Sebastian Banescu is an IT Security Specialist at BMW AG in Munich, where he is involved in various projects regarding the security of the connected car against remote attackers, tunning garages and malicious car owners.
In July 2017, he received his PhD, with distinction, at the Technical University of Munich under the supervision of Prof. Alexander Pretschner. The topic of his PhD thesis was to characterize the strength of software obfuscation against automated man-at-the-end attackers.
The end goal of his work was to develop a framework that allows software defenders to easily choose which software protections to employ in order to protect their software against malicious end-users.
Before moving to Germany in 2013, Dr. Banescu, received a MSc. in Computer Science and Engineering, "cum laude", from Eindhoven University of Technology in the Netherlands, and a BSc. in Computer Science and Engineering, from the Technical University of Cluj-Napoca in Romania.

Forschungsbeitrag zu Software Schutz

Sebastian Banescu

Evaluating Software Protection against Automated Reverse Engineering Attacks

1. Introduction

Software protection focuses on defending software applications against attackers who have access to the (binary) code of the application. Two common examples of such attackers are malicious end-users of (1) websites, who can inspect or tamper with the JavaScript code sent by the website or (2) games, who can inspect or tamper with the machine code of the game executable. These malicious end-users are called man-at-the-end (MATE) attackers, because they are recipients of the software applications and they control the execution environment of the application.

Software developers know that some of the end-users of their software may be MATE attackers, who try to steal their intellectual property or try to change the intended behavior of the application, e.g. such that they no longer need to pay a license or subscription fee in order to use the software. Therefore, many developers employ software protection techniques in order to prevent or detect MATE attackers who want to inspect or tamper with the software. Software protection, sometimes also called *code-hardening* includes dozens of code transformation techniques which can be grouped in the following categories:

- ***Obfuscation*** transforms the syntax of the code such that the result is harder to analyze. The functionality of the code is preserved even after the obfuscation transformation.
- ***Tamper-proofing*** and *tamper-detection* protect the integrity of the code. This implies code transformations which add integrity checks that continuously verify the state or functionality of the code.
- ***Tool-targeted*** or *environment-targeted* transformations take advantage of weaknesses in tools or execution environments to prevent analysis, e.g. anti-debugging, anti-sandboxing.
- ***Watermarking*** adds a distinctive pattern inside the code, which is hard to detect or remove by attackers, but can be easily recovered by the software developer. This pattern is useful to track unauthorized copies of code, that are used by competitors.

Software protection focuses on defending software applications against malicious end-users, also called man-at-the-end (MATE) attackers, who have access to the (binary) code of the application and its execution environment. The biggest problem of software developers who want to protect their software, is that there are dozens of software protection transformations and it is not clear how much effort the attacker will need if certain transformations are combined.
This paper proposes a framework for quantifying the effort needed by MATE attackers against a given protected software. Our framework helps software developers identify software features which are crucial for MATE attacks and which can be transformed by software protection in such a way to make the attack more difficult. These features are then used to train regression models, which predict how much time the MATE attack will take on a given protected software.

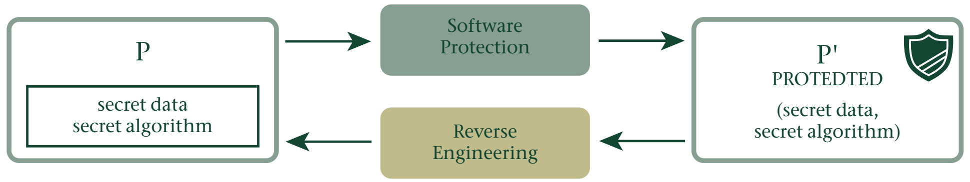


Figure 1:
Input and output of software protection and reverse engineering.

The top of *Figure 1* shows that software protection takes a program *P* as input and outputs a protected program *P'* having the same semantics and protected data and algorithms. If the software is not protected well enough, MATE attackers will be able to reverse engineer program *P' (bottom of Figure 1)* and achieve their end-goal of extracting secrets or tampering with the software. Given enough time and resources a MATE attacker will eventually be able to reverse engineer any protected software, however, software developers only want to raise the bar for the MATE attacker such that the task of reverse engineering is not economically attractive anymore.

For example if the resources needed to bypass a license check for a game costs 100 times more than the license fee, then an attacker is more likely to buy the license than reverse engineer the game executable. The biggest problem of software developers is that there are dozens of software protection transformations and it is not clear how much effort the attacker will need to invest in reversing the software if certain transformations are combined.

This paper proposes a framework for quantifying the effort needed by MATE attackers against a given protected software. Our framework helps software developers to identify software features which are crucial for MATE attacks and which must be transformed by software protection in such a way to make the attack more difficult. The paper also describes a use-case where the attacker employs a state-of-the-art program analysis technique called *symbolic execution* to extract license keys from obfuscated programs. These features are then used to train machine learning models, which are able to predict how much time the attack based on symbolic execution will take on a given protected software.

2. Software Protection Transformations

There are dozens of software protection transformations published in literature[5].

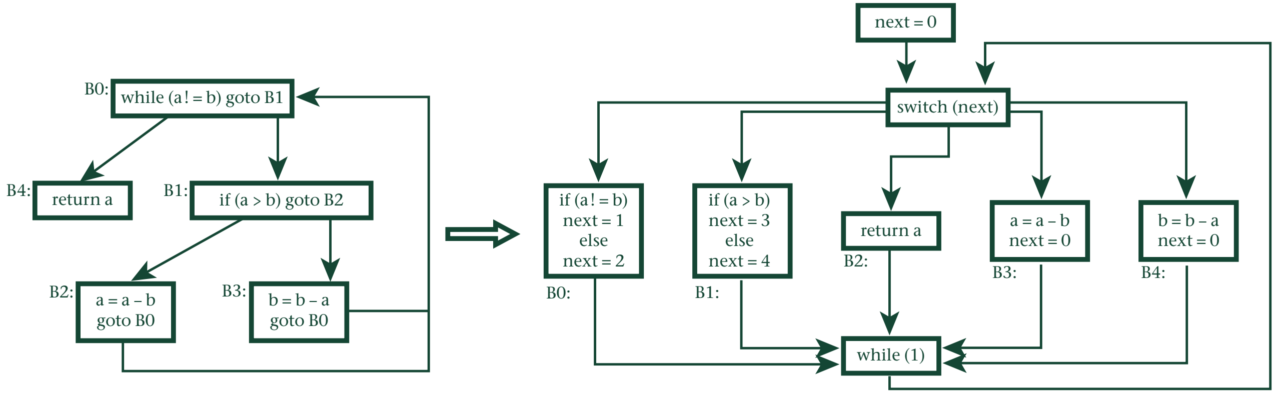


Figure 2:
Control-Flow-Flattening example for a function computing the greatest common divisor of two integers.

Moreover, these transformations can be applied successively to a given software application in various orders, practically entailing an unlimited number of different versions of that application, which are syntactically different, but functionally equivalent. In this section, we only describe a subset of some of the most popular obfuscation transformations, which are used in the experiments presented Sections 4:

- ***Encode literals*** (EncL) takes constant numbers or strings and encodes them into other data values, which are converted back to the original value during runtime, by one or more decoding functions. For example a constant string value is split into separately stored ASCII values and each value is XOR-ed with a constant value. At runtime the reverse process is applied to each of the encoded characters, in order to obtain the original string.
- ***Opaque predicates*** (AddO) are complex boolean expressions, which are always true or always false regardless of the value of the variables. However, for a MATE attacker it is difficult to determine that an expression is an opaque predicate. For instance, $x^2 + x = 0 \bmod 2$ is always true, regardless of the value of x, because if x is even then $x^2 + x$ is even and if x is odd, then $x^2 + x$ is still even. Opaque predicates are generally used in conditional statements, where the branch that is never taken is filled with dead code, which cannot be removed by the compiler. Moreover, opaque predicates can also be updated during runtime (UpdO) to frustrate the attacker.
- ***Encode arithmetic*** (EncA) also called *mixed Boolean-arithmetic* (MBA) takes simple arithmetic operations (e.g. addition, subtraction, multiplication, division) or simple Boolean operations (e.g. AND, OR, NOR, XOR) involving multiple variables and transforms them into functionally equivalent complex expressions consisting of multiple arithmetic and Boolean operations. For example, $x + y$ may be transformed into $2(x \vee y) - (x + y)$. Moreover, such transformations may be applied multiple times successively,

which leads to a highly complex expression, which is difficult for an attacker to simplify.
- ***Control-flow flattening*** (Flat) takes each basic block of a program and puts them into different *case*-clauses of a large *switch*-statement. *Figure 2* illustrates flattening applied on a function computing the greatest common divisor of two integers *a* and *b* (on the left). After transformation, the *switch*-statement *(right part of Figure 2)* is dependent on a control variable (called *next*), which is set accordingly inside each of the *case*-clauses, such that the original control flow of the program is preserved.
- ***Virtualization obfuscation*** (Virt) takes a sequence of one or more instructions and converts it into a new instruction having a random opcode and operands. Doing this mapping until all the code of a program is covered, gives rise to a new instruction set architecture (ISA). Afterwards, the code of the input program is translated to the new ISA and stored as bytecode. An emulator which maps the bytecode instructions back to native instructions is also generated. The obfuscated program consists of the bytecode and the emulator.

The papers which introduce these obfuscation transformations, provide ad-hoc evaluations of their strength. However, the evaluations often refer to different MATE attackers, i.e. having different goals and employing different program analysis techniques. In this paper we introduce a framework to compare differently protected programs, including programs protected using multiple layers of obfuscation, from the point of view of the same MATE attacker.

3. Software Protection Evaluation Framework

Our framework (first presented in [4]) is built on the premise that the strength of software protection is proportional to the effort the MATE attacker must spend breaking the protected code. In the following we present the steps of our general framework for characterizing the strength of software protection:

1. Survey different published approaches for achieving the goal of a MATE attacker.
2. Model all these approaches as one large attack net [6], which is a Petri-net depicting the different steps of each attack. The input of the attack net is the protected program and the output is the information needed by the MATE attacker, e.g. the secret key hidden inside the obfuscated program.
3. Select the best overall attack from the attack-net by empirical observations. This is facilitated by Petri-nets which allow running all attacks in parallel.
4. Model the steps of the best attack (represented as transitions of the Petri-net) as search problems. This enables the identification of the key parameters of the search, i.e. the most important parameters that affect the speed of the search.
5. Map the search parameters to code features of the program entered as input to the attack net. These are the features that must be changed by software protection transformations, in order to make MATE attacks slower.

In the following subsections we describe these 5 steps in more detail using the running example of bypassing license checks in protected programs.

3.1 Modeling MATE Attacks as Attack Nets

Each MATE attack is split into one or more subsequent actions or steps, which are called *transitions* (represented as rectangles) inside an attack-net (Petri-net) [6]. Each transition has at least one input and one output *place* (represented as circles), which hold the input and output information, respectively. Two transitions are linked by a common place if the output of the former transition is the input of the latter transition. Note that multiple transitions may share the same input or output place, if their input, respectively output have the same type. An attack-net starts with an input place (also called a source), which holds the program that is under attack. An attack-net ends with an output place (also called a sink), which holds the information representing the goal of the MATE attacker.

For the purpose of illustrating the steps of our framework enumerated above, we consider the MATE attack goal of bypassing the license check inside a software application. Such a check is meant to enforce the purchase of a license key if users want to use all features of a certain software. During our literature survey we have identified 5 different MATE attack techniques to achieve this goal, which are illustrated inside the attack-net from *Figure 3* and described in the following:

1. The first attack is to guess the right license key via Random Testing. This attack may be very expensive if the range of the key is large, because all possible keys need to be enumerated.
2. An alternative attack (described in Section 3.2), is to make the input for the license key symbolic and then employ symbolic execution and SMT-/SAT-solvers in order to find the license key.
3. One may also find a license key by searching for hard-coded strings inside the binary and then trying these values as license key inputs. However this attack will fail if the license key is not stored as a printable string or as soon as

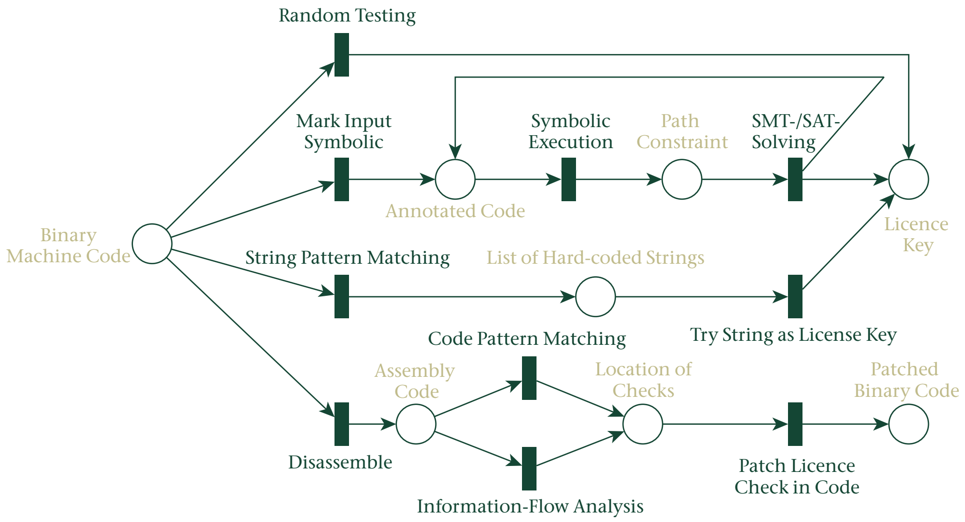


Figure 3: *Attack-net containing MATE attacks for bypassing license checks.*

any string obfuscation transformation (e.g. encode literals) is applied.

4. Instead of finding the license key an attacker could try to find and patch all the license checks by applying pattern matching on the disassembly of the program, using the assembly code representation of the license check. However, this attack will fail as soon as any obfuscation that breaks the code pattern is applied (e.g. virtualization obfuscation).
5. To be more robust against obfuscation, the MATE attacker could use taint analysis to identify the license checks. Once the checks are found, they need to be disabled via patching, which is difficult to automate, due to the multitude of ways in which a check can be represented in obfuscated code.

Note that we do not claim that these are all possible MATE attacks against bypassing license checks. However, these are the techniques we uncovered in the literature. If additional attacks are uncovered in the future they can easily be added to the attack-net from *Figure 3*.

3.2 Symbolic Execution as Best MATE Attack

Random testing, the top-most attack in *Figure 3*, does not scale if the license key is long and contains alphanumeric characters. Symbolic execution has problems if the license check is a cryptographically secure hash function, because the underlying SMT solvers cannot break such hash functions. However, such functions are easy to find via pattern matching and the they can be patched out. From the short description provided in the previous enumeration of attacks, we can also notice that the third and fourth attacks have significant weaknesses when it comes to analyzing obfuscated code. The last attack based on taint analysis is also problematic due to the fact that it does not scale when a large number of checks are used.

```
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) {
                  if (c != a) {
                           c = a + b
                  }
                  b = 1;
         }

         return 0;
}
```

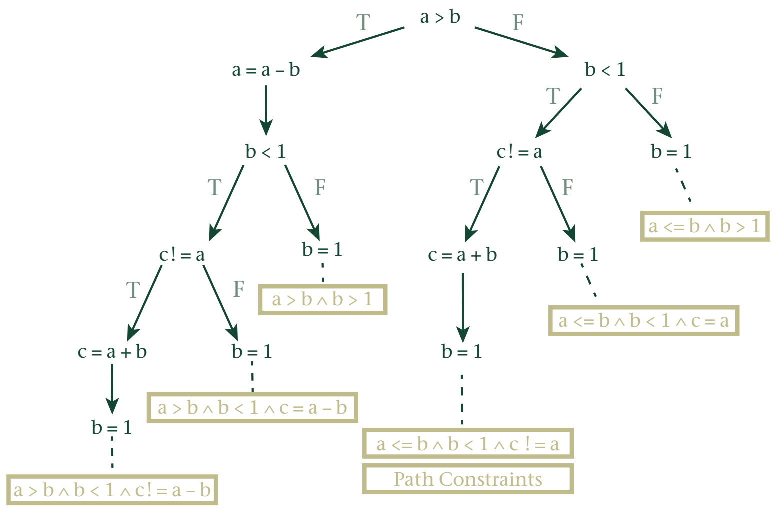


Figure 4 (right): *Random program.*

Figure 5 (left): *Symbolic execution tree with path constraints.*

Using self-developed or existing tools for each of these 5 MATE attacks, we empirically experimented with a small set of manually developed programs containing license checks. Based on these preliminary experiments and the previous arguments, we determined that the second attack from *Figure 3*, i.e. the attack based on symbolic execution is the fastest and most resilient to obfuscation. In the following we describe how symbolic execution works on a given program written in C.

The program in *Figure 4* consists of a single main function, which takes 3 command line arguments as inputs and assigns them to variables a, b and c. We mark these 3 variables as symbolic, which means that they no longer represent concrete values, but the range of values corresponding to their type. As a symbolic value is processed by program instructions, path constraints are added to it. The symbolic execution tree corresponding to this program is illustrated in *Figure 5*. Symbolic execution forks whenever there is a branch inside the code, which depends on at least one symbolic variable. At every fork, the state of the program is cloned together with the path constraints. The true branch state is appended with the constraint of the condition evaluating to true, while the other branch with the constraint evaluating to false. After such forks the path constraints are checked by an SMT solver, to verify if there is any possible assignment of concrete values to the symbolic variables, which could satisfy the path constraints. If so, the path is continued, otherwise it is discarded. Examples of path constraints are shown inside rectangles at the bottom of *Figure 5*. Note that values that satisfy these

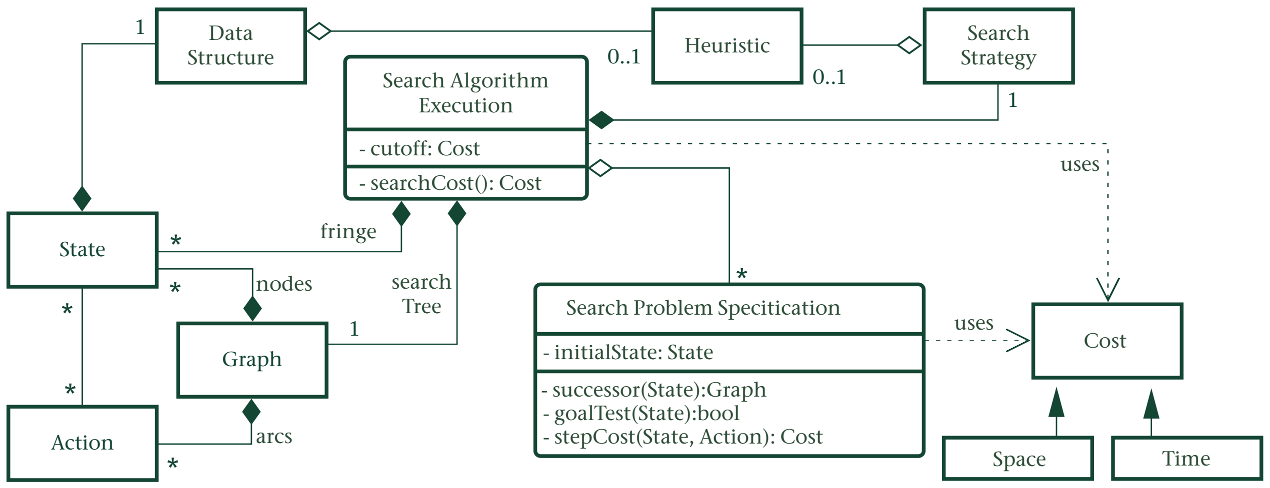


Figure 6:
Search Model.

constraints would lead the execution along the corresponding path.

If we consider that a license check is also a conditional statement dependent on the value passed as input as the license key, then symbolic execution will be able to find the correct value of the license key if we mark the license key input as symbolic. The symbolic execution will explore all execution paths and one of these paths contains the logic for the license check. The result of the SMT solver for that path is equal to the correct license key value. This attack was first presented by Banescu et al.[3]

3.3 Modeling MATE Attack Steps as Search Problems

The different steps of an attack (i.e. transitions of an attack net), can be formulated as search problems. The advantage of doing this is that there exists a vast literature regarding how to solve and quantify the effort of search problems and search algorithms, respectively.

The anatomy of a search problem and a search algorithm is represented in *Figure 6*. The fundamental part is the *data structure* on which the search is executing (e.g. an array of bytes representing machine code, a graph representing the control-flow graph), shown in the top-left corner. During search these data structures are annotated to show the state of the search. Therefore, the same data structure with different annotations, represents different *states*. From any state, one or more *actions* are possible to be performed on that state, which leads to a different state. The first state given in the *search problem specification* is called the *initial state* and it represents the root of the *search tree*. Each action taken in a certain state leads to a *successor* state in the search tree. As the *search algorithm execution* proceeds, the search tree keeps expanding, until a *goal* state is reached. The leafs of the search tree constitute the *fringe*. The size of the search tree indicates the number of visited states. Hence the cost of the search algorithm execution is proportional to the size of the search tree. The size of the search tree strongly depends on the chosen *search strategy* and its associated *heuristic*. Cost can be measured in terms of *space* or *time*.

The cost of a MATE attack is the sum of the search problem efforts of solving each *step* of the attack. Therefore, the strength of a software protection transformation can be quantified w.r.t. the effort increase of the MATE attack before and after that transformation is applied. Another advantage of formulating MATE attacks as search problems is that one obtains the code features which represent complexity factors for the search algorithms. By knowing these features, the software developer can apply the software protection transformation which targets exactly those features in order to slow down MATE attacks. For instance, in our running example of bypassing a license check via symbolic execution, the relevant code features are:

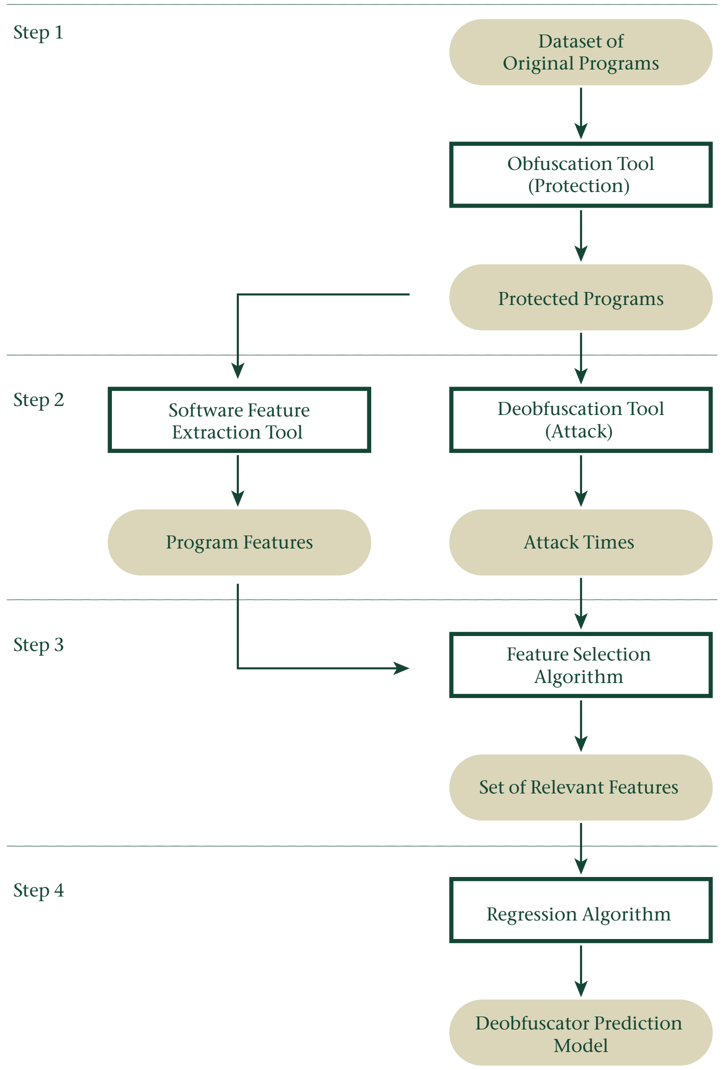
- The number of branches and loops depending on symbolic variables, because it determines the branching factor of the search tree.
- The number and complexity of Boolean and arithmetic operations, because it determines the complexity of the SMT queries corresponding to the path constraints.
- The data types of symbolic variables, because larger types increase the number of possible assignments made by the SMT solver to these variables.

4. Evaluation

To confirm that our approach has identified the most important code features, we follow the 4-step process depicted in *Figure 7*:

1. Generate a representative set of protected programs with variable values for all of the code features.
2. Record time needed by the best MATE attack and extract the features from the programs.
3. Select only the relevant features.
4. Build a regression model for predicting the time needed by the MATE attack against any given program.

In the following sections we describe each of these steps in more detail. Several of the



experiments described in the following sections are presented in more detail in [2].

4.1 Generating and Protecting Programs

For the purpose of creating a large dataset of programs, we have developed a C code generator, which was used to generate over 4500 random C functions having different code feature values. All of the generated programs mimic the structure of a license checker such that we can apply our MATE attack based on symbolic execution on each of these programs. However, before attacking these programs, we apply the 5 software protection transformations described in Section 2 and combinations of each pair of these transformations. Some transformations are applied twice in order to check the increase in attacker effort. In total we obtained 30 syntactically different, but semantically equivalent variants of each of the more than 4500 C programs.

4.2 Attacking Protected Programs

When applying the symbolic execution attack to each of the protected program versions we noticed that it was successful on all program variants [1]. However, the attack execution times varied greatly as a function of the obfuscation transformations which were applied to protect the program.

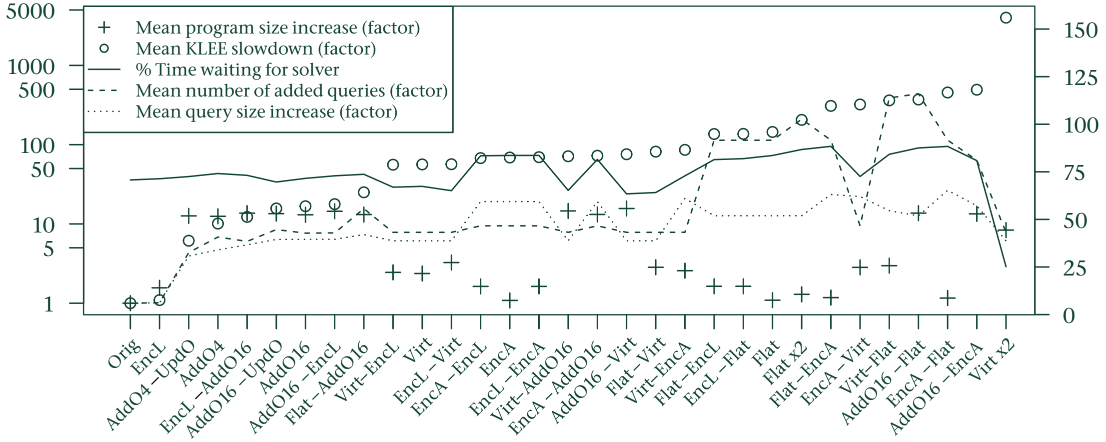
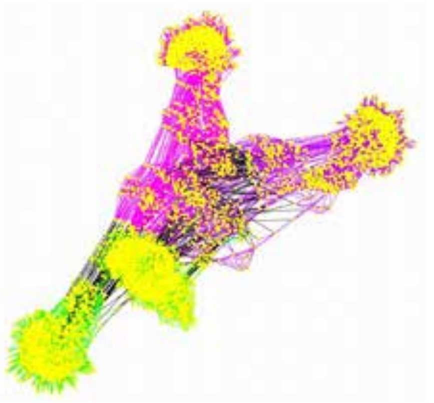


Figure 8 uses circles to show the average slowdown factor of the symbolic execution based attack (y-axis) on all programs obfuscated using different transformations and combinations thereof (x-axis). The left-most tick mark on the x-axis is the original program. The other tick marks represent obfuscated programs with the transformations presented in Section 2. *Figure 8* also shows the average increase in file size (plus signs), the percentage of attack execution time spent waiting for the SMT solver (solid line and right y-axis scale), the average number of queries sent to the SMT solver (dashed line) and the average increase in query size (dotted line). The most important observations from *Figure 8* are that:

- Contrary to expectations applying Encode Literals and Opaque Predicates alone, do not affect any of the code features we identified by our framework. This is because the dynamic nature of symbolic execution is able to bypass these software protection transformations.
- Virtualization increases the number of instructions, hence the number of operations during program execution.
- Control-Flow Flattening increases the number of branches, hence the number of queries sent to the SMT solver, by introducing more branches.
- Encode literals increases the size, hence the complexity of the queries sent to the SMT solver.

Figure 7 (S. 38):
Overview of evaluation experiment steps.

Figure 8:
Impact of different protection transformations on symbolic execution attack.



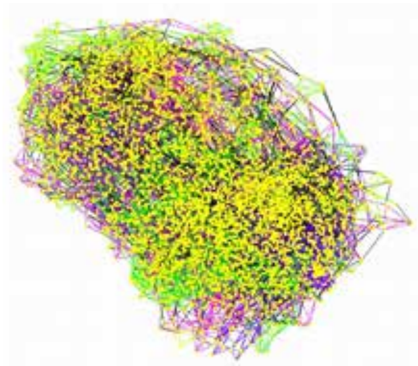


Figure 9 (top):
Before software protection.

Figure 10 (bottom):
After software protection.

4.3 Extracting and Selecting Program Features

We use existing tools to extract software features like code complexity metrics, resource usage and SAT features. SAT features are graph metrics applied to SAT instances represented as graphs, i.e. each literal is a node and each disjunction is an edge in the graph. For example the SAT instance of a non-obfuscated C program is illustrated in *Figure 9*, while the SAT instance of that same program after obfuscation using Flattening and Virtualization is illustrated in *Figure 10*. Notice that the community structures (separate groupings of nodes) in the graph are destroyed by these strong obfuscation transformations. Since we extracted a total of 64 code features, we perform recursive feature selection, which is able to reduce the number of features to 15, which correspond to the features identified by our framework at the end of Section 3.

4.4 Predicting Attack Times via Regression

Using the 15 features extracted in the previous step, we perform 10-fold cross validation using 4 state of the art machine learning (ML) algorithms: Support Vector Machines (SVM), Genetic Programming (GP), Random Forrest (RF) and Neural Networks (NN). *Figure 11* shows the normalized relative prediction error (y-axis) for each of the 4 ML algorithms, in a cumulative manner for the entire dataset of programs (x-axis). The maximum error is depicted with solid lines while the median error with dashed lines. It is important to notice that there are some differences between different ML algorithms and that RF has the lowest prediction error. Moreover, even for 85% of all programs the maximum prediction error of RF is less than 15% and the median error is less than 5%, which we believe is acceptable for predicting the time needed by a symbolic execution attack on any given program.

5. Conclusions and Future Work

In this paper we have presented a framework for evaluating the strength of software protection against MATE attacks. Our framework formulates attacks as search problems in order to identify the most relevant code features that will slow down the attack. To evaluate our attack we have generated thousands of C programs and protected them using popular obfuscation transformations. By recording the time needed by a symbolic execution based attack to bypass the license check of all protected programs, we were able to confirm that our framework has identified the most relevant code features for MATE attacks. Moreover, we used standard tools to extract other code features from all the programs. Recursive feature selection narrowed down the relevant features to the same features identified by our framework. We built regression models using these features, which were able to predict the time needed by the symbolic execution attack with high accuracy. This again confirms that our software protection evaluation framework has identified the most relevant code features.

In future work we plan to evaluate our framework for case studies based on different MATE attacks. Moreover we are interested in applying ML to automatically extract features relevant for slowing down MATE attacks, which would automate our software protection evaluation framework.

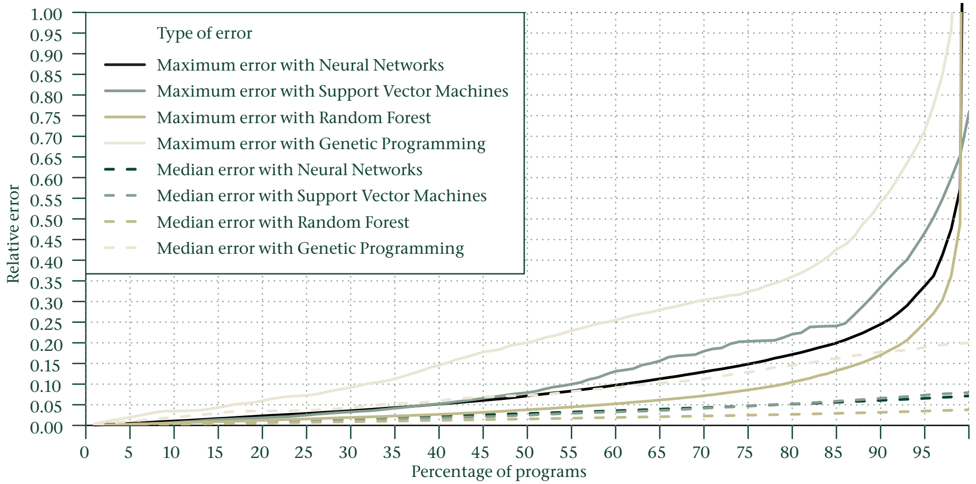


Figure 11:
Comparison of prediction error of different regression models.

References

[1] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In Proc. *of 2016 Annual Computer Security Applications Conference. ACM, 2016.*

[2] Sebastian Banescu, Christian Collberg, and Alexander Pretschner. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 661–678, Vancouver, BC, 2017. USENIX Association.

[3] Sebastian Banescu, Martín Ochoa, and Alexander Pretschner. A framework for measuring software obfuscation resilience against automated attacks. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop* on, pages 45–51. IEEE, 2015.

[4] Sebastian-Emilian Banescu. *Characterizing the Strength of Software Obfuscation Against Automated Attacks.* Dissertation, Technische Universität München, München, 2017.

[5] Christian Collberg and Jasvir Nagra. Surreptitious software. *Upper Saddle River, NJ: Addision-Wesley Professional,* 2010.

[6] James P McDermott. Attack net penetration testing. In *Proceedings of the 2000 workshop on New security paradigms*, pages 15–21. ACM, 2001.