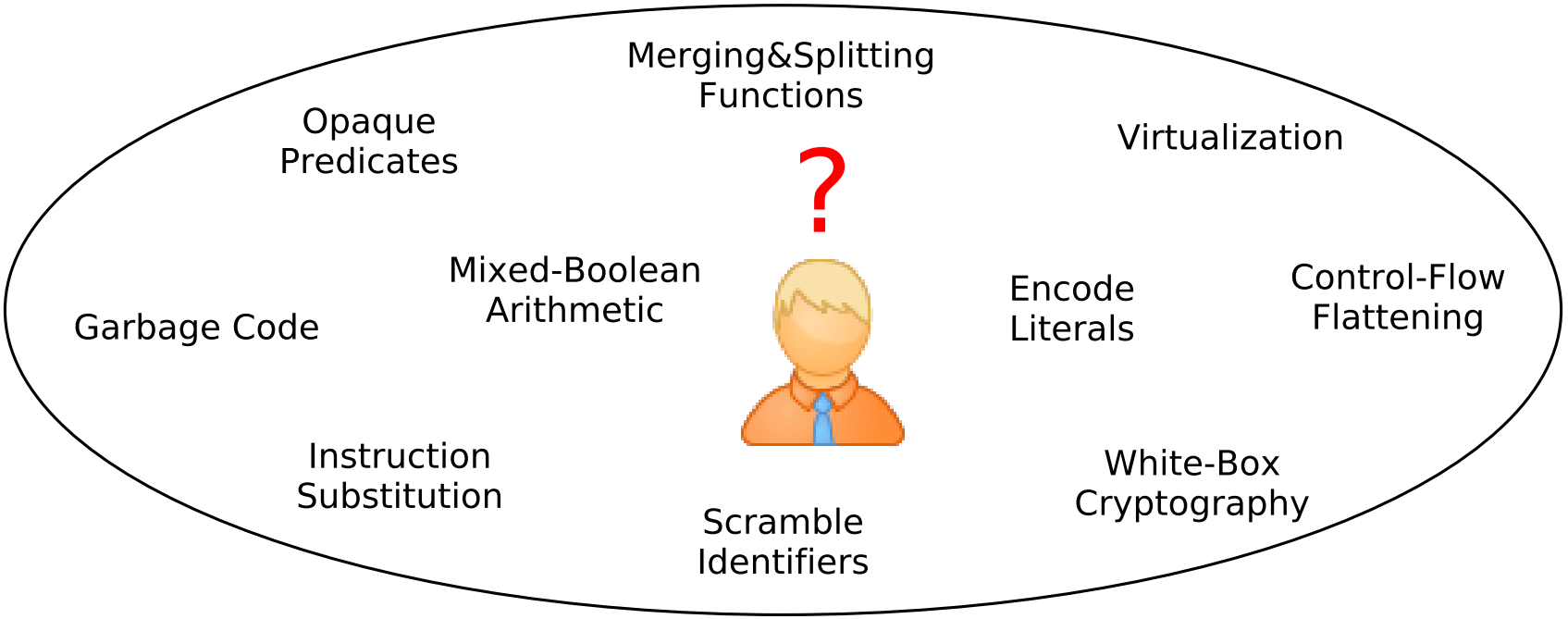
- **Problem:** Software developer wants to protect software for 100 days



Motivation

Motivational Example

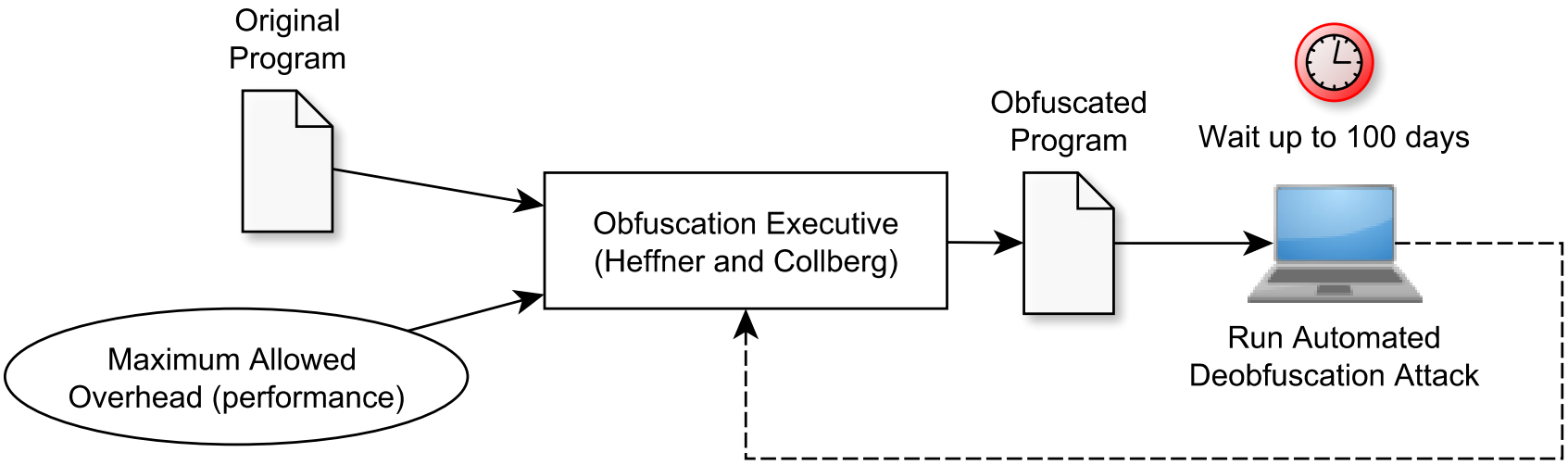
- **Problem:** Software developer wants to protect software for 100 days



Motivation

Motivational Example

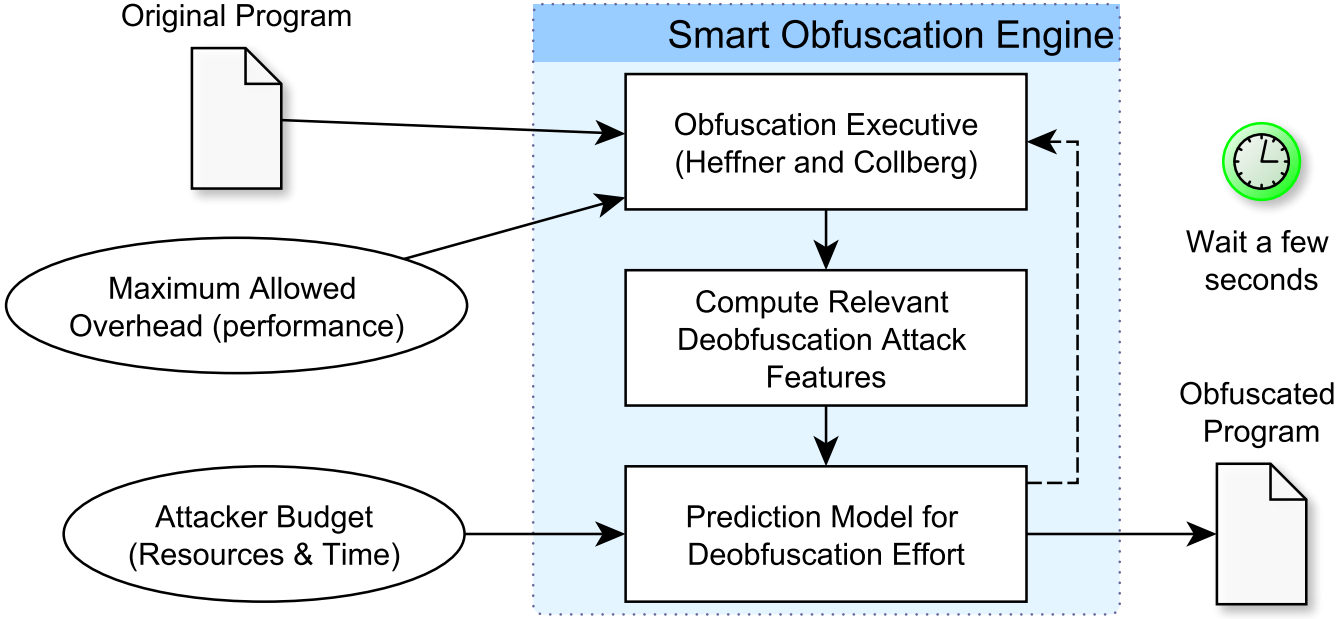
- **Problem:** Software developer wants to protect software for 100 days
- **Current approach:** Obfuscate program and run attack (slow)



Motivation

Motivational Example

- **Problem:** Software developer wants to protect software for 100 days
- **Current approach:** Obfuscate program and run attack (slow)
- **Our approach:** Obfuscate program and predict effort (fast)



Research Questions Answered in This Paper

1. How can we predict the effort of automated MATE attacks?

Research Questions Answered in This Paper

1. How can we predict the effort of automated MATE attacks?

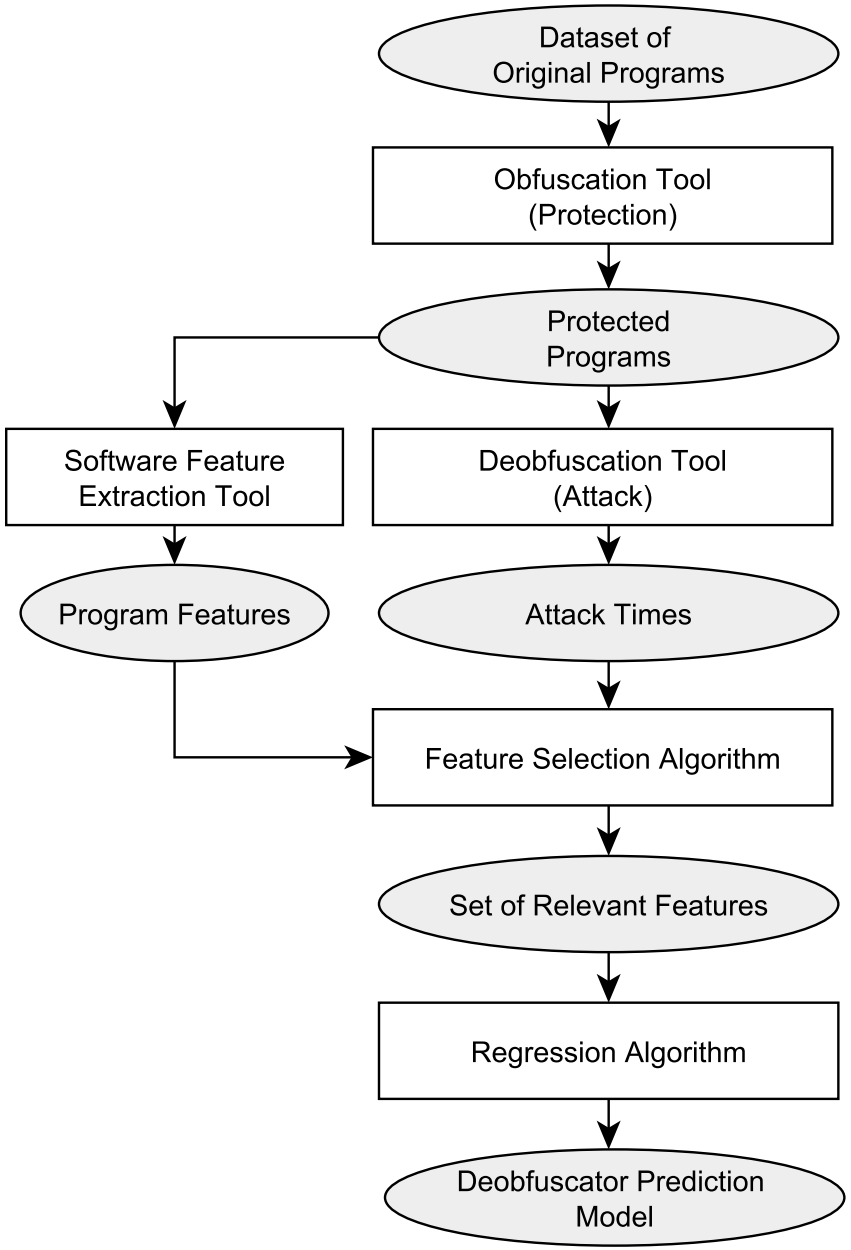
2. Which code features are most relevant for predicting the time needed to successfully run an attack based on symbolic execution?

Research Questions Answered in This Paper

1. How can we predict the effort of automated MATE attacks?
2. Which code features are most relevant for predicting the time needed to successfully run an attack based on symbolic execution?
3. Which regression algorithms generate models that can predict the attack effort with the lowest error?

Overview of Methodology Steps

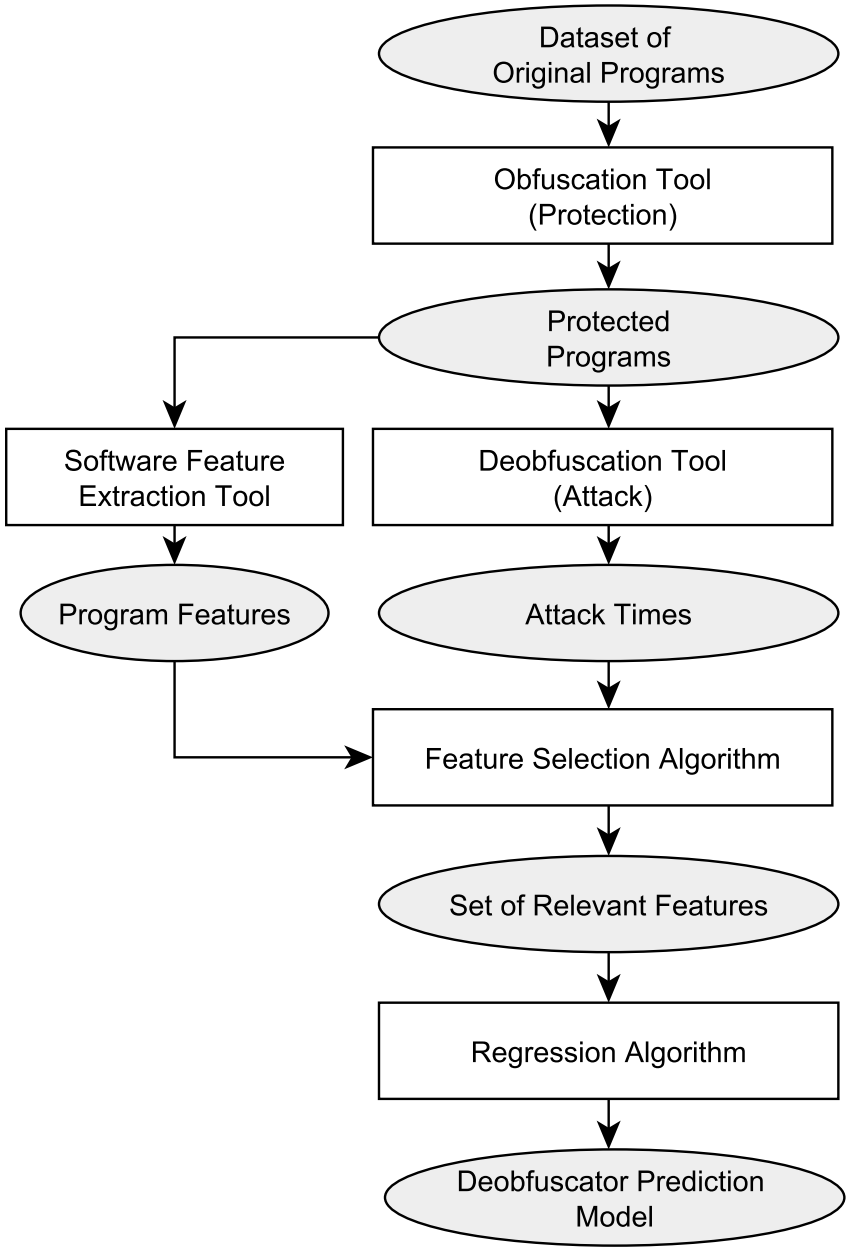
1. Create large dataset of original programs for training
2. Obfuscate programs
3. Attack obfuscated programs
4. Feature extraction
5. Create prediction model for deobfuscation effort



Overview of Methodology Steps

1. Create large dataset of original programs for training
2. Obfuscate programs
3. Attack obfuscated programs
4. Feature extraction
5. Create prediction model for deobfuscation effort

Case study: deobfuscation attack based on symbolic execution



Symbolic Execution in a Nutshell

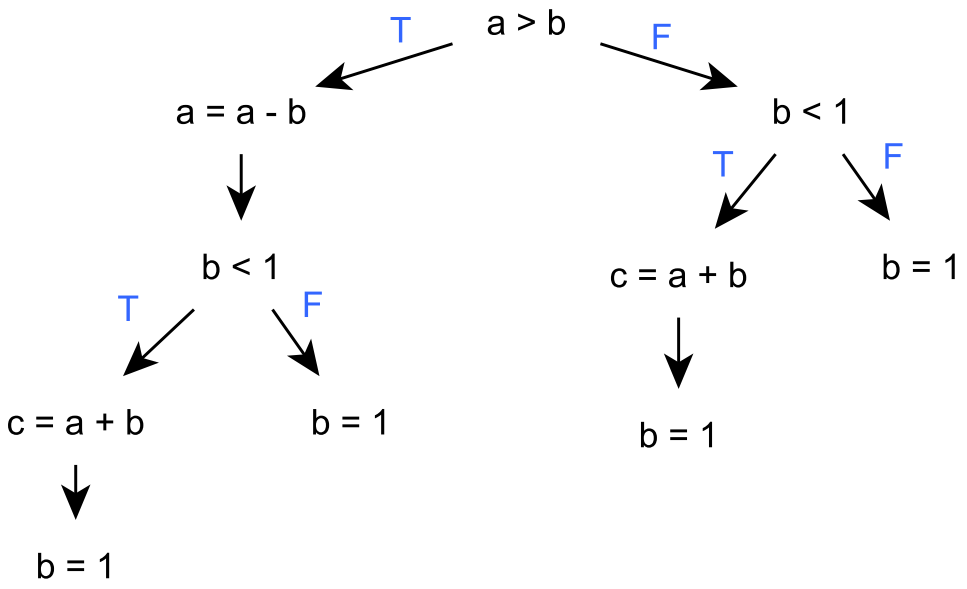
- Interpret program using symbolic values instead of concrete ones

```
int main(int ac, char* av[]) {
  int a = atoi(av[1]);
  int b = atoi(av[2]);
  int c = atoi(av[3]);

  if (a > b)
    a = a - b

  if (b < 1)
    c = a + b

  b = 1;

  return 0;
}
```

Symbolic Execution in a Nutshell

- Interpret program using symbolic values instead of concrete ones
- Fork execution on each branch dependent on symbolic values

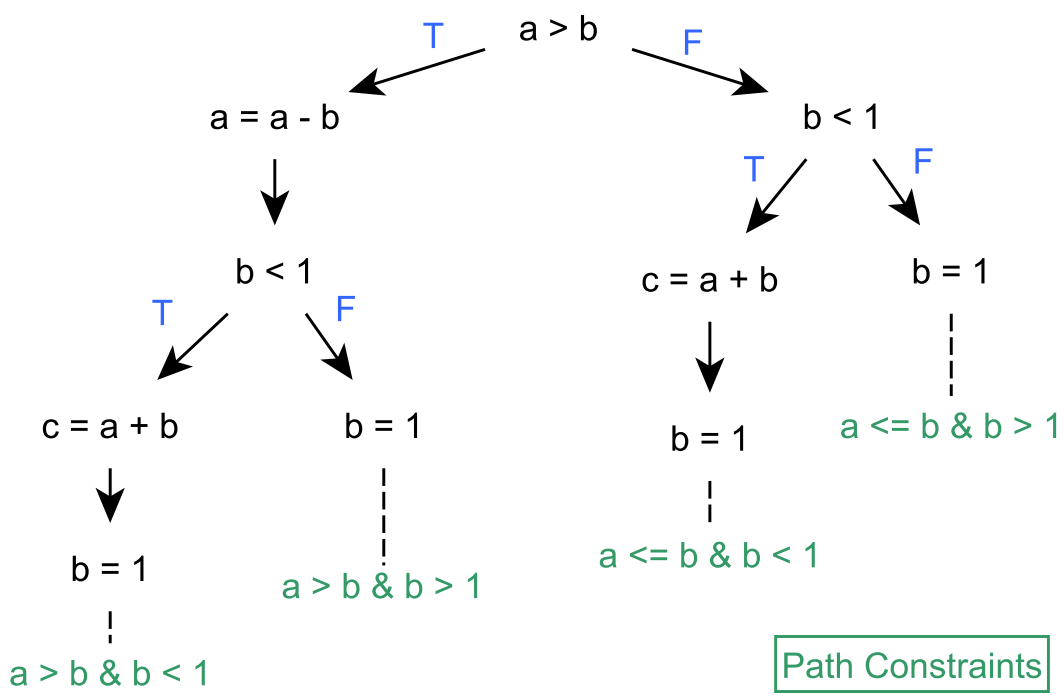
```
int main(int ac, char* av[]) {
  int a = atoi(av[1]);
  int b = atoi(av[2]);
  int c = atoi(av[3]);

  if (a > b)
    a = a - b

  if (b < 1)
    c = a + b

  b = 1;

  return 0;
}
```



Symbolic Execution in a Nutshell

- Interpret program using symbolic values instead of concrete ones
- Fork execution on each branch dependent on symbolic values
- Collect *path constraints* for each execution path

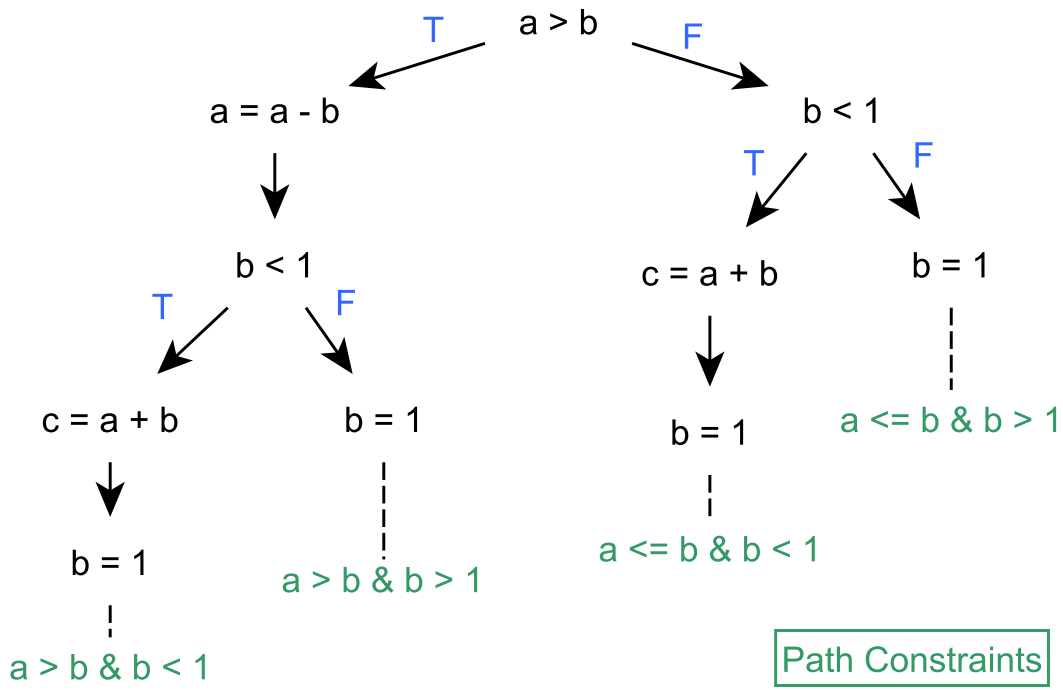
```
int main(int ac, char* av[]) {
  int a = atoi(av[1]);
  int b = atoi(av[2]);
  int c = atoi(av[3]);

  if (a > b)
    a = a - b

  if (b < 1)
    c = a + b

  b = 1;

  return 0;
}
```



Symbolic Execution in a Nutshell

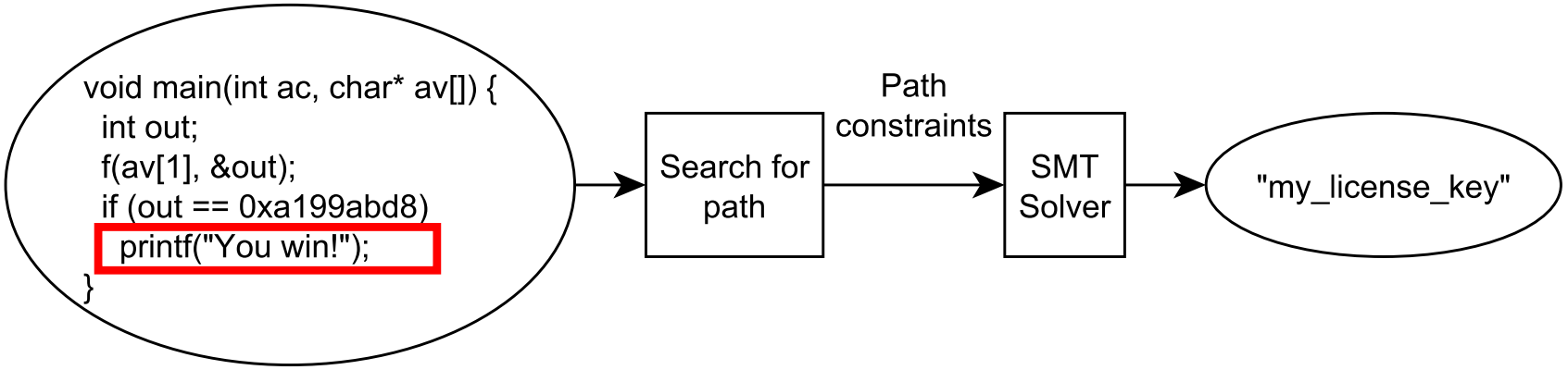
- Interpret program using symbolic values instead of concrete ones
- Fork execution on each branch dependent on symbolic values
- Collect *path constraints* for each execution path
- Get concrete input values from path conditions using SMT solver

```
int main(int ac, char* av[]) {
  int a = atoi(av[1]);
  int b = atoi(av[2]);
  int c = atoi(av[3]);

  if (a > b)
    a = a - b

  if (b < 1)
    c = a + b

  b = 1;

  return 0;
}
```



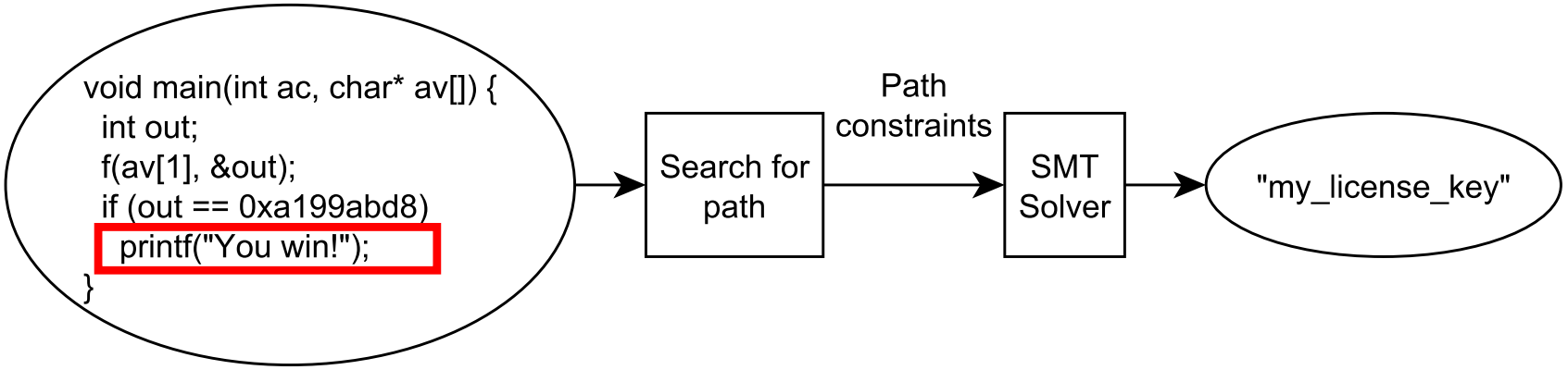
Bypassing License Checks via Symbolic Execution

- Make license input symbolic
- Indicate distinct statement executed when license key is correct
- Symbolic execution finds correct license key (even in obfuscated code)



Bypassing License Checks via Symbolic Execution

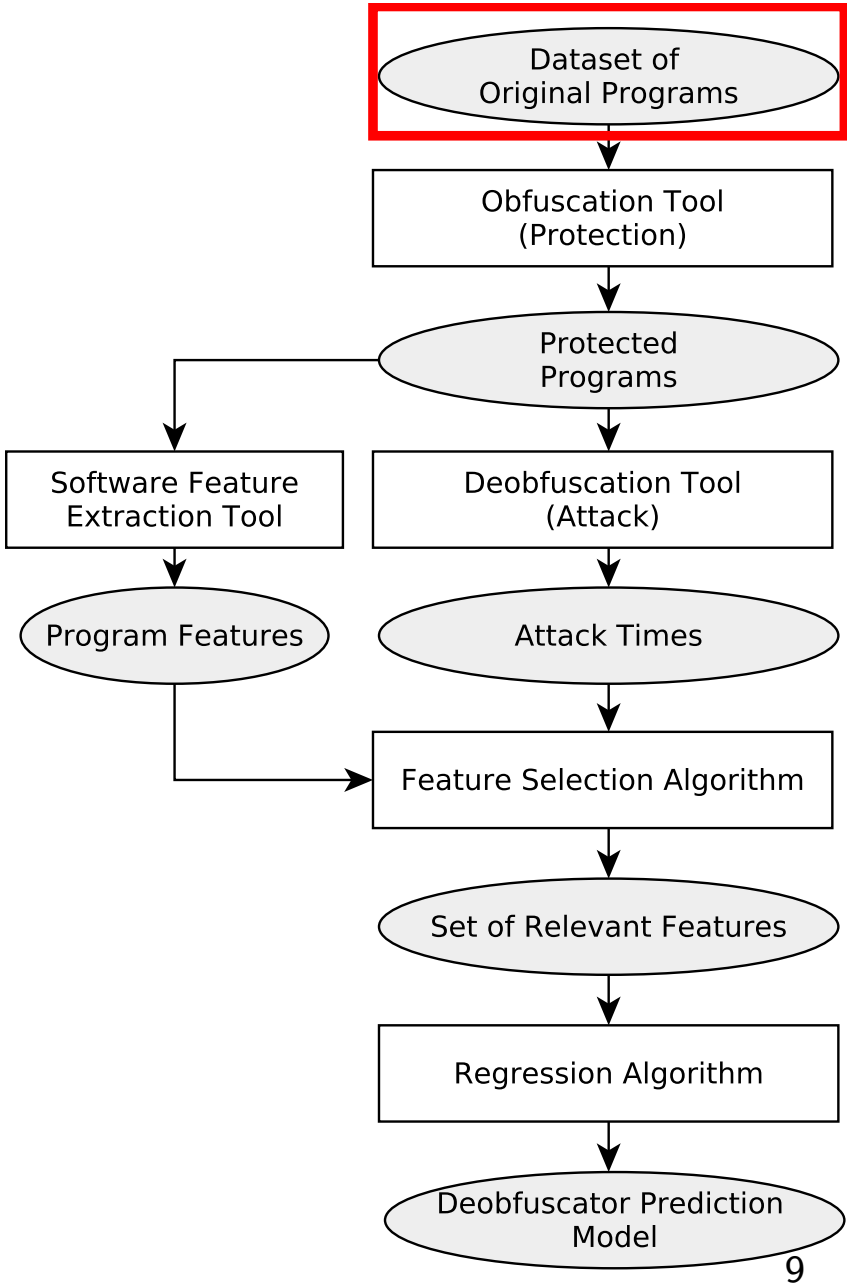
- Make license input symbolic
- Indicate distinct statement executed when license key is correct
- Symbolic execution finds correct license key (even in obfuscated code)



Case study in this paper: Predict time of this attack for given program

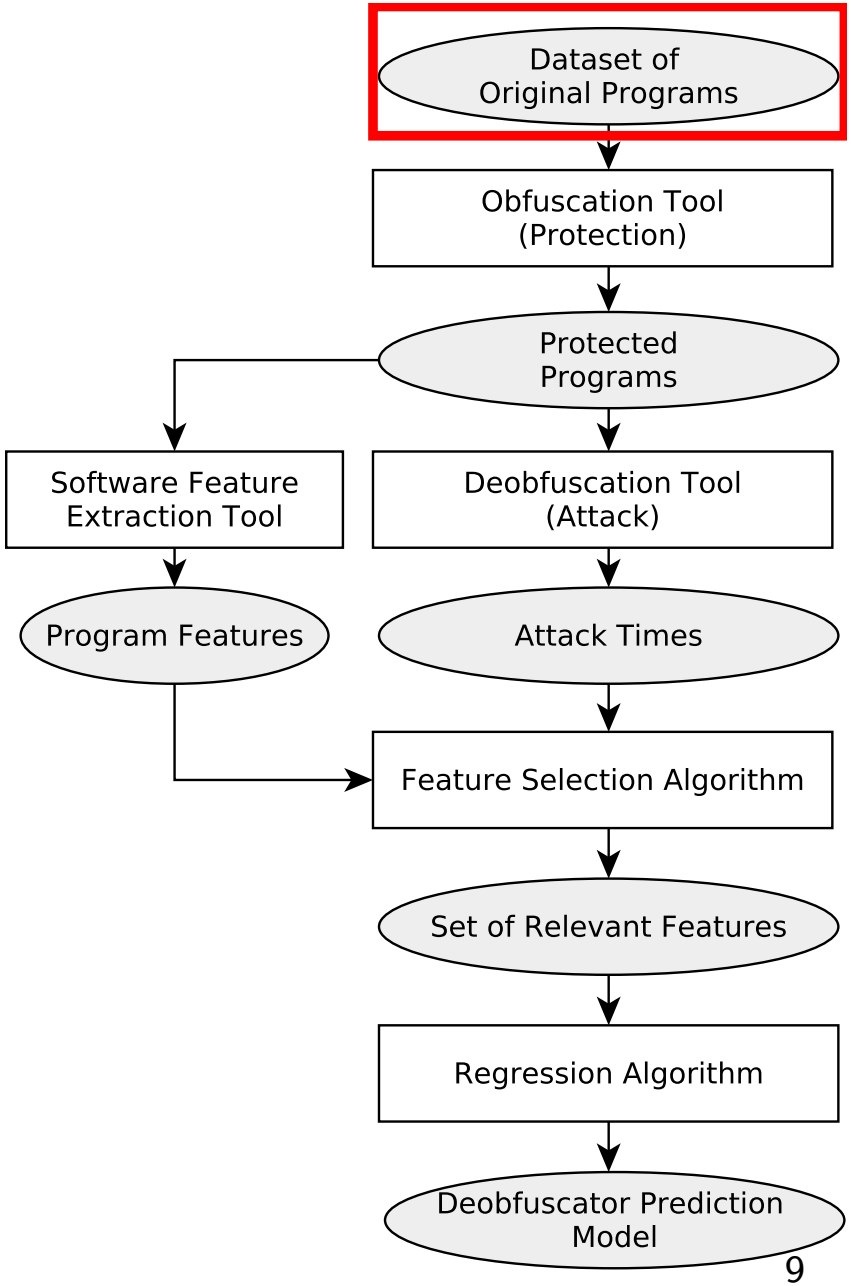
Step 1: Create Dataset of Original Programs

- Need large dataset for training



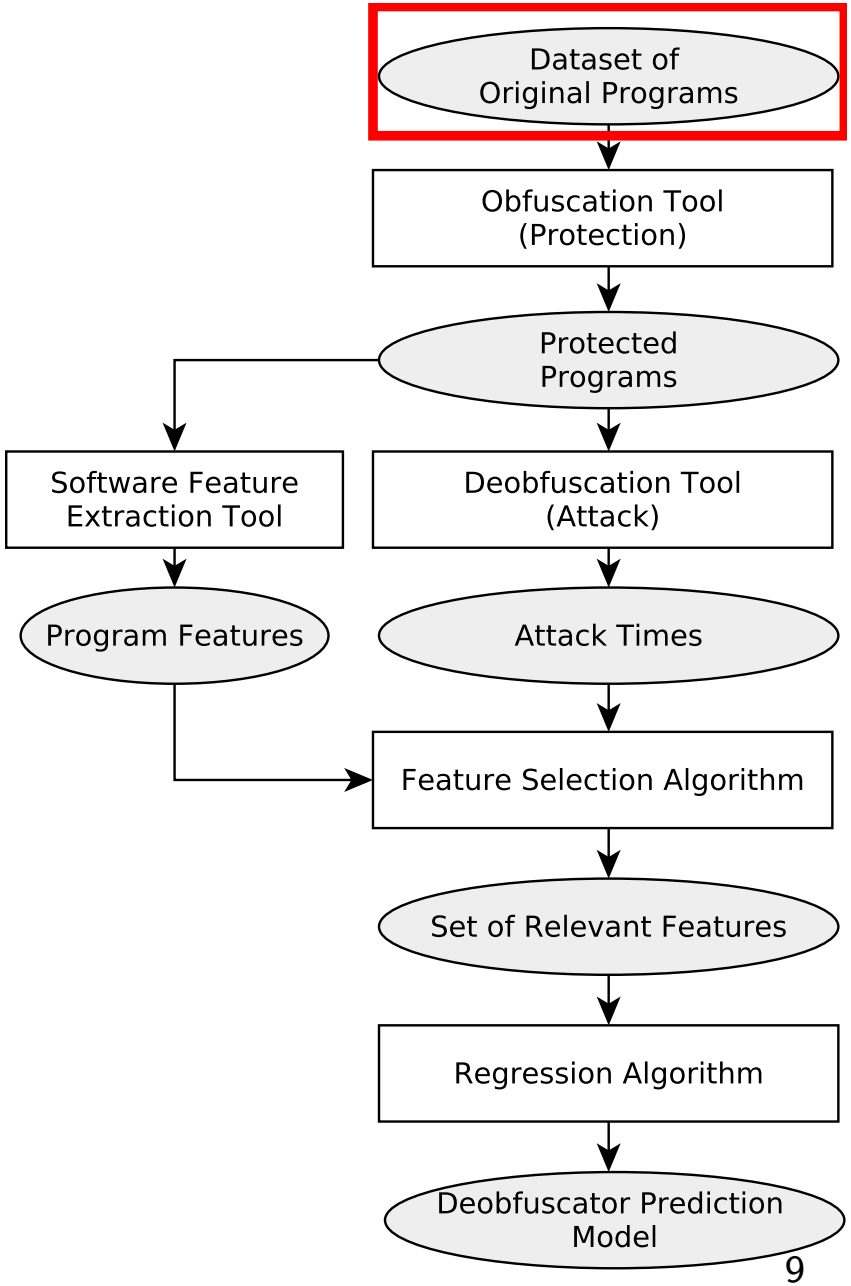
Step 1: Create Dataset of Original Programs

- Need large dataset for training
- Empirically identify features:
  1. Range of the symbolic variable
  2. Depth of nested control flow
  3. Total number of branches
  4. Number of branch statements dependent on symbolic variables
  5. The types of operators



Step 1: Create Dataset of Original Programs

- Need large dataset for training
- Empirically identify features:
  1. Range of the symbolic variable
  2. Depth of nested control flow
  3. Total number of branches
  4. Number of branch statements dependent on symbolic variables
  5. The types of operators
- Generate dataset of programs with varying feature values



Step 1 (cont): C Program Generator

Randomly generated function `f` consists of 3 phases:

- Expansion
- Mixing
- Compression

```
1  void f(int *in, int *out) {
2    long s[2], local1 = 0;
3    // Expansion phase
4    s[0] = in[0] + 762;
5    s[1] = in[0] | (9 << (s[0] % 16 | 1));
6    // Mixing phase
7    while (local1 < 2) {
8      s[1] |= (s[0] & 15) << 3;
9      s[(local1 + 1) % 2] = s[local1];
10     local1 += 1;
11   }
12   if (s[0] > s[1]) {
13     s[0] |= (s[1] & 31) << 3;
14   } else {
15     s[1] |= (s[0] & 15) << 3;
16   }
17   s[0] = s[1];
18   // Compression phase
19   out[0] = (s[0] << (s[1] % 8 | 1));
20 }
21 void main(int ac, char* av[]) {
22   int out;
23   f(av[1], &out);
24   if (out == 0xa199abd8)
25     printf("You win!");
26 }
```

Step 1 (cont): C Program Generator

Randomly generated function `f` consists of 3 phases:

- Expansion
- Mixing
- Compression

Parameters:

- Random seed (3)
- Data types (4)
- Type of loop bounds (3)
- Type of operators (4)
- Control structures (16)
- Size of basic blocks (2)

```c
 void f(int *in, int *out) {
  long s[2], local1 = 0;
  // Expansion phase
  s[0] = in[0] + 762;
  s[1] = in[0] | (9 << (s[0] % 16 | 1));
  // Mixing phase
  while (local1 < 2) {
    s[1] |= (s[0] & 15) << 3;
    s[(local1 + 1) % 2] = s[local1];
    local1 += 1;
  }
  if (s[0] > s[1]) {
    s[0] |= (s[1] & 31) << 3;
  } else {
    s[1] |= (s[0] & 15) << 3;
  }
  s[0] = s[1];
  // Compression phase
  out[0] = (s[0] << (s[1] % 8 | 1));
}
void main(int ac, char* av[]) {
  int out;
  f(av[1], &out);
  if (out == 0xa199abd8)
    printf("You win!");
}
```

Step 1 (cont): C Program Generator

Randomly generated function `f` consists of 3 phases:

- Expansion
- Mixing
- Compression

Parameters:

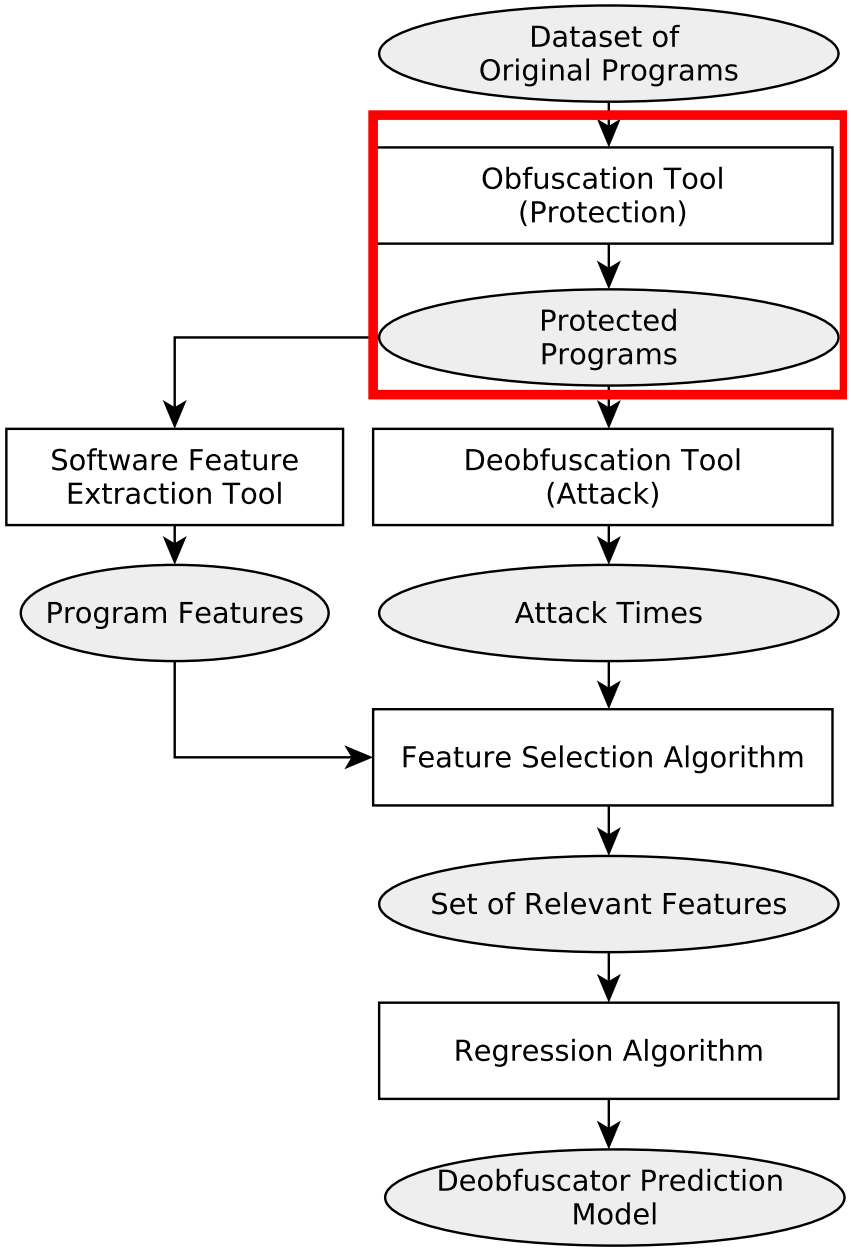
- Random seed (3)
- Data types (4)
- Type of loop bounds (3)
- Type of operators (4)
- Control structures (16)
- Size of basic blocks (2)

Total number of generated programs:
$3 \times 4 \times 3 \times 4 \times 16 \times 2 = 4608$

```c
 void f(int *in, int *out) {
  long s[2], local1 = 0;
  // Expansion phase
  s[0] = in[0] + 762;
  s[1] = in[0] | (9 << (s[0] % 16 | 1));
  // Mixing phase
  while (local1 < 2) {
    s[1] |= (s[0] & 15) << 3;
    s[(local1 + 1) % 2] = s[local1];
    local1 += 1;
  }
  if (s[0] > s[1]) {
    s[0] |= (s[1] & 31) << 3;
  } else {
    s[1] |= (s[0] & 15) << 3;
  }
  s[0] = s[1];
  // Compression phase
  out[0] = (s[0] << (s[1] % 8 | 1));
}
void main(int ac, char* av[]) {
  int out;
  f(av[1], &out);
  if (out == 0xa199abd8)
    printf("You win!");
}
```

Step 2: Obfuscate Programs

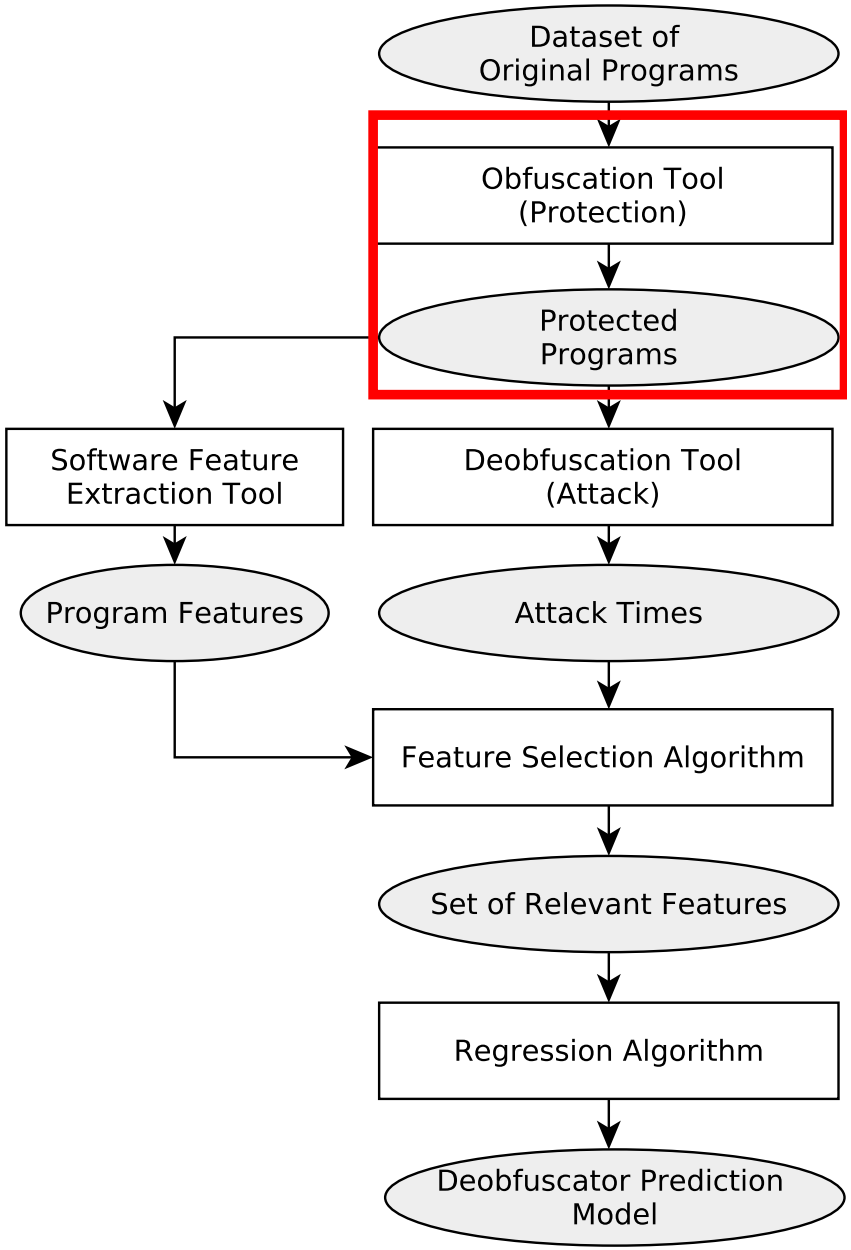
- Each program generated in Step 1 obfuscated with Tigress C Obfuscator



Step 2: Obfuscate Programs

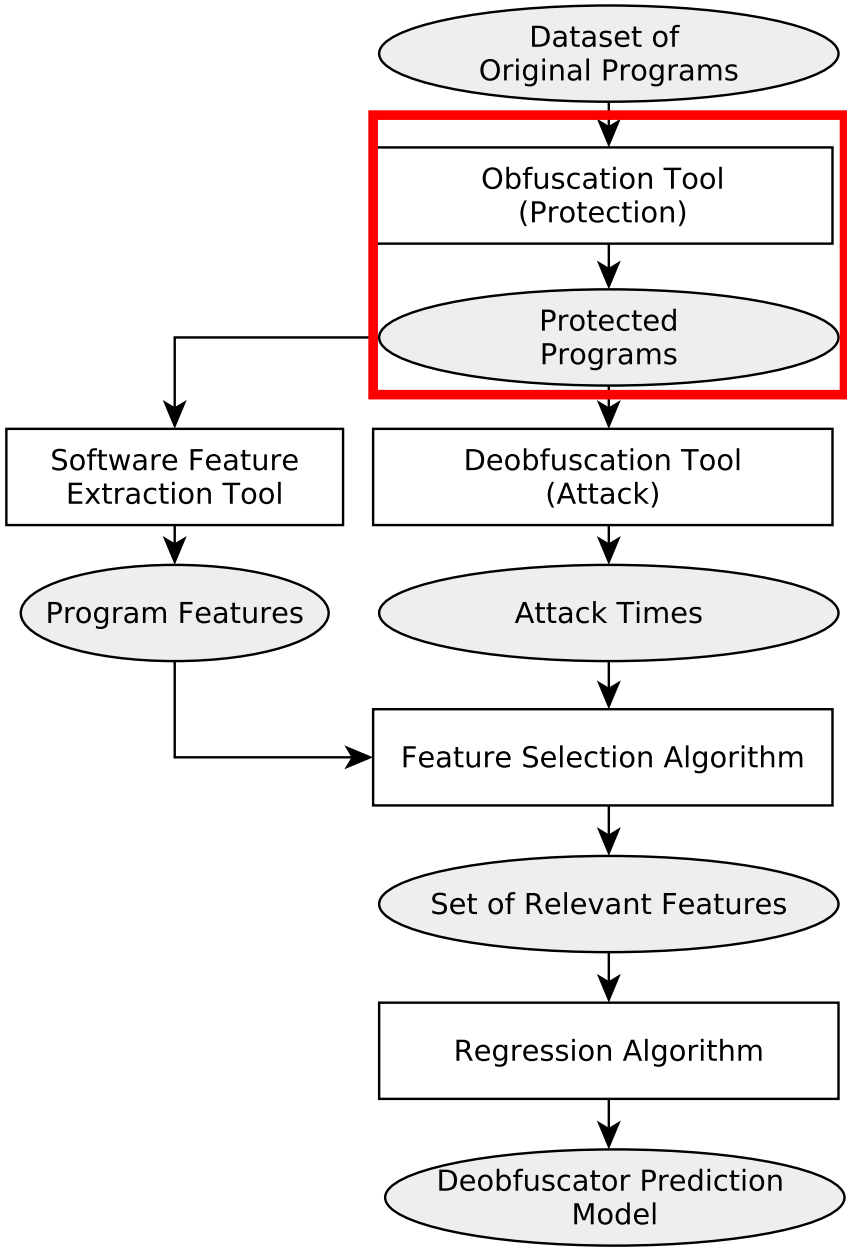
- Each program generated in Step 1 obfuscated with Tigress C Obfuscator

- Transformations used:
  1. Virtualization
  2. Flattening
  3. Opaque Predicates
  4. Encode arithmetic
  5. Encode literals



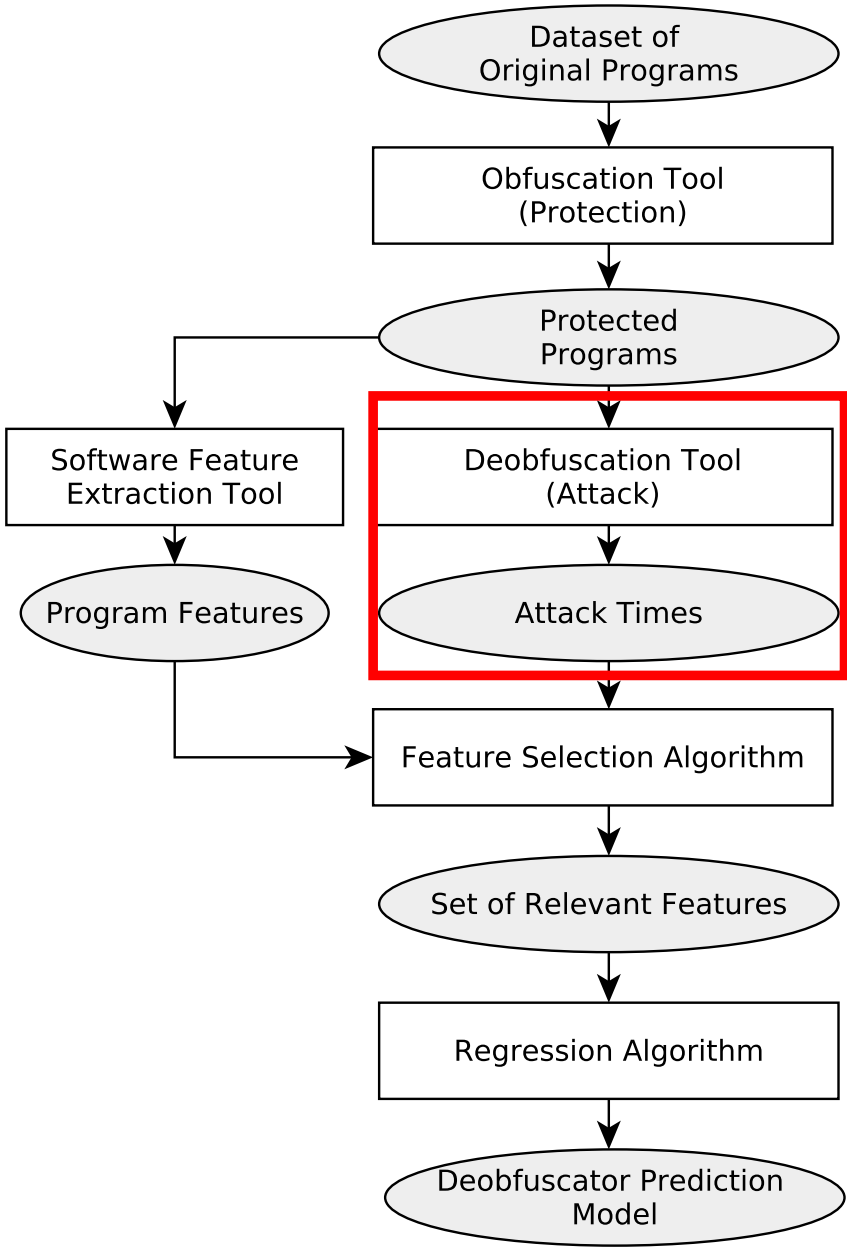
Step 2: Obfuscate Programs

- Each program generated in Step 1 obfuscated with Tigress C Obfuscator
- Transformations used:
  1. Virtualization
  2. Flattening
  3. Opaque Predicates
  4. Encode arithmetic
  5. Encode literals
- Total number of obfuscated programs: $5 \times 4608 = 23040$



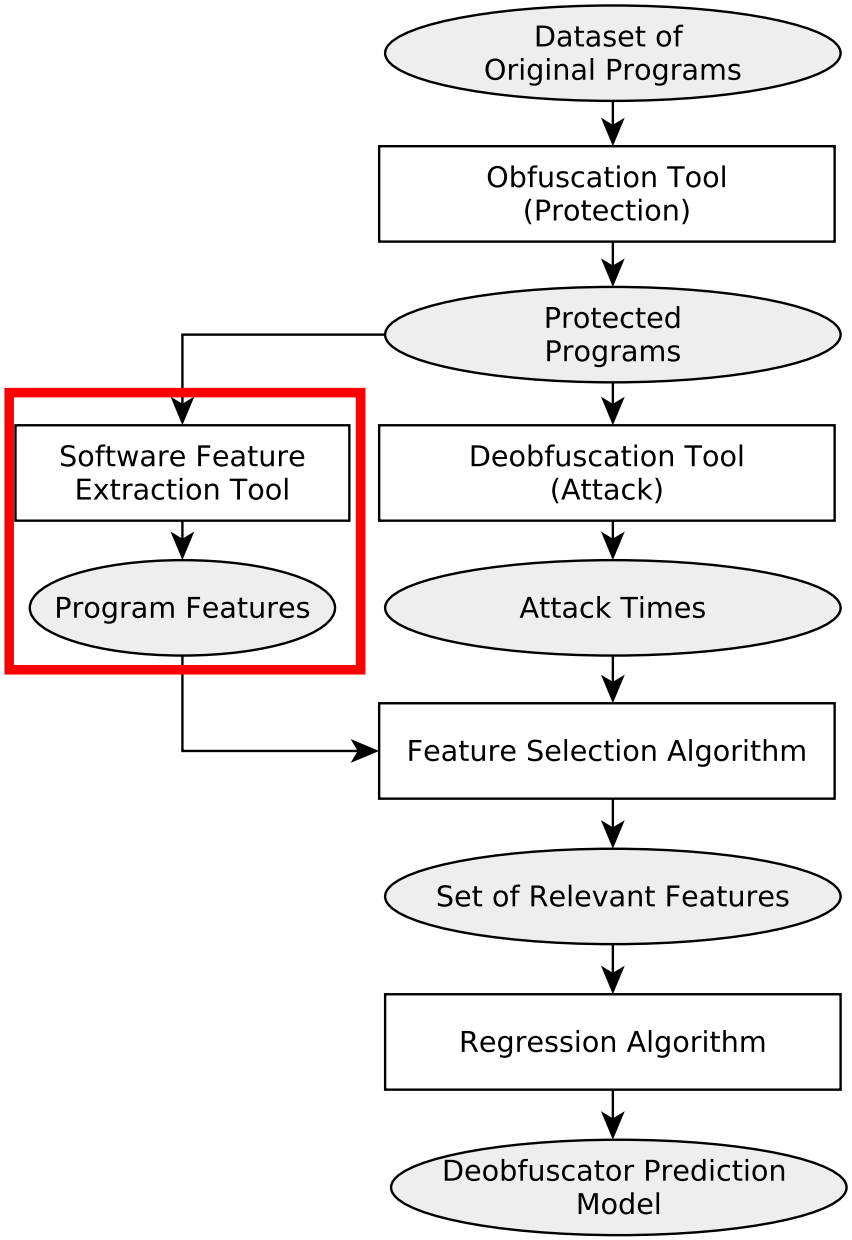
Step 3: Attack Obfuscated Programs

- **Attacker goal:** bypass license check
- Execute attack based on symbolic execution on programs from Step 2
- Record time of successful attacks



Step 4: Feature Extraction

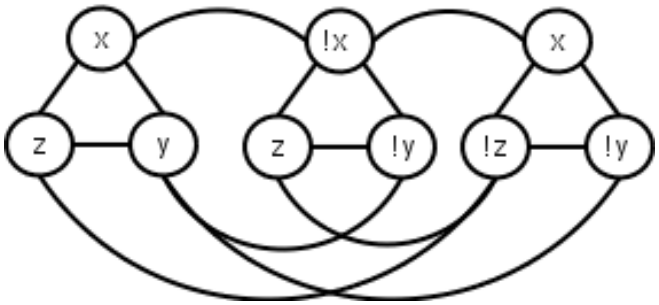
- Extracted 64 features in total:
  - Static code metrics
  - Dynamic code metrics
  - SAT metrics

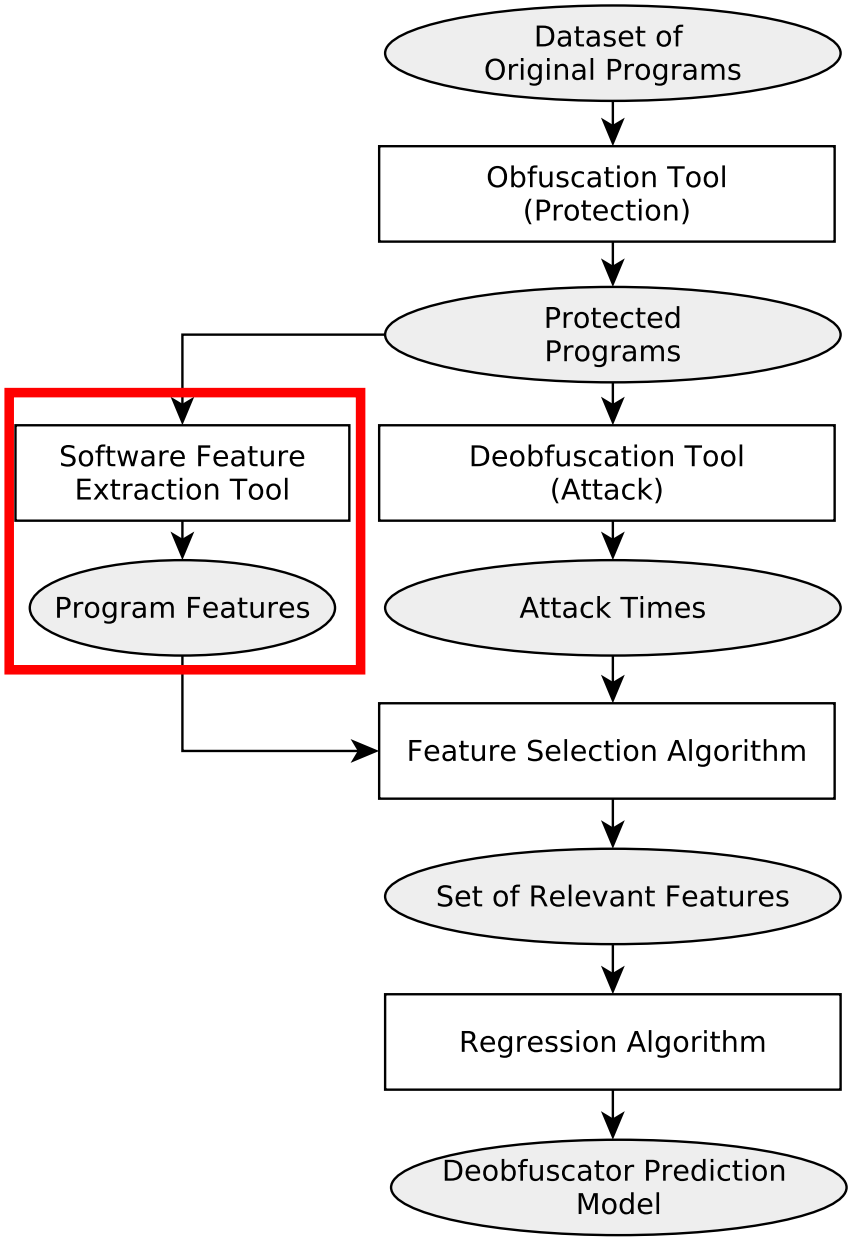


Step 4: Feature Extraction

- Extracted 64 features in total:
  - Static code metrics
  - Dynamic code metrics
  - SAT metrics
- Graph metrics on a SAT formula represented as a graph

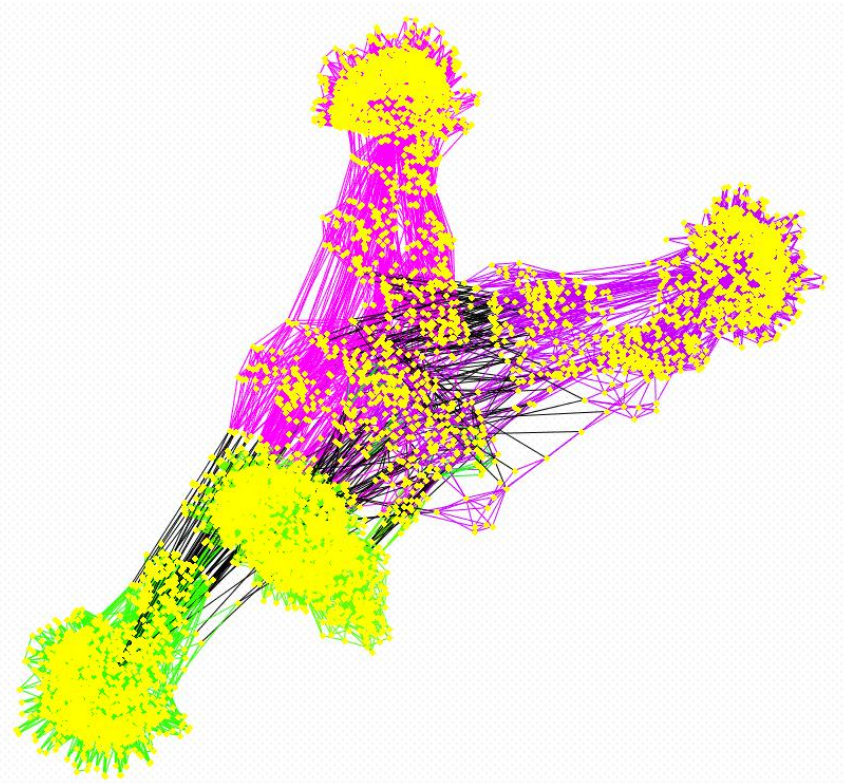
$$(x+y+z)\cdot(!x+!y+z)\cdot(x+!y+!z)$$





Step 4 (cont): SAT Before & After Obfuscation

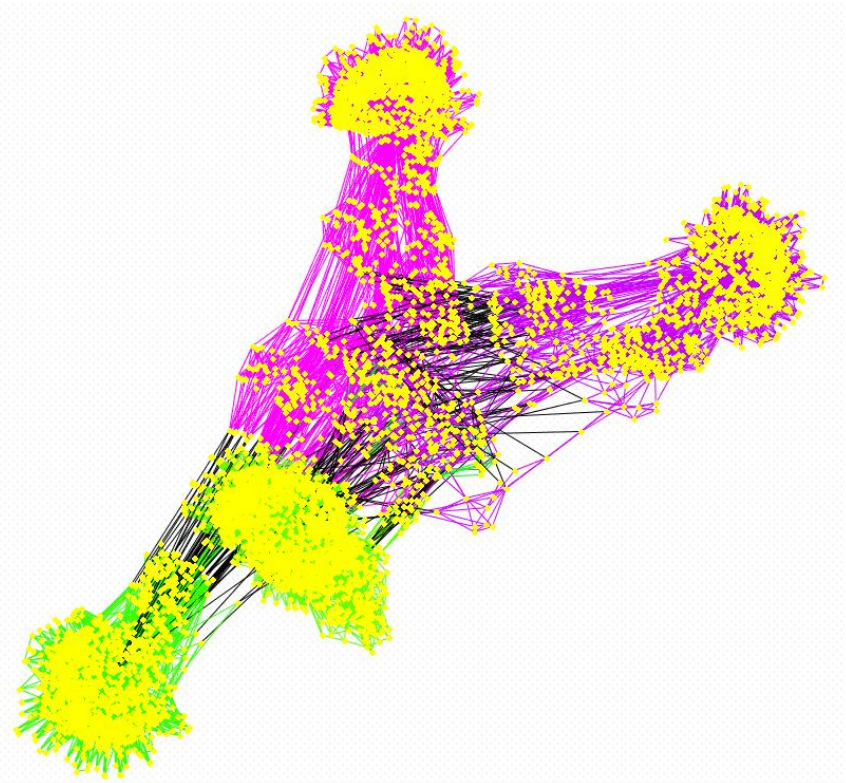
Before Obfuscation (7.5 sec)



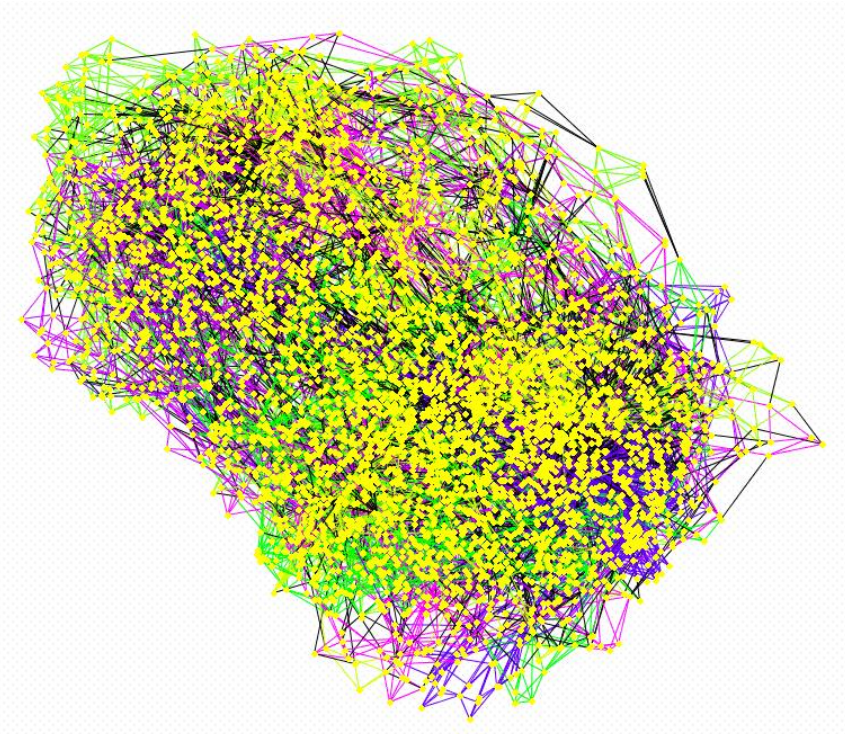
```c
unsigned int SDBMHash(char* str, unsigned int len)
{
   unsigned int hash = 0;
   unsigned int i    = 0;
   for(i = 0; i < len; str++, i++)
      hash = (*str) + (hash << 6)
             + (hash << 16) - hash;
   return hash;
}

int main(int argc, char* argv[]) {
  unsigned char *str = argv[1];
  unsigned int hash = SDBMHash(str, strlen(str));

  if (hash == 0x89dcd66e)
    printf("You win!\n");
  return 0;
}
```

Step 4 (cont): SAT Before & After Obfuscation

Before Obfuscation (7.5 sec)



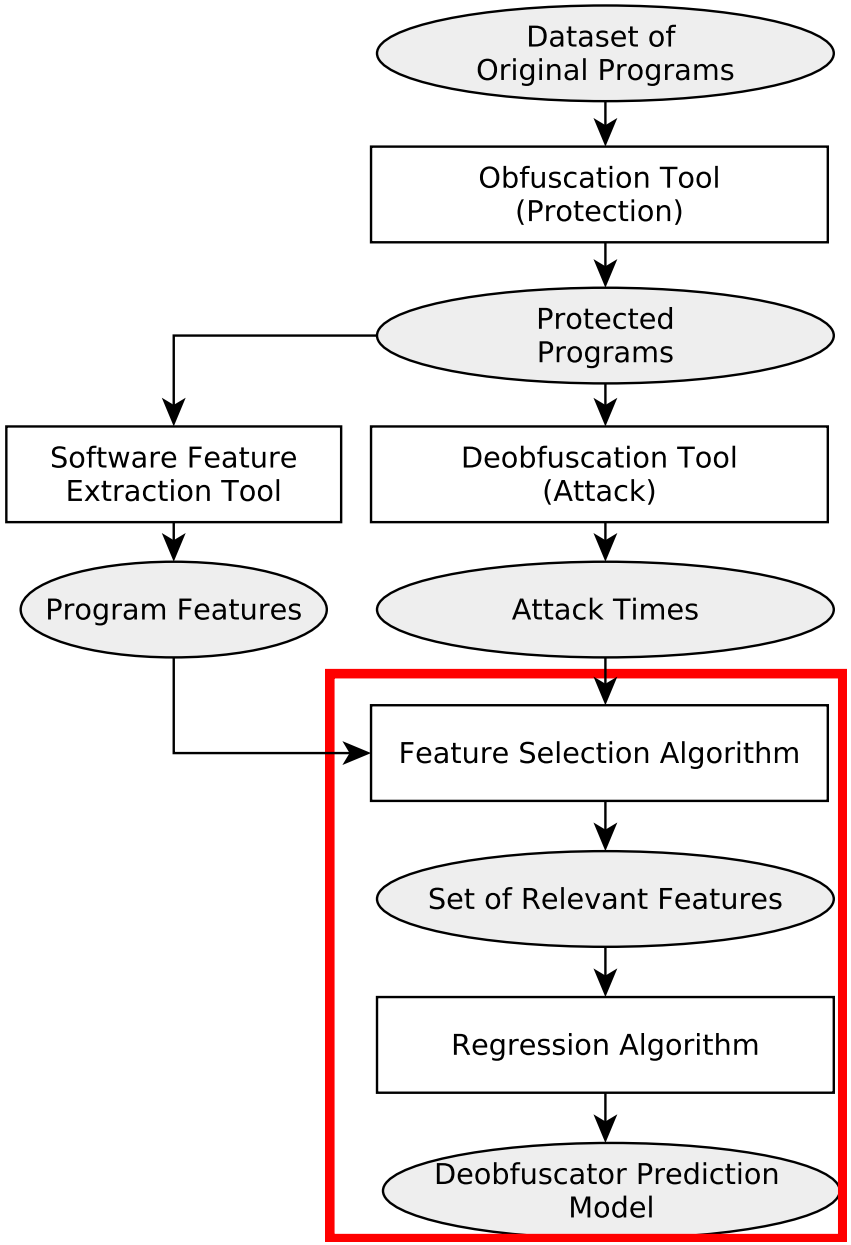
After Obfuscation (438 sec)



Strong obfuscation transformations destroy community structures

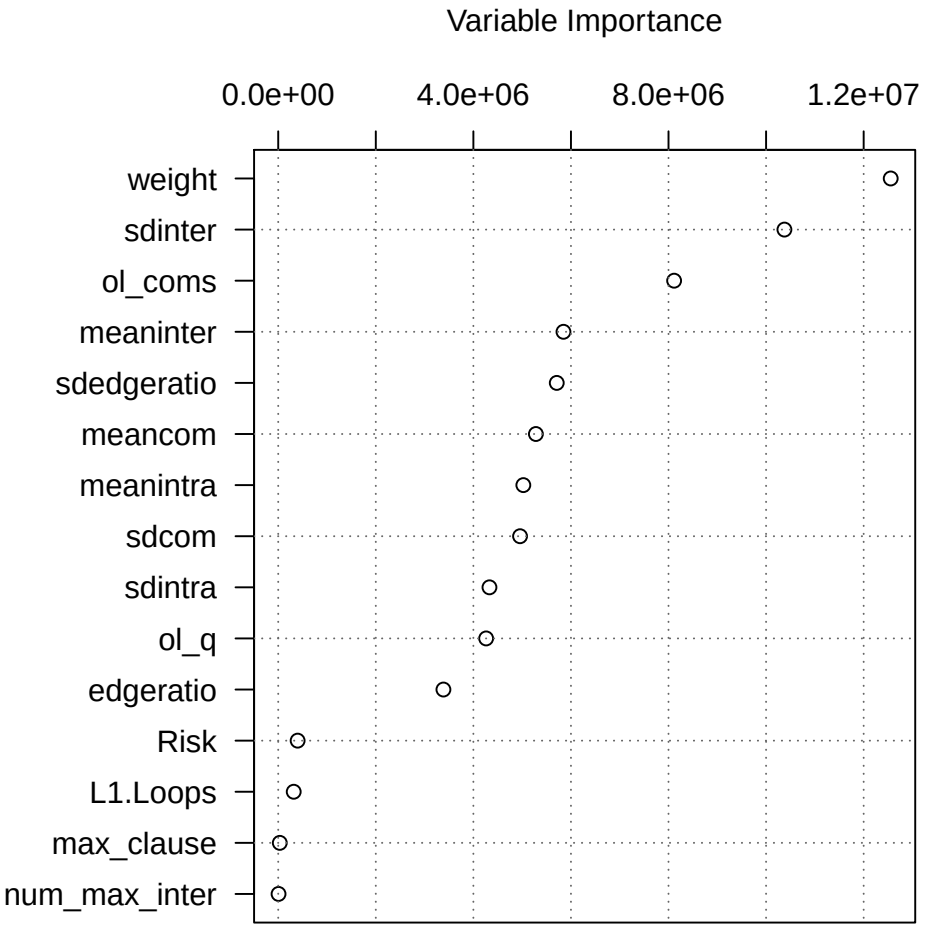
Step 5: Create Prediction Model for Deobfuscation

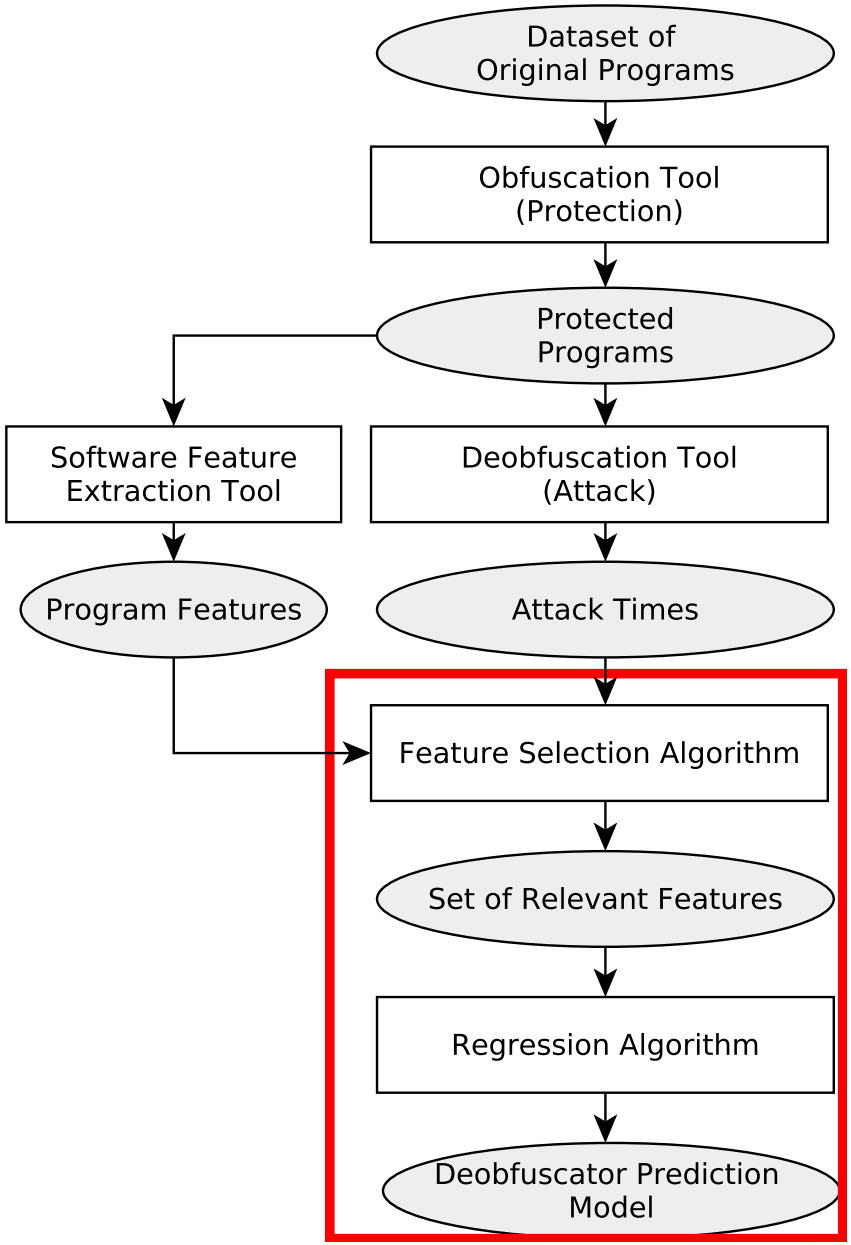
- Performed recursive feature selection → 15 features



Step 5: Create Prediction Model for Deobfuscation

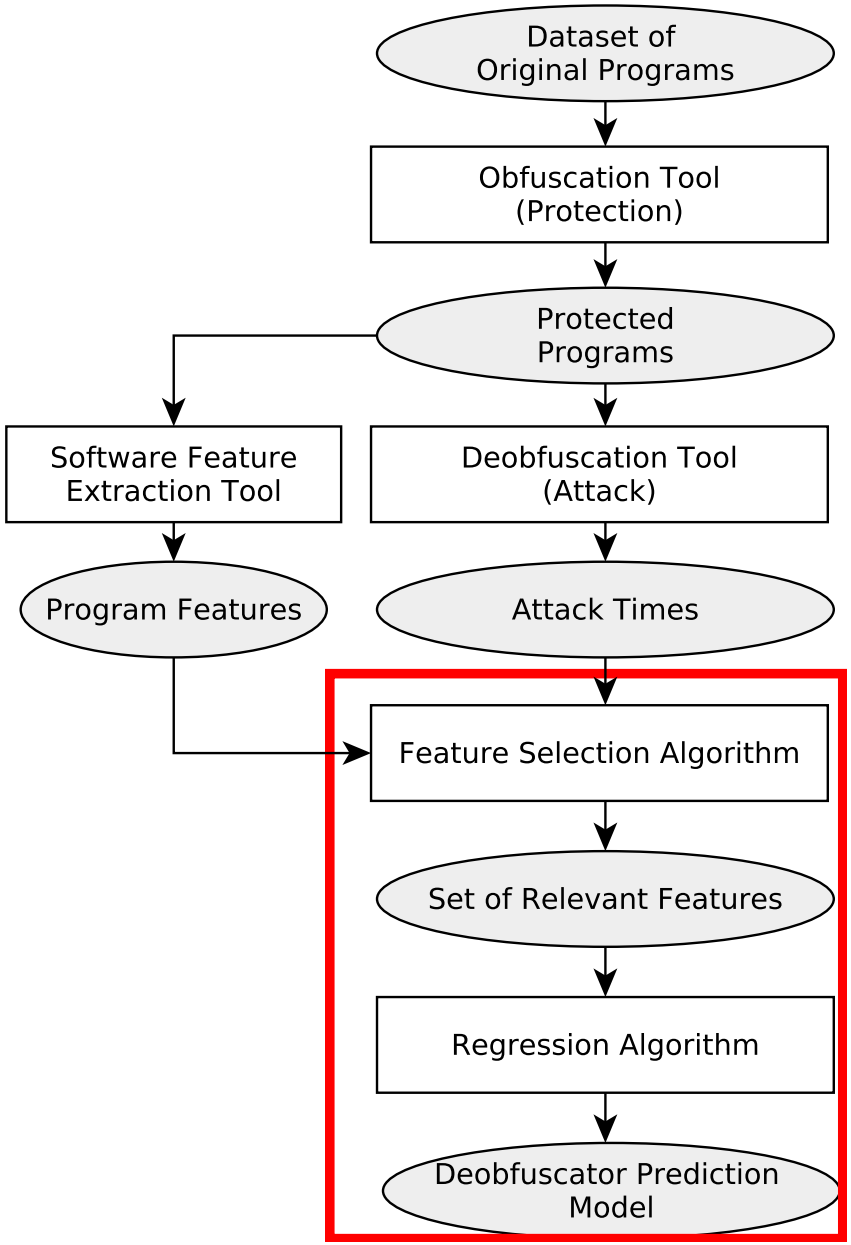
- Performed recursive feature selection → 15 features





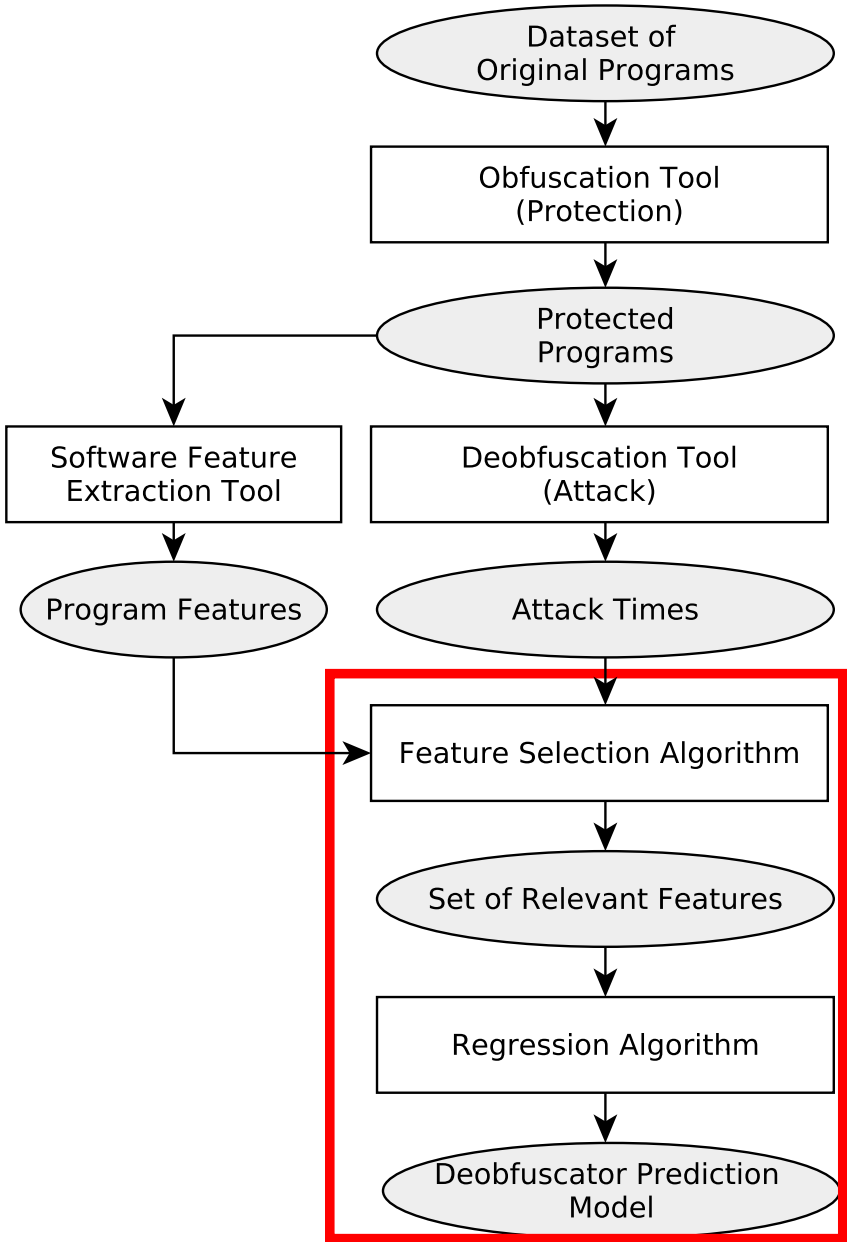
Step 5: Create Prediction Model for Deobfuscation

- Performed recursive feature selection → 15 features
- Most important are SAT features



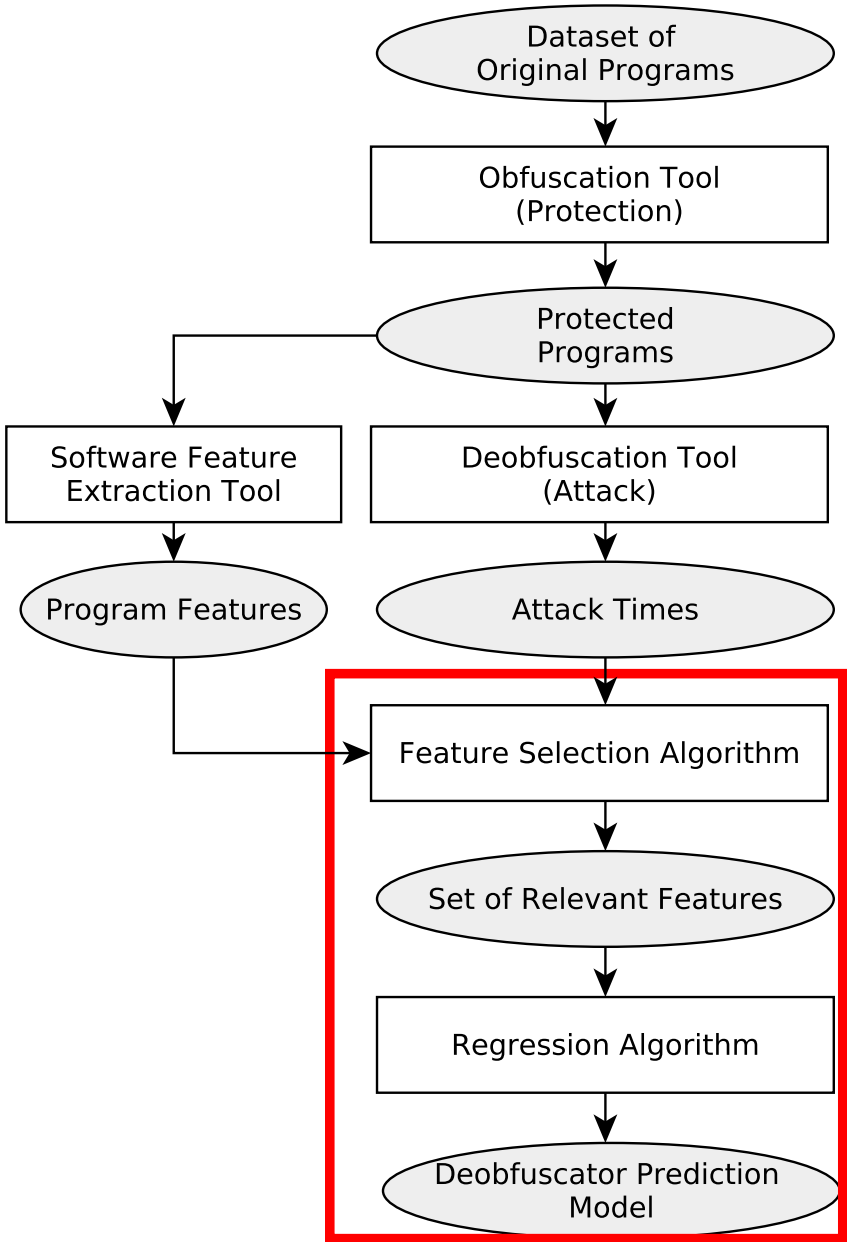
Step 5: Create Prediction Model for Deobfuscation

- Performed recursive feature selection → 15 features
- Most important are SAT features
- These features stem from complexity of path constraints



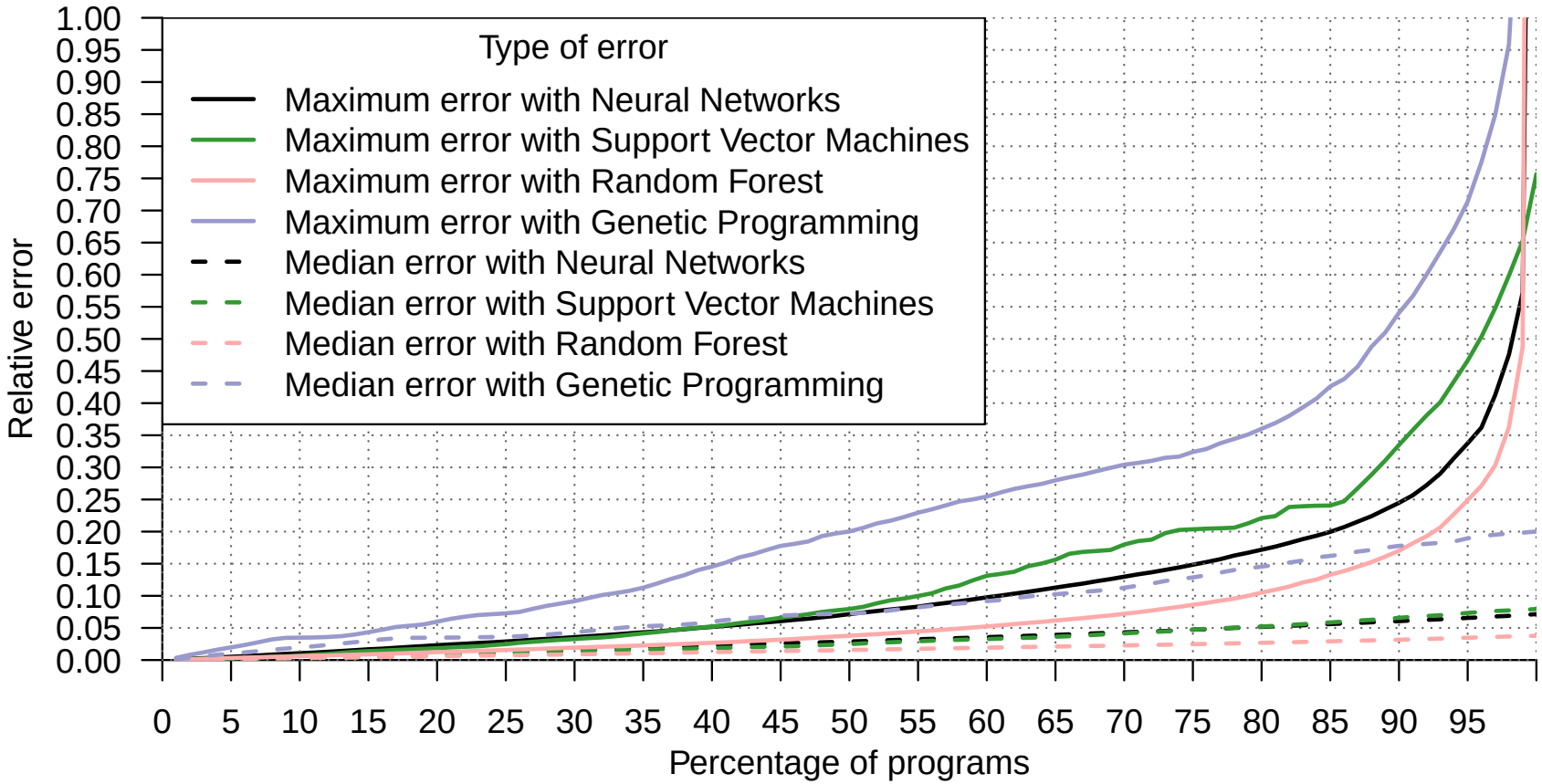
Step 5: Create Prediction Model for Deobfuscation

- Performed recursive feature selection → 15 features
- Most important are SAT features
- These features stem from complexity of path constraints
- Employed different ML algorithms for predicting attacker effort:
  - Neural Networks
  - Support Vector Machines
  - Random Forest
  - Genetic Programming



Step 5: Predict Average Effort Needed by Attack

Comparison of prediction error from different ML algorithms



Do our results generalize?

Parameters:

- Random seed (3)
- Data types (4)
- Type of loop bounds (3)
- Type of operators (4)
- Control structures (16)
- Size of basic blocks (2)

Each program obfuscated with Tigress C Obfuscator

1. Virtualization
2. Flattening
3. Opaque Predicates
4. Encode arithmetic
5. Encode literals

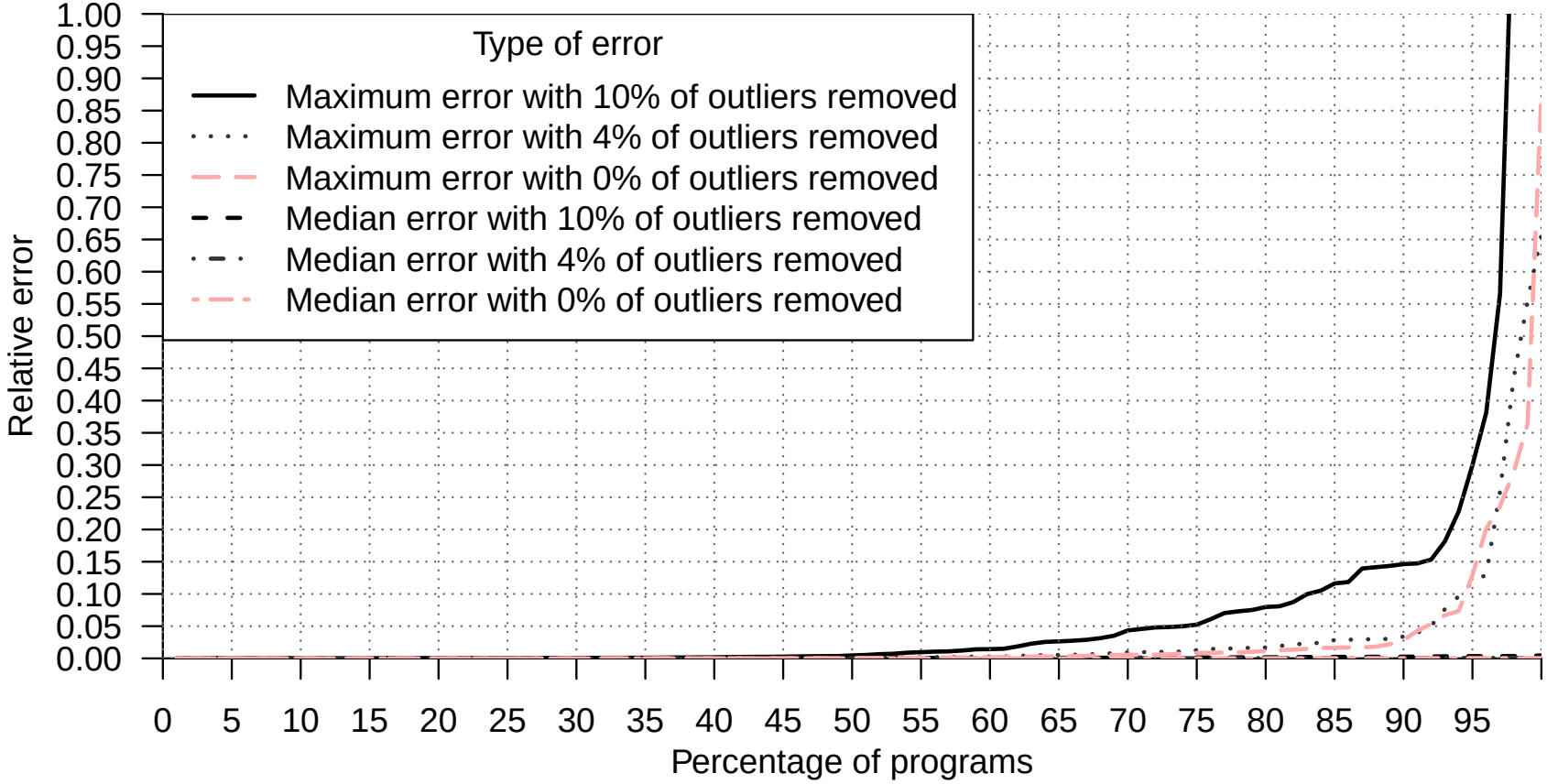
```
1  void f(int *in, int *out) {
2    long s[2], local1 = 0;
3    // Expansion phase
4    s[0] = in[0] + 762;
5    s[1] = in[0] | (9 << (s[0] % 16 | 1));
6    // Mixing phase
7    while (local1 < 2) {
8      s[1] |= (s[0] & 15) << 3;
9      s[(local1 + 1) % 2] = s[local1];
10     local1 += 1;
11   }
12   if (s[0] > s[1]) {
13     s[0] |= (s[1] & 31) << 3;
14   } else {
15     s[1] |= (s[0] & 15) << 3;
16   }
17   s[0] = s[1];
18   // Compression phase
19   out[0] = (s[0] << (s[1] % 8 | 1));
20 }
21 void main(int ac, char* av[]) {
22   int out;
23   f(av[1], &out);
24   if (out == 0xa199abd8)
25     printf("You win!");
26 }
```

Generality of Results

- Collected 11 non-cryptographic hash functions
- Used them as part of license checking algorithms in C programs
- Obfuscated using Tigress (2-layers of obfuscation): 275 programs
- Applied prediction model trained on randomly generated C programs

Generality of Results

- Collected 11 non-cryptographic hash functions
- Used them as part of license checking algorithms in C programs
- Obfuscated using Tigress (2-layers of obfuscation): 275 programs
- Applied prediction model trained on randomly generated C programs



Generality of Results (cont)

- Collected 11 SAT-Competition instances corresponding to cryptographic hash functions
- Trained RF model with top 10 SAT features using SAT instances of randomly generated C programs
- Applied RF model to SAT instances of cryptographic hash functions

Generality of Results (cont)

- Collected 11 SAT-Competition instances corresponding to cryptographic hash functions
- Trained RF model with top 10 SAT features using SAT instances of randomly generated C programs
- Applied RF model to SAT instances of cryptographic hash functions

Instance Name	Solver (sec)	Predicted (sec)	$\frac{Predicted}{Solver}$
MD5-27-4	25.37	71.56	2.82
mizh-md5-47-3	681.29	950.43	1.39
mizh-md5-47-4	235.53	1069.19	4.53
mizh-md5-47-5	1832.96	437.98	0.23
mizh-md5-48-2	445.19	523.70	1.17
mizh-md5-48-5	227.05	644.38	2.83
mizh-sha0-35-2	330.48	158.57	0.47
mizh-sha0-35-3	139.93	213.03	1.52
mizh-sha0-35-4	97.62	214.61	2.19
mizh-sha0-35-5	164.71	193.49	1.17
mizh-sha0-36-2	85.44	222.07	2.59

Conclusions and Future Work

Conclusions

- General approach towards deobfuscation effort prediction
- C code generator: $>$ 4500 non-obfuscated programs
- Obfuscated using 5 transformations: $>$ 23000 obfuscated programs
- SAT features most relevant for prediction
- Prediction models using GP, SVM, NN, RF
- Train using randomly generated functions, predict effort of (non-)cryptographic hash functions

Conclusions and Future Work

Conclusions

- General approach towards deobfuscation effort prediction
- C code generator: $>$ 4500 non-obfuscated programs
- Obfuscated using 5 transformations: $>$ 23000 obfuscated programs
- SAT features most relevant for prediction
- Prediction models using GP, SVM, NN, RF
- Train using randomly generated functions, predict effort of (non-)cryptographic hash functions

Future perspectives:

- Apply approach to other auto-MATEd attacks, e.g. CFG simplification, disassembly, etc.
- Extend benchmark program generator
- Use machine learning to derive features

Thank you for your attention

- C Code Generator:
  `http://tigress.cs.arizona.edu/transformPage/docs/randomFuns`

- Datasets and scripts:
  `https://github.com/tum-i22/obfuscation-benchmarks`

Questions ?