A Framework for Measuring Software Obfuscation Resilience Against Automated Attacks

Sebastian Banescu, Martín Ochoa and Alexander Pretschner
Technische Universität München, Germany
Email: {banescu, ochoa, pretschn}@cs.tum.edu

***Abstract*—Software obfuscation of programs, with the goal of protecting against attackers having physical access to the machine executing them, is a common practice motivated by the necessity of keeping intellectual property (such as business critical algorithms) and critical data (such as cryptographic keys) secret. However, as of today, it is unclear how secure popular obfuscation operators are relative to each other or to other protection techniques. In this paper we propose a formal framework to characterize attacker models and guarantees, inspired by similar notions from cryptography. We then map prior work in the area of deobfuscation to our formal model to the possible extent. We also perform a case-study about using symbolic execution for deobfuscation, concretely mapped onto our formal model.**

I. Introduction

Although traditionally deemed as a bad security practice, software obfuscation of programs is a practical necessity in many contexts, and is widely used [7]. Typical goals of obfuscation include protection against attackers having physical access to the machine executing a certain program, motivated by the necessity of keeping intellectual property (such as business critical algorithms) and critical data (such as cryptographic keys) secret. Recent theoretical breakthroughs in cryptography such as *indistinguishability obfuscation* [16] are promising because they offer provably secure obfuscation, but are still far from being practical [1]. Moreover, it is unclear if the provably secure guarantees of *indistinguishability obfuscation* correspond to the intuitive goals practitioners have in mind when obfuscating their programs. Also, it is unclear how secure commonly used obfuscation operators are. As a consequence, practical software obfuscation is merely intuitively secure, and there exist no guidelines that give an idea on what amount of effort is needed to break them. In our view, this situation resembles the early days of cryptography, and to some extent even the current state of certain cryptographic primitives. Although the introduction of reduction proofs for cryptographic constructions has increased the confidence in the security of certain algorithms, note that for many popular cryptographic algorithms such as AES there exists no reduction proof to a hard problem. On the other hand, reduction proofs shift the problem of the security of a primitive to the hardness of a problem, which is not provable but conjectured (such as the Diffie-Hellman decisional problem).

Despite the absence or presence of reduction proofs, probably the most useful outcome for practitioners is a table such as the one described in the *EcryptII* project [23], which has an estimate of key sizes for different algorithms that provide protection for a certain period of time according to the computational strength of attackers. Implicitly this sort of table assumes the existence of a *best attacker* against a certain cryptographic construction. For instance, the best attacker for 256-bit AES was considered to be brute-force, however, [3] presents a better attacker that has 2^{131} time complexity.

Section II introduces a formal framework to characterize attacker models and guarantees, inspired by similar notions from cryptography. The core observation of the framework is that, given an explicit notion of the best attacker against a certain obfuscation operator, it is possible to derive empirical guarantees on the effort needed to break it. Section III gives examples of published results for concrete attacker models. In Section IV we present a case-study on using open source symbolic execution and obfuscation tools for one of the attacker models, paving the way for further investigations in this direction for other obfuscation operators. Our final goal is to develop a table similar to the one from the *EcryptII* project. However, we would indicate the time needed to break an obfuscation transformation depending on the size of configuration space of the transformation and the characteristics of the program being transformed, instead of the time needed to break a cryptographic cipher with a certain key size.

II. Formal Model

Let $\mathcal{P}$ be the universe of all executable programs. Let $[\![\cdot]\!]_{BB} : \mathcal{P} \rightarrow (\mathcal{I} \rightarrow \mathcal{O})$ be the semantic characterization of the *black-box* behavior of any program, where $\mathcal{I}$ and $\mathcal{O}$ represent universal input, respectively output domains. Let $\mathcal{T}$ be the universe of all obfuscation transformations applicable to programs in $\mathcal{P}$. An obfuscation transformation $\tau \in \mathcal{T}$ is a mapping $\tau : \mathcal{P} \rightarrow \mathcal{P}$ such that $[\![p]\!]_{BB} = [\![\tau(p)]\!]_{BB}$.

Software obfuscation mainly aims to protect either the original *control-flow of a program* (i.e. intellectual property of the vendor) or *data embedded in a program* (e.g. secret keys, hardcoded passwords). These goals can be mapped directly to 2 different attackers: $\mathcal{A}_{CF}$ and $\mathcal{A}_{D}$, corresponding to recovery of control-flow, respectively data, which obfuscation transformations aim to hide. We are aware that, given enough resources, an attacker can recover the secret hidden by many obfuscation transformations. The overarching goal of our work is to show how good certain obfuscation transformations are against different automated attackers.

Intuitively, a defender is chiefly interested in a quantifiable expression over his/her program and all attackers, saying that

attacking a particular obfuscation transformation is bounded below by a certain work-factor. However, we believe this to be a very lofty goal. On the other hand, an attacker is mainly interested in developing an attack which outperforms any prior known attacks, especially if these attacks are not efficient in the attacker's context. We believe that these 2 perspectives complement each other. Therefore, we propose a model for quantifying obfuscation resilience by stipulating a – possibly hypothetical, unknown, non-computable – lower bound. This reflects the perspective of the defender's interests. In practice, we are confined to providing *single data points*; their values define *upper bounds for the lower bounds*. These are interesting if a defender can conclude that even though she may not know the lower bound, available data already suggests that the obfuscation mechanism is too weak, assuming that $\mathcal{A}_{CF}$, $\mathcal{A}_D$ are the *best attackers* according to today's knowledge.

A. Automated Control-Flow Recovery Attacks

Let the semantic characterization of control-flow recovery be denoted by $[\![\cdot]\!]_{CF} : \mathcal{P} \rightarrow \mathcal{CF}$, where $\mathcal{CF}$ represents the universe of program control-flow graphs (CFGs). The goal of an automated attacker $\mathcal{A}_{CF}$ (having a fixed amount of computational power s) is to minimize the result of a function t that measures the time needed by $\mathcal{A}_{CF}$ to recover a CFG c, which is closer than a threshold value δ to the CFG of the original program $[\![p]\!]_{CF}$ according to some metric dif_{CF}. This can be formally written as:

$$t[\mathcal{A}_{CF}(\tau(p), s) = c \in \mathcal{CF} \mid dif_{CF}(c, [\![p]\!]_{CF}) < \delta] \geq T_{CF}(s, \tau(p)),$$

where $T_{CF}(s, \tau(p))$ is the shortest time needed by an attacker having power s, to recover the CFG in a program p, obfuscated with transformation $\tau \in \mathcal{T}$. Examples of $\mathcal{A}_{CF}$ will be given in Section III.

B. Automated Data Recovery Attacks

Similarly to $\mathcal{A}_{CF}$, for $\mathcal{A}_D$ we first define the semantic characterization of data recovery from a program by $[\![\cdot]\!]_D : \mathcal{P} \rightarrow \mathcal{D}$, where $\mathcal{D}$ represents the universe of all data items that can be extracted from any program's binary or process memory at various points in time. The goal of the automated attacker $\mathcal{A}_D$ (having a fixed amount of computational power s) is to minimize the result of the same function t that measures the time needed by $\mathcal{A}_D$ to recover a data item d, which is closer than a threshold δ to the data ($[\![p]\!]_D$) hidden in the original program p. This can be formally written as:

$$t[\mathcal{A}_D(\tau(p), s) = d \in \mathcal{D} \mid dif_D(d, [\![p]\!]_D) < \delta] \geq T_D(s, \tau(p)),$$

where $T_D(s, \tau(p))$ is the shortest time needed by an attacker having power s, to recover the hidden data in a program p, obfuscated with transformation $\tau \in \mathcal{T}$. Examples of $\mathcal{A}_D$ will be given in Section III.

III. Mapping Prior Works Onto Our Formal Model

In this section we present several automated attacks presented in the literature and we map them to the two types of attacks presented in Section II for particular obfuscation transformations.

a) Virtualization obfuscation: (denoted $\mathcal{T}_v$) aims to hide the control-flow of a program $p \in \mathcal{P}$ through the means of program translation into a random language and building an interpreter for it. The following three steps concisely describe the process of virtualization obfuscation: (1) Generation of a random bytecode instruction set architecture (ISA) covering all instructions of p; (2) Translation of p into a bytecode program written in the previously generated ISA; (3) Generation of an emulator which can interpret the generated bytecode program on the native machine ISA (e.g. x86). This emulator contains handlers for each instruction opcode of the random ISA.

After performing these steps, the obfuscated program, consisting of the bytecode and the emulator, are distributed to the end-users. Note that the bytecode ISA consists of instructions each characterized by a random opcode and one or more operands. One instruction in the bytecode ISA may correspond to one or more instructions in the native ISA (e.g. x86). This also means that multiple locations in the original binary may correspond to the same instruction in the bytecode ISA, hence the same opcode handler or the emulator.

An automated attack $\mathcal{A}_{CF}$ on $\tau_v(p)$, where $\tau_v \in \mathcal{T}_v$ is a concrete virtualization implementation, consists in recovering the original CFG of p by: (1) identifying the bytecode inside $\tau_v(p)$, (2) mapping each bytecode instruction to its corresponding emulator handler and (3) disassembling the bytecode program into a common language (e.g. assembly code). Obtaining the CFG from assembly code is straight-forward.

A few automated attacks against $\mathcal{T}_v$ have been published in the literature [22], [17], [12], [18]. The work of Guillot and Gazet [17] based on concolic execution and the work of Kinder [18] based on abstract-interpretation are the only ones that offer an open implementation of such an automated attacker. Both make the assumption that the location of the bytecode and internal variables are known. We applied these implementations to several small C programs obfuscated with the Tigress[1] virtualization obfuscator. However, we could not replicate the results of the authors. We believe that the main cause behind our unsuccessful replication of the experiments from [17], [18] was due to the aforementioned assumption made in these works, which did not hold for the latest version of the Tigress obfuscator. Therefore, we aim to implement our own tool based on a symbolic execution engine like S^2E [8] leveraging the ideas presented by Sharif et al. [22], which do not assume that the location of the bytecode and internal variables are know. The approach of Sharif et al. [22] aims to first find the location of bytecode via dynamic analysis and then performs static analysis and symbolic execution of the emulator handlers to recover the original CFG.

The aforementioned works on deobfuscating virtualization do not present a performance evaluation, with the exception of Kinder [18]. He shows that even for a 10-line C program which computes the Fibonacci sequence, deobfuscation may take up to 40 seconds and around 100 MBs of memory if one wants to obtain a CFG close to the original one. Unfortunately,

[1] http://tigress.cs.arizona.edu/

nothing is said regarding the relationship between the size of the program and the time needed for deobfuscation. However, we expect deobfuscation to be at least linear in the size of the original program. Therefore, depending on the size of p, $\tau_v(p)$ may be good to withstand such automated attackers for a certain amount of time. Due to the lack of a performance evaluation we cannot fully map the previous works to the model presented in Section II, because of the missing time bound for the automated attacker.

b) Opaque Predicates: (denoted $\mathcal{T}_o$) are boolean expressions whose value is constant and known to the one who inserts it, however they are difficult to statically evaluate by an attacker [11]. Opaque predicates are used to insert bogus branches into a program $p \in \mathcal{P}$, thus hiding its original control-flow. Opaque predicates can be added at arbitrary positions in the code, but also appended to existing conditional statements such as `if`s or loops. We denote a program obfuscated with α randomly generated opaque predicates as $\tau_o^{\alpha}(p)$, where $\tau_o \in \mathcal{T}_o$.

The approach of Preda and Giacobazzi [14] based on abstract interpretation tackles the problem of automatically recovering the original control flow graph by eliminating opaque predicates from obfuscated instances. Unfortunately, there is no open implementation of this approach available. Rolles offers a partial implementation of this approach, which is based on the Pandemic program analysis framework which has not been published [21].

Another opaque predicate deobfuscation approach has been published by Gabriel [15]. He uses the Miasm framework[2] to recover the CFG of a 10-line C function containing 3 if-statements and no loops. This program is obfuscated using the Obfuscator-LLVM[3] tool. Gabriel uses pattern matching based on disjunction operators in if-statement conditions: when an if-statement condition contains an OR, the right-hand side operator is assumed to be an opaque predicate and it is eliminated. This assumption does not hold in general, but it holds for the target program transformed by the *Bogus Control Flow* transformation of Obfuscator-LLVM. Since the strong assumptions made by Gabriel prevent general automated attacks, we plan to implement our own tool using a satisfiability modulo theory (SMT) solver such as STP [6] along the lines of the sample OCaml code offered by Rolles [21].

c) White-Box Cryptography: (denoted $\mathcal{T}_w$) is a data hiding obfuscation transformation, which hides the secret key of a cryptographic cipher in software, without the need of a trusted computing base. The first white-box cryptographic technique for DES and AES were proposed by Chow et al. [10], [9]. Other techniques for AES were proposed by Bridger et al. [4] and Xiao et al. [28]. We denote a program containing a hard-coded secret key which has been obfuscated using white-box cryptography as $\tau_w(p)$, where $\tau_w \in \mathcal{T}_w$ is a concrete white-box cipher instance.

[2]https://github.com/cea-sec/miasm

[3]https://github.com/obfuscator-llvm/obfuscator

The goal of the attacker is to recover the secret key denoted $[\![p]\!]_{\mathcal{D}}$ from the cipher; such attacks have been proposed [2], [27], [19], [20]. The only assumption of these attacks is that the precise location and structure of the lookup tables in the binary is known. Although, this assumption can be broken by combining white-box cryptography with other obfuscation transformations, many works focus on attacking white-box cryptography alone. Given the contents and structure of the lookup tables of a white-box AES cipher instance from [9], the attack described in [20] can recover the hidden 128-bit key, with a work factor of 2^{22}. This is an *upper bound on the lower bound* (denoted $\leq_{\mathcal{E}}$) for the work factor of automatic secret key recovery attacks against this white-box AES cipher i.e.

$$t[\mathcal{A}_D(\tau_w(p), s) = d \in \mathcal{D} \mid dif_D(d, [\![p]\!]_D) = 0] \leq_{\mathcal{E}} 2^{22}/s,$$

where $\mathcal{A}_D$ is described in De Mulder et al. [20], $\tau_w(\cdot)$ is described in Chow et al. [9], dif_D is bitwise equality comparison and s is the power of the attacker's CPU.

d) Encoding Literals: (denoted by $\mathcal{T}_{el}$) "breaks-down" all constants into pieces that are put together by a sequence of instructions. The implementations of this obfuscation transformation vary widely, since one integer or string constant assignment can be substituted by a arbitrary number of instructions, which when executed result in the same constant that they replaced. Plus these instructions can also be further substituted by other equivalent instruction sequences [26].

Some works on deobfuscating various instances $\tau_{el} \in \mathcal{T}_{el}$ [17], [15] use heuristics from compiler optimizations to eliminate superfluous instructions and reduce the sequence of instructions to the shortest possible sequence. Generally, this process is not time-consuming although it is super-linear in the number of instructions of the obfuscated program. Unfortunately, the authors of the previously mentioned works do not perform a performance evaluation of their attack. This is probably due to the high flexibility regarding the different implementations of τ_{el} and its strong dependence on the program being obfuscated. Nevertheless, due to this reason we cannot map these works onto the model from Section II. However, we will see an example of attacking one τ_{el} implementation in Section IV.

IV. Case-study: Using KLEE for Automated Data Retrieval

In Section III we argued that many prior works fit into the model described in Section II. However, the majority of these works do not quantify the time needed to attack obfuscated software instances. Therefore, in this section we wish to provide an illustrative case study where we use the *Tigress Diversifying C Virtualizer*[4] as an obfuscation tool. It is freely available and provides a large set of configurable obfuscation transformations ($\tau \in \mathcal{T}$) such as: virtualization obfuscation at function-level, opaque predicates, literal encoding and many others. We set the goal of the attacker

[4]http://tigress.cs.arizona.edu/

to be data recovery and we choose the KLEE symbolic execution engine [5] as a concrete instance of $\mathcal{A}_D$. We chose a couple of license-checker programs which are generally protected against tamper-proofing attacks, however, they are also concerned with hiding data, i.e. the hard-coded license checks, such that attacker cannot simply guess valid license keys. The goal of $\mathcal{A}_D$ is to find concrete values for license keys, instead of buying such keys from the software vendor.

In this section we probe KLEE as a candidate for *best attacker* on obfuscated license checkers, i.e. check if it finds valid license keys and how long it takes to do so. We describe our experimental setup including our hardware power (s) such that our experiments can be replicated.

A. Symbolic Execution and KLEE in a Nut-shell

Before presenting the details of the case study, we give a brief introduction about symbolic execution and the motivation behind using KLEE as an automatic data recovery attacker.

The idea behind symbolic execution is to simulate execution of a program under test using "symbolic" inputs that can take all values allowed by a certain type (e.g. $x \in [0, 255]$) instead of concrete inputs (e.g. $x = 2$). Whenever a conditional branch bearing a predicate (π) which depends on the values of a symbolic input is encountered by the simulation, the internal state of the simulator and its execution are forked into 2 paths, one following the true-branch (π) and the other following the false-branch ($\neg\pi$). The 2 execution paths are pursued independently thereafter. For instance if $\pi = x < 128$ then the true-branch would continue with the so called *path condition* $x \in [0, 127]$, while the else branch would continue with *path condition* $x \in [128, 255]$.

Note that if a program has several symbolic inputs then all of them are part of the path condition. After encountering several branches the simulation generates a so-called *execution tree*, where every inner-node represents a predicate that has caused the execution to fork. Each leaf of this tree contains a path condition that can be sent to a constraint solver e.g. SMT solver, which returns a set of concrete values. These concrete values returned by the solver constitute a test case which when given to the program under test will exercise exactly the corresponding path from the execution tree. Then, a test suite is therefore automatically obtained by sending path conditions of all the leaves from the execution tree to the constraint solver. The test cases are less redundant than those generated by fuzzing, because each test-case is guaranteed to exercises a different path through the program.

This brings us to the reason why we believe that a symbolic execution tool would be a perfect data extraction attacker for programs which perform license checking. In a simplified view, a license check can be seen as a predicate π, which either evaluates to true or false. Therefore, the output of the SMT solver that satisfy the constraints leading to the true branch of the license check represent a concrete license key value for the attacker.

KLEE is a symbolic execution engine for LLVM bitcode. It directly interprets LLVM instructions and maps them to constraints without making approximations about their semantics (i.e. offering bit-level accuracy of process memory). In order to handle calls to the C library, KLEE either replaces the concrete call with a symbolic model of the C library if available. However, it also performs concrete calls if a symbolic model is not available. For KLEE this model is obtained via a customized version of μclibc, a minimal implementation of the C library for embedded systems. This library was modified such that calls to file-IO functions can use buffers that hold symbolic data instead of actual files.

The automatic data recovery attack for retrieving the secret value from a program p using KLEE consists of the following steps: (1) compile p to LLVM bitcode, (2) run the bitcode with KLEE which results in a set of test-cases (one of which contains the password), (3) replay each test-case on the target program using the *klee-replay* tool and (4) pick the test-case that causes p to produce the output we are interested in, e.g. the output that is different from a "license incorrect" message.

B. Experiment Setup

The experiments described in this paper were performed on the following setup. We used Tigress version 1.3 and a recent version of KLEE checked-out from Github[5]. We built KLEE with LLVM version 3.4.2 on a 64-bit Ubuntu 14.04.1 VM having a 2.8 GHz CPU frequency with 1 core, and 4 GB of memory. This hardware configuration also indicates the power (s) of the attacker $\mathcal{A}_D$. We also built a recent version of STP[6] which KLEE uses as a constraint solver, as well as a recent version of the KLEE specific μclibc library[7], which provides KLEE with a symbolic model of the POSIX runtime. To reduce the size of the LLVM bitcode being executed by KLEE we used the `--optimize` parameter.

C. A Simple License Checking Program

Listing 1. Simple license checking program.

```
1 int main(int argc, char* argv[]) {
2   if (strcmp(argv[1], "my_license_key") == 0)
3     printf("The license key is correct!\n");
4   else
5     printf("The license key is incorrect!\n");
6   return 0;
7 }
```

The first target program we choose is a simple license checking program p_1 which contains a hard-coded license key illustrated in Listing 1. p_1 checks whether its first input argument is equal to the hard-coded key (line 2 in Listing 1) and prints out a message representing the comparison outcome. Therefore, the semantic characterization of data recovery from program p_1 is $[\![p_1]\!]_D$ ="my_license_key". If we build the binary executable of p_1 and apply the `strings` tool to it, the output is a list of 14 hard-coded strings found in the binary, 3 of which are the strings from lines 2, 3 and 5 in Listing 1.

[5]https://github.com/klee/klee/commit/e72b75e
[6]https://github.com/stp/stp/commit/e2cf0c6
[7]https://github.com/klee/klee-uclibc/commit/a8af87c

We obfuscated p_1 using the *Encode Literals* transformation (τ_{el}) from Tigress, which replaces all integer and string literals with one function containing opaque expressions. When this function is called with a certain parameter value, it returns a value corresponding to one of the literals which were replaced. We cannot show the C source code of $\tau_{el}(p_1)$ in this paper because it has between 250 and 300 lines of code depending on whether the structures used to compute opaque expressions are arrays or lists. However, we offer the source code online[8]. If we build the binary executable of $\tau_{el}(p_1)$ and apply the `strings` tool to it, the output no longer includes the strings from lines 2, 3 and 5 in Listing 1. However, note that the Tigress user guide indicates that the string encoding function is trivial by design and it should be further virtualized. By looking both at the C source code and at the disassembled binary code in *IDAPro*[9], we could confirm this claim. An high-level overview of the CFG of the binary is illustrated in Fig. 1. It shows an `if-else-if`-statement with 4 branches corresponding to: the encoding of the strings from lines 2, 3 and 5 in Listing 1 through simple character concatenation and an `else`-branch which returns the empty string.

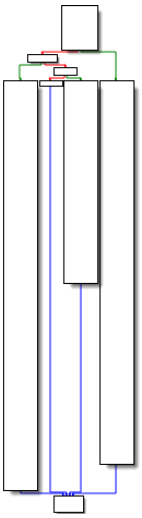

Fig. 1. CFG of string encoding function in $\tau_{el}(p_1)$.

To further raise the bar for the attacker, we applied the *Function Virtualization* transformation (τ_v) from Tigress on the encoding function from $\tau_{el}(p_1)$ with the default parameters. We write this as $\tau_v(\tau_{el}(p_1))$. The source code of this program has over 1300 lines of C code and it is harder to deduce the values of the encoded strings from it in comparison to $\tau_{el}(p_1)$. To support this claim we show in Fig. 2 the overview of the CFG of the string encoding function as extracted by *IDAPro* from the binary corresponding to $\tau_v(\tau_{el}(p_1))$. Therefore, in the following we only present the attack on $\tau_v(\tau_{el}(p_1))$, since $\tau_{el}(p_1)$ is trivial to deobfuscate.

D. Automated Data Recovery from the Obfuscated License Checking Program

As a preliminary check for semantic equivalence of the obfuscated program $\tau_v(\tau_{el}(p_1))$ with the original program, we tested the obfuscated binary with different input values and observed that its behavior is the same for both p_1 and $\tau_v(\tau_{el}(p_1))$, i.e. $[\![p_1]\!]_{BB} = [\![\tau_v(\tau_{el}(p_1))]\!]_{BB}$. This was also confirmed by the output of KLEE when executed on the LLVM bitcode of p_1 and of $\tau_v(\tau_{el}(p_1))$, which could not find an input for which the two programs return different outputs. For executing KLEE we enabled the symbolic POSIX runtime and indicated that the target programs have a symbolic argument

[8]https://www.dropbox.com/s/90q5xlxymsw2bcs/spro.zip
[9]https://www.hex-rays.com/products/ida/

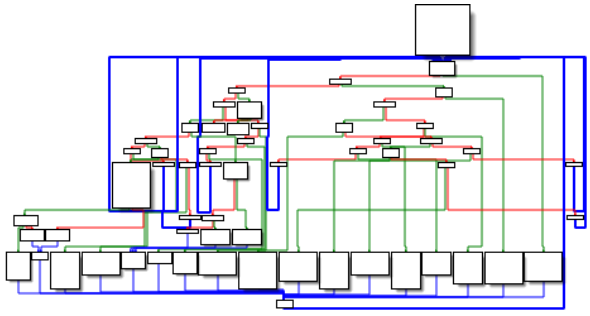

Fig. 2. CFG of string encoding function in $\tau_v(\tau_{el}(p_1))$.

of 32 bytes, which is an average length for a license key in general, even though our key is only 14 bytes.

The only difference between the 2 programs is their execution time, which is longer for the obfuscated program as expected due its larger size. This also led to a longer symbolic execution of the obfuscated program using KLEE. In the case of p_1, KLEE ran for about 0.9 seconds on average, while for $\tau_v(\tau_{el}(p_1))$ KLEE ran for 1.5 seconds on average. Each of these 2 KLEE runs generated a test-suite consisting of 2 test-cases covering disjunct paths through the code of the 2 programs. Using the `klee-replay` tool we replayed the 2 test cases generated by KLEE when applied to $\tau_v(\tau_{el}(p_1))$. We observed that one of the test-cases caused $\tau_v(\tau_{el}(p_1))$ to print the message on line 3 of Listing 1, while the other test-case resulted in the output of message from line 5.

This means our automated attacker KLEE has managed to extract a test-suit in 1.5 seconds, in which a test-case contains the secret ("my_license_key") from the obfuscated program $\tau_v(\tau_{el}(p_1))$. Replaying the test cases and selecting the one that outputs "The license key is correct!", took less than a tenth of a second. Therefore, KLEE was able to extract d from $\tau_v(\tau_{el}(p_1))$ such that it is identical to the hidden secret key, i.e. $dif_D(d, [\![p_1]\!]_D) = 0$. Putting it all into our formal model introduced in Section II, we write

$$t[\mathcal{A}_D(\tau_v(\tau_{el}(p_1)), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 1.5sec,$$

where $\mathcal{A}_D$ is KLEE, s is described in Section IV-B and dif_D is string equality comparison. The transformations are the one's from Tigress and we have used their default parameters.

The beauty of virtualization obfuscation is that it can be applied recursively as many times as desired. Each time it is applied, it creates a larger source code file, and hence a slower execution time. For instance, reapplying virtualization to the already virtualized $\tau_v(\tau_{el}(p_1))$ results in the program denoted by $\tau_v^2(\tau_{el}(p_1))$ having over 3300 lines of C code. However, KLEE requires 8.8 seconds on average to generate the same test-suite as before and find the secret key, i.e.

$$t[\mathcal{A}_D(\tau_v^2(\tau_{el}(p_1)), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 8.8sec.$$

Virtualizing the program once more results in the program denote by $\tau_v^3(\tau_{el}(p_1))$ having over 6600 lines of C code and an average automatic secret recovery time of over 13 minutes:

$$t[\mathcal{A}_D(\tau_v^3(\tau_{el}(p_1)), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 13min.$$

The trade-off between performance overhead and security of virtualization obfuscation is visible from these experiments.

Another feature offered by Tigress in order to further obfuscate the emulator of the virtualization transformation is adding a number of opaque predicates to each instruction handler. Intuitively, adding opaque predicates would cause KLEE to trigger more calls to the SMT solver, hence slowing it down. However, we have found that the impact of adding opaque predicates is minor with respect to the average automatic data recovery attack time and major with respect to file size. We denote adding α opaque predicates to each instruction handler of a virtualization transformation as $\tau_o^{\alpha}(\tau_v(\cdot))$. Note that $\tau_o^0(\tau_v(\tau_{el}(p_1))) = \tau_v(\tau_{el}(p_1))$, which we presented earlier. We generated programs for $\alpha \in \{1, 5, 10, 20\}$, having 1355, 1605, 2076, respectively 7300 lines of C code. Running the previous attack resulted in the following upper bounds:

$$t[\mathcal{A}_D(\tau_o^1(\tau_v(\tau_{el}(p_1))), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 1.5sec,$$

$$t[\mathcal{A}_D(\tau_o^5(\tau_v(\tau_{el}(p_1))), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 1.6sec,$$

$$t[\mathcal{A}_D(\tau_o^{10}(\tau_v(\tau_{el}(p_1))), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 1.7sec,$$

$$t[\mathcal{A}_D(\tau_o^{20}(\tau_v(\tau_{el}(p_1))), s) = d | dif_D(d, [\![p_1]\!]_D) = 0] \leq_{\mathcal{E}} 2.3sec.$$

E. A More Complex License Checking Program

One may argue that the reason for KLEE's success in the previous automatic data recovery attack is due to the nature of license checking used on line 2 of the program from Listing 1. Realistic license checking programs do not simply compare their given input with a hard-coded input; they apply one-way hash functions to the input and compare the result with a hard coded value instead. To check if KLEE is able to automatically recover the license key for a more complex license checking program we modified the program from Listing 1 by adding the DJB2 hash algorithm [29] to it and replaced the comparison with the hard-coded string with a comparison with the hash value of "my_license_key". The resulting program (p_2) can be seen in Listing 2. The reason for selecting DJB2 is its compact implementation in C, its non-invertibility and efficiency.

Listing 2. License checking program using the DJB2 hash function.

```
1  int main(int argc, char* argv[]) {
2    unsigned long hash = 5381;
3    unsigned char *str = argv[1];
4
5    while (int c = *str++)
6      hash = ((hash << 5) + hash) + c;
7
8    if (((hash >> 32) == 0xbc150c6e) &&
9        ((hash & 0xffffffff) == 0x49a54935))
10     printf("The license key is correct!\n");
11   else
12     printf("The license key is incorrect!\n");
13   return 0;
14 }
```

KLEE was able to find inputs for p_2 which lead to the output of the message "The license key is correct!". However, these inputs were not equal to "my_license_key", instead they are different hash values such as `I)_NpMy1Aa!G`, which lead to a collision with "my_license_key" for the DJB2 hash function. KLEE found 21 such inputs that collide to the same hash value as "my_license_key" for DJB2. Although this does not mean that KLEE found all the collisions, it does lead to an interesting application of such symbolic execution tools for the purpose of testing collision resistance of cryptographically secure hash functions. This observation also helps us define the difference function dif_D for this type of programs (which check hashes of their inputs), as a collision detection function that compares the hash of its inputs, instead of directly comparing its inputs as done in the previous experiment.

Running p_2 under KLEE with the same command line parameters as for the previous experiment took around 15 minutes on average instead of less than 1 second as was the case for p_1. We also applied virtualization obfuscation to p_2 and the resulting program $\tau_v(p_2)$ had 360 lines of C code. We ran the same data recovery attack as described in the previous sub-sections against $\tau_v(p_2)$ using KLEE. As expected the attack took around 56 minutes on average, i.e.

$$t[\mathcal{A}_D(\tau_v(p_2), s) = d | dif_D(d, [\![p_2]\!]_D) = 0] \leq_{\mathcal{E}} 56min.$$

Going further with an even more complex license checking program, we implemented the white-box AES cipher described by Chow et al. [9]. We picked a random key which was embedded in the look-up tables of our implementation and computed the AES encryption of the string "my_license_key" using that same key and we obtained the ciphertext c. Afterwards we changed the string comparison on line 2 in Listing 1 to a comparison between the white-box AES encryption of the input argument and c. The resulting program, denoted $\tau_w(p_1)$, had 4875 lines of code, 90% of which were hard coded look-up tables. This led to a large LLVM bitcode file of over 3 MB, which is 3 times the size of the largest bitcode file corresponding to $\tau_v^3(\tau_{el}(p_1))$, which we have presented in our previous experiments. After running $\tau_w(p_1)$ for 12 hours without finding the key, we decided to stop it since it is far from being the *best attacker* against τ_w.

This experiment shows both that symbolic execution can be used for performing data extraction attacks on more complex programs such as hash functions, but also that it has issues with scalability when the program under analysis grows.

V. Related Work

Many related works which focus on developing techniques for automatic control-flow and data recovery attacks have already been mentioned in Section III. However, we are also aware of related works which aim to formally characterize attackers with respect to obfuscation transformations. Dalla Preda [13] models attackers of obfuscation transformations as abstract domains expressing certain properties of program behaviors. Since obfuscation transformations are characterized by the most concrete preserved property, the complete lattice of abstract domains allows comparing obfuscation transformations with respect to their resistance against various attackers. However, contrary to our model, an obfuscation transformation is either effective against an attacker or not, regardless of the

time needed by the attacker for deobfuscation. Our model focuses in quantifying the effort needed for deobfuscation.

Ceccato et al. [7] present a set of experiments for quantifying the *potency* of obfuscation transformations against human understanding, using various code metrics. As mentioned in this work, there is no strong correlation between the value of code metrics and *resilience* of obfuscation against automated de-obfuscation attacks [11]. However, Sutherland et al. [24] and Udupa et al. [25] proposed alternative metrics to measure the *resilience* of particular obfuscation transformations against attacks. Our work is similar, in that we focus on resilience against automatic attacks, not potency against manual attacks.

VI. Conclusions and Future Work

In this paper we have proposed a formal framework to evaluate the strength of software obfuscation aiming at protecting the CFG or secret data of a program. We show how this strength evaluation can be applied to existent obfuscation techniques, and discuss how it relates to the idea of *best attacker*. We empirically instantiate our model for concrete automatic data recovery attacks for the obfuscation operators presented earlier in the paper, depending on random seeds of different length (or other suitable security parameters), and produce a first instance of the reference table for attackers with different computational power.

One important observation from our experiments is that our *best attacker* candidate, KLEE strives to achieve instruction coverage and stops once it is at 100%. However, for data recovery attacks such as the one presented in Section IV-D we need 100% branch coverage to be certain that we have extracted the license key. This leaves room for future work in the area of tooling based on symbolic execution for automated data extraction attacks, which we plan to pursue.

The case study we performed started from the assumption that we have the LLVM bitcode of the application we want to test. In reality attackers generally have access to the binary, although it is not uncommon for attackers to also have access to Java bytecode in the case of Android applications. Nevertheless, as our next steps we plan to further explore and extend tools based on abstract interpretation (e.g. Jakstab) and symbolic execution (S^2E) which are directly applicable to binaries. We also intend to perform experiments using other obfuscation tools such as Obfuscator-LLVM and commercial obfuscation tools.

References

[1] S. Banescu, M. Ochoa, A. Pretschner, and N. Kunze. Benchmarking indistinguishability obfuscation - a candidate implementation. In *Proc. of 7th International Symposium on ESSoS*, number 8978 in LNCS, 2015.

[2] O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a white box AES implementation. In *Selected Areas in Cryptography*, number 3357 in LNCS, pages 227–240. Springer Berlin Heidelberg, Jan. 2005.

[3] A. Biryukov, D. Khovratovich, and I. Nikolić. Distinguisher and related-key attack on the full aes-256. *Advances in Cryptology-CRYPTO 2009*, pages 231–249, 2009.

[4] J. Bringer, H. Chabanne, and E. Dottax. White box cryptography: Another attempt. *located at, last visited on Jul*, 22(2011):14, 2006.

[5] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2):10:1–10:38, Dec. 2008.

[7] M. Ceccato, M. D. Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19(4):1040–1074, Feb. 2013.

[8] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. ASPLOS XVI, pages 265–278, New York, NY, USA, 2011. ACM.

[9] S. Chow, P. Eisen, H. Johnson, and P. C. V. Oorschot. White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*, number 2595 in LNCS, pages 250–270. Springer Berlin Heidelberg, Jan. 2003.

[10] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot. A white-box DES implementation for DRM applications. In *Digital Rights Management*, pages 1–15. Springer, 2003.

[11] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[12] K. Coogan, G. Lu, and S. Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 275–284, New York, NY, USA, 2011. ACM.

[13] M. Dalla Preda. *Code obfuscation and malware detection by abstract interpretation*. PhD thesis, University of Verona, 2007.

[14] M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *Third IEEE International Conference on Software Engineering and Formal Methods.*, pages 301–310. IEEE, 2005.

[15] F. Gabriel. Deobfuscation: recovering an OLLVM-protected program. http://blog.quarkslab.com/deobfuscation-recovering-an-ollvm-protected-program.html, 2014. Quarkslab, Accessed: 2014-01-20.

[16] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. In *Proc. of the 54th Annual Symp. on Foundations of Computer Science*, pages 40–49, 2013.

[17] Y. Guillot and A. Gazet. Automatic binary deobfuscation. *Journal in computer virology*, 6(3):261–276, 2010.

[18] J. Kinder. Towards static analysis of virtualization-obfuscated binaries. In *19th Working Conference on Reverse Engineering (WCRE)*, pages 61–70, Oct 2012.

[19] W. Michiels, P. Gorissen, and H. D. L. Hollmann. Cryptanalysis of a generic class of white-box implementations. In R. M. Avanzi, L. Keliher, and F. Sica, editors, *Selected Areas in Cryptography*, number 5381 in LNCS, pages 414–428. Springer Berlin Heidelberg, Jan. 2009.

[20] Y. D. Mulder, B. Wyseur, and B. Preneel. Cryptanalysis of a perturbated white-box AES implementation. In *Progress in Cryptology - INDOCRYPT 2010*, number 6498 in LNCS, pages 292–310. Springer Berlin Heidelberg, 2010.

[21] R. Rolles. Control Flow Deobfuscation via Abstract Interpretation. https://www.openrce.org/blog/view/1672/Control_Flow_Deobfuscation_via_Abstract_Interpretation, 2011. OpenRCE, Accessed: 2014-01-20.

[22] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 94–109, May 2009.

[23] N. Smart. ECRYPT II Yearly Report on Algorithms and key-sizes (2011-2012), 2012. http://www.ecrypt.eu.org/documents/D.SPA.20.pdf.

[24] I. Sutherland, G. E. Kalb, A. Blyth, and G. Mulley. An empirical examination of the reverse engineering process for binary files. *Computers & Security*, 25(3):221–228, 2006.

[25] S. Udupa, S. Debray, and M. Madou. Deobfuscation: reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering*, 2005.

[26] H. S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[27] B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In *Selected Areas in Cryptography*, number 4876 in LNCS, pages 264–277. Springer Berlin Heidelberg, 2007.

[28] Y. Xiao and X. Lai. A secure implementation of white-box AES. In *2nd International Conference on Computer Science and its Applications, 2009. CSA '09*, pages 1–6, 2009.

[29] O. Yigit. Hash Functions. http://www.cse.yorku.ca/~oz/hash.html. York University, Accessed: 2014-01-27.