Obfuscation: Where Are We in Anti-DSE Protections? (a first attempt)

Mathilde Ollivier
CEA, LIST,
Paris-Saclay, France
mathilde.ollivier2@cea.fr

Sébastien Bardin
CEA, LIST,
Paris-Saclay, France
sebastien.bardin@cea.fr

Richard Bonichon
CEA, LIST,
Paris-Saclay, France
richard.bonichon@cea.fr

Jean-Yves Marion
Université de Lorraine, CNRS, LORIA
Nancy, France
Jean-Yves.Marion@loria.fr

ABSTRACT

Obfuscation is widely used to protect software against *man-at-the-end* attacks. Recent attacks based on semantic methods, especially *dynamic symbolic execution* (DSE), have proven extremely powerful against standard obfuscation techniques, leading several teams to investigate anti-DSE protections. Yet, the domain is in its infancy, and the current state of research on the topic is quite unclear. We propose a systematic review of anti-DSE techniques. In particular, we propose a classification and identify strengths and weaknesses of the current lines of research, as well as promising future directions.

ACM Reference Format:
Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. Obfuscation: Where Are We in Anti-DSE Protections? (a first attempt). In *Software Security, Protection, and Reverse Engineering Workshop (SSPREW9), Decembre 9–10, San Juan, PR, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3371307.3371309

1 INTRODUCTION

Context. *Man-at-the-end* attack (MATE) designs a general attack scenario where the attacker has access to the program's executable code as well as to the entire execution environment. The attacker typically looks for stealing sensitive data (e.g., credential, private keys) or sensitive code (secret algorithms), or for directly tampering with the code (e.g., bypass security checks or licence). Reverse-engineering an unprotected program may not be difficult for seasoned experts, thus making it important to *protect* the code, i.e., make it hard to understand. *Obfuscation* [18, 19] precisely aims at protecting software against such MATE attacks.

While standard static and dynamic disassembly techniques cannot cope with current obfuscation methods, recent attacks based on advanced code analysis methods have proven very powerful against standard obfuscation methods [13, 21, 29, 30, 45], leading Schrittwieser *et al.* to ask whether *"Obfuscation can keep pace with progress in code analysis"* [34]. In particular, attacks based on Dynamic Symbolic Execution [15, 35] (DSE), collectively coined as Symbolic Deobfuscation [4, 8, 13, 32, 40, 45], have been particulary successful, as they benefit from both the robustness of dynamic analysis and the inference ability of (semantic) static analysis.

Problem and goal. New protections are clearly needed, especially against DSE-based attacks. However the current state of anti-DSE protections is pretty unclear. Several techniques have been proposed [31, 36, 39, 46], but there is no clear classification nor comparison. In particular, Schrittwieser *et al.*'s survey [34] does not apply well to DSE, as it separates static methods from dynamic ones – while the power of DSE exactly comes from a tight combination of both of them – and the discussion is too generic to understand precisely the issues inherent to DSE.

We thus aim to propose a systematic survey of anti-DSE protections as well as elements of classification and comparison. We also want to offer a clear vision of the available implementations as well as their — theoretical or practical — performance and robustness.

Contributions. We investigate currently proposed protections and identify their practical weaknesses and strengths. In more details,

- We summarize the state of knowledge and classify existing protections hindering dynamic symbolic execution,
- We compare these protections using several key parameters — strength, cost, stealth, availability of implementation, etc.;
- Finally, we conclude by pointing out some deficiencies in the current state-of-the-art, and propose a short-term call for action and longer term research directions to remedy them.

The aim of this paper is thus to set the ground for further systematic studies of anti-DSE protections. Moreover, as code protections is a vast topic, we believe some of the deficiencies pointed out here, and some of the proposed solutions, may be of general interest for other classes of software protections.

2 BACKGROUND

2.1 Obfuscation

Obfuscation [19] aims at hiding a program's behavior or protecting proprietary information such as algorithms or cryptographic keys by transforming the program to protect $\mathcal{P}$ into a program $\mathcal{P}'$ such that (1) $\mathcal{P}'$ and $\mathcal{P}$ are semantically equivalent, (2) $\mathcal{P}'$ is roughly as efficient as $\mathcal{P}$, and (3) $\mathcal{P}'$ is *harder* to understand. While it is still unknown whether applicable theoretical criteria of obfuscation exist [6, 7], practical obfuscation techniques and tools do exist [18]. Let us present briefly a few such important techniques.

Virtualization* and *Flattening [42] transform the control flow into an interpreter loop dispatching every instruction. Virtualization even adds a virtual machine interpreter for a custom bytecode program encoding the semantics of the original program. Consequently, the visible control flow of the protected program is very far from the original control flow. Virtualization can be nested, encoding the virtual machine itself into another virtual machine. While highly expensive in terms of runtime and code size overhead, the technique is very powerful against human attackers.

Packing* and *Self-modification insert new instructions or modify existing ones at runtime. These techniques seriously damage static analyses by hiding the real instructions. However, extracting the hidden code can be done dynamically [24, 28].

Opaque predicates [20] are branching conditions whose predicates either always evaluate to true or always evaluate to false. Hence, while the real program behaviors always follow the same side of the condition, analyzers (human or tools) may be lured into considering spurious dead code as legitimate, wasting time and efforts trying to understand it. Two-way opaque predicates are a variant where both sides of the condition lead to functionally equivalent code.

2.2 Dynamic Symbolic Execution

Symbolic execution [15] simulates the execution of a program along its paths, systematically generating inputs for each new discovered branch condition. This exploration process consider inputs as *symbolic variables* whose value is not fixed. SE follows a path and each time a conditional statement involving the input is encountered, it adds a constraint to the symbolic value related to this input. Solving these *path constraints* (a.k.a. path predicates) automatically – typically with off-the-shelf SMT solvers [41] – then allows to generate new input values leading to new paths, progressively crawling the set of paths of the program – up to a user-defined bound. Since covering *all* paths is in general infeasible, SE usually tries to cover all branches and instructions. The technique has seen a strong renewed interest in the last decade to become a prominent bug finding technique [15, 16, 26].

When the symbolic engine cannot perfectly handle some constructs of the underlying programming language — like system calls or self-modification — the symbolic reasoning is interleaved with a dynamic analysis allowing meaningful (and fruitful) approximations – *Dynamic Symbolic Execution* [26]. Typically, (concrete) runtime values are used to complete missing parts of path constraints that are then fed to the solver through *concretization* [22]. This feature allows the approach to be especially robust against complicated constructs found in obfuscated binary codes, typically packing or self-modification, making DSE a strong candidate for automated deobfuscation: it is as *robust* as dynamic analysis, with the additional ability to *infer* trigger-based conditions.

KLEE [14] is a popular source-level DSE tool. Binary-level DSE tools include Angr [37], Binsec [23], Fuzzball [3], S2E [17] and Triton [33].

3 SYMBOLIC DEOBFUSCATION

Deobfuscation is a specific case of reverse engineering where the attacker seeks to simplify or remove the protections embedded in the program in order to achieve one of the following goals [34]:

(1) Locate code or data;
(2) Retrieve the CFG;
(3) Explore code to understand behavior;
(4) Retrieve the original code.

Symbolic deobfuscation. The term *Symbolic deobfuscation* refers to deobfuscation methods based on Dynamic Symbolic Execution. They are especially powerful as they combine both the reasoning ability of (static) semantic methods with the robustness of dynamic approaches, enabling them both to survive protections such as packing or self-modification *and* to infer input triggers.

Typical DSE-based attacks include finding rare behaviors [13, 23, 45] (triggers, secret, etc.) of the whole program as well as local exhaustive exploration (for precise CFG recovery [9], local proofs [8, 40] or simplifications [32]). In the case of simplifications, DSE can be complemented by standard static simplification techniques, such as forward tainting [35], backward slicing [38] or compiler-like rewriting rules [32, 45].

Attacks. Recent attacks based on DSE include the following publications:

- Brumley *et al.* [13] is probably the first work to leverage DSE for deobfuscation, actually malware comprehension. They point out the ability of DSE to bypass complex standard protections such as packing or self-modification, and to recover trigger-based behaviors;
- Yadegari *et al.* [45] proposes a generic approach to deobfuscation, independent from the exact protection mechanism, based on dynamic analysis, symbolic reasoning, tainting, slicing and local simplifications. The technique is able to greatly simplify simple programs protected by well-known commercial protection tools — Themida, CodeVirtualizer, Execrytpor and VMProtect;
- Salwan *et al.* [32] presents an approach based on DSE, tainting and compiler optimizations for simplifying protected codes. The technique succeeds in automatically solving the Tigress challenge [1], managing to recover unprotected code functionally equivalent to the original one with similar (or even smaller) size. The technique has also been demonstrated to be useful over a large range of virtualization techniques;
- Bardin *et al.* [8] proposes a so-called *backward-bounded* variant of DSE in order to identify several classes of protections, including opaque predicates and call stack tampering. The technique has been successfully applied over a few benign packed programs and an APT-class malware. Doser [40] employs similar techniques for identifying and simplifying duplicate codes and two-way opaque predicates.

While not directly providing new attacks, Banescu *et al.* [4] shows that standard obfuscation methods are very weak against DSE-based attacks. Finally, other attacks based on DSE or close concepts have been developped by other authors [27, 29].

Versus standard protections. Most standard obfuscation techniques are ineffective against DSE-based attacks. Packing and self-modification are handled by the dynamic part of DSE [13, 45], opaque predicates can be efficiently identified and removed [8], virtualization can be completely simplified away [32, 45], and trigger-based behaviors can be inferred [13]. Actually, the systematic studies by Banescu *et al.* [4] and Salwan *et al.* [32] (focused on virtualization) demonstrate that DSE is mostly not affected by standard protections. The only exception is nested virtualization (3 layers), but in that case the defender must be ready to pay a high price in term of runtime overhead (up to ×100).

4 CLASSIFICATION OF ANTI-DSE TECHNIQUES: A RATIONAL

Families of anti-DSE weaknesses. While very successful in practice, symbolic execution suffers from a few issues and weaknesses that may make the analysis fail or dramatically slow down. It turns out that, naturally, anti-DSE protections try to leverage such weaknesses in order to prevent symbolic deobfuscation attacks. Anand *et al.* [2] proposes to classify the weaknesses of DSE into three main categories: (1) complex constraints; (2) path divergence and (3) path explosion. Let us describe more precisely these three categories.

Complex constraints Path constraints can be hard — if not impossible — to solve for a SMT solver. For example, specific non-linear operations such as multiplication or division substantially increase the complexity of path constraints. This issue is critical to symbolic deobfuscation because if a path constraint cannot be solved, then it reduces the subset of feasible paths that the analysis is able to explore;

Path divergence Computing precise path constraints can be difficult for certain programs. If path constraints are not computed precisely by the symbolic execution engine it can lead to path divergence: feasible paths are missed or unfeasible paths are taken by mistake. Either way, the symbolic analysis is not able to give a correct set of paths for the program;

Path explosion To symbolically explore the set of paths of a program, DSE needs to solve the constraints for each path and store all pending states in memory. For these two reasons it rapidly becomes overwhelming to explore a large subset of paths. Realistically, only a reduced amount of paths can be explore in a limited amount of time. An attacker thus has to wisely select the subset of paths he wants to explore to avoid high analysis duration.

Key characteristics. Now that we have a rational criterion to classify the different anti-DSE protections, the next step is to identify a set of relevant characteristics and metrics for evaluating these protections. We take inspiration from metrics proposed by Collberg *et al.* [19] for generic obfuscation, namely: potency (improvement of program complexity from a human perspective), resilience (resistance to automated attack), stealth (ability to remain undetected) and cost (runtime or code size overhead). Yet, we modify them slightly as we feel that the case of obfuscation techniques specifically designed to prevent some form of automated analysis is different from the generic case. We thus propose the following characteristics:

- **Correctness** evaluates whether the technique preserves the semantics of the original program;
- **Strength** is the ability to break or delay DSE-based attacks;
- **Cost** describes the runtime and code size overhead;
- **Resilience** is the ability to resist against automated attacks.

As most anti-DSE protections are proven correct, we will discuss this point in the remaining part of the paper only when it is an issue. We present complex constraints in Section 5, path divergence in Section 6 and path explosion in Section 7. A summary is given in Section 8.

5 COMPLEX CONSTRAINTS

The first category of obfuscations against DSE consists in complexifying the constraints sent to the SMT solver. The goal is to increase the time needed by the solver to resolve the constraints such that analysis is not possible for an attacker with limited ressources within a restricted amount of time.

5.1 Mixed Boolean Arithmetic

Method. The first protections focusing on complex constraints is *mixed boolean arithmetic* expressions [46]. This protection replaces simple arithmetic expressions throughout the code with more complex expressions that have both arithmetic and boolean operators. Symbolic analysis generates constraints when it encounters input-dependent conditional statements which are then sent to a SMT solver to generate new input values to explore new parts of the code. The goal is to make these constraints much more difficult to solve — ideally impossible within a restricted amount of time. Thus the attacker is unable to access new parts of code: exploration is hindered.

Strength and Cost. Solving expressions with a SMT solver is already known to be a NP-hard problem. However, in practice, solvers do offer good results when solving exploration constraints generated by a symbolic execution engine [12]. There is, as of now, no general result indicating that solving MBA expressions is significantly harder than solving more standard expressions. Moreover, experimental results seem to be contradictory: some of them do not really report any strong protection [4, 31] (Tigress encodig of MBA), while other do [25]. From our own experience, MBA are very powerful against *simplification queries* (when one asks whether two expressions are equivalent [25]), but they offer a more underwhelming protection in the case of *exploration queries* (when one asks for satisfiability of a path constraint [4, 31]).

The cost of MBA expressions appears to stay minimal in some recent experiments [31], however in that case the protection offered is rather weak. It would be interesting to have a systematic study of cost for large and complex MBA expressions.

Stealth and Mitigation. MBA inserts very typical constructs in the code, namely deep combinations of bitwise and arithmetic operations. These expressions are easy to spot by an attacker.

Eyrolles *et al.* [25] propose to counter MBA expressions by simplifying them using arithmetic simplification coupled with a library of equivalent MBA expressions. The method alternates steps of substitutions of expressions using the library and arithmetic simplification until the original MBA cannot be simplified anymore. This method is able to simplify hard MBA expressions in a few seconds. However it is not clear whether this method is efficient against MBA expressions created by unknown patterns.

5.2 Cryptographic Hash Functions

Method. Sharif *et al.* [36] propose to replace an equality check in an `if` statement with the equality over the corresponding values encoded using a cryptographic hash function. As these functions are (believed to be) irreversible, it is impossible for the solver to retrieve the trigger value. Moreover the following block of code is encrypted using the input value as the key such that attackers are unable to reverse the executable code.

Strength and Cost. This protection is extremely powerful, as it relies on strong cryptographic assumptions. DSE engines will not solve such queries, save for a dramatic breakthrough in cryptanalysis. Moreover, in [36], the relevant part of the code is encrypted using this input value, the attacker cannot analyze any of the decrypted instructions. Hence, this obfuscation does not leak any information about the program's original behavior and data.

Encryption is a very expensive transformation. It is thus not realistic to protect the entire code with it. Only very specific functions can be protected using this obfuscation for the overhead induced by decryption routines would rapidly render the code too inefficient. If hash functions are used without encryption the cost is significantly lower. However the protection would thus be effective only for trigger-based behaviors because otherwise the dynamic aspect of DSE would be able to retrieve the input value associated with the hash during any execution.

Stealth and Mitigation. The technique introduces cryptographic routines and encrypted instructions, it is thus fairly straigthforward to detect and localize within the executable code.

The major issue with this protection is its somewhat limited scope, as it is specialized in hidding triggers. While there are some important applications (malware, remote attestation), many legitimate programs cannot take the intended execution path only under specific circumstances from the environment or network. Maybe the technique could be adapted to the generic case by relying on white-box cryptography [10] – but this is another form of MATE scenario.

6 PATH DIVERGENCE

Obfuscations exploiting the path divergence weakness aim to take advantage of a mismatch between the concrete execution of the program under attack and the semantics of the underlying DSE engine. The symbolic engine may be tricked into either missing feasible paths or taking infeasible ones.

6.1 Self-Modifying Code

Method. The first category of protection is self-modifying code. This transformation inserts instructions at source or assembly level that will modify the executed code on-the-fly at runtime. A well-known application of self-modification protections is packers where, at runtime, an unpacking routine replaces visible instructions inserted by the obfuscation with the original instructions of the programs. While not a specific anti-DSE technique, we still include it as it hinders standard semantic static analysis.

Strength and Cost. In principle, self-modification should not be a problem for DSE: several works [11, 13, 45] have shown how to take advantage of the dynamic analysis component of DSE to handle it. Yet, in practice, many DSE tools lag behind state-of-the-art principles and still do not support self-modification. For example, only Angr [37] does have an option to handle self-modification, while KLEE [14], Triton [33] or Binsec [23] do not. But defenders must be aware supporting self-modification is not a hard feature to add for binary-level DSE engines: it is mostly an engineering effort.

Regarding cost, while a few scattered self-modifications are essentially inexpensive, full program (un-)packing may come at a huge runtime cost, and must be used with care, for example through partial code packing, or one-time unpacking.

Stealth and Mitigation. Self-modifying code is arguably stealthy against static analysis. Packed binaries for example do not have a common structure implying to the analyst that the visible code is not the original. Yet, a dynamic analysis will easily reveal self-modification instructions by monitoring the write instructions within the executable part of memory. Mitigations exist [11, 13, 45] and are discussed above.

6.2 Symbolic Code

Method. Yadegari *et al.* [44] propose a specific type of self modification called symbolic code. This obfuscation relies on the fundamental principals of self-modification, yet it is specifically designed to hinder dynamic symbolic execution. Indeed, symbolic code adds an instruction that triggers the self-modification only if a certain value of an input is entered. As this instruction is not a conditional statement, the symbolic engine does not see that a different code could be executed with a different input value. This obfuscation is used to hide existing trigger-based behaviors in programs.

Strength and Cost. This obfuscation has a trigger-based behavior that encodes a `jmp` instruction ahead of the program counter. If the trigger condition is not met the symbolic analysis will not see the `jmp` instruction and thus will not even try to generate an input for the alternate path. Currently, DSE engines supporting standard self-modification cannot cope with this protection, even though Yadegari *et al.* [44] proposes a solution (again, current implementations lag behind state of the art principles).

Runtime cost is expected to be low as the protection does not add many instructions.

Symbolic code needs a trigger-based condition in the original code which is not always the case in legitimate programs.

Stealth and Mitigation. Even if symbolic code is a specific type of self-modification, it is much stealthier because the code is rewritten only when the trigger condition is met. A standard dynamic analysis will not detect the self-modification. A mitigation proposed by Yadergari *et al.* [44] can defeat it using bit-level taint analysis.

6.3 Covert Channels

Method. Stephens *et al.* [39] propose an obfuscation using covert channel to hide information flow and trick the symbolic execution engine into missing relations between variables in the program. This obfuscation uses the program's runtime system or operating system to reroute visible information flow. For example, it can use system's timers to silently pass information (typically, value of a variable) through a covert channel. Concretely, the primitives uses *slow* (*resp. fast*) threads to transfer the value 0 (*respectively* 1) of the n^{th} bit of the variable. The symbolic analysis is thus not able to know which variables are symbolic and then misses feasible paths. Several primitives are proposed by the authors and more could be imagined.

Strength and Cost. State-of-the-art symbolic engines cannot cope with this obfuscation. Typically, those relying on tainting will miss some input dependent variables, and current symbolic engines do not model precisely physical ressources such as time or cache.

Regarding cost, primitives based on `JIT` and filing caching are much more expensive than the other primitives presented, taking between $\times 10^2$ and $\times 10^4$ more time to execute than the original code. No code size increase should be observed as the protection only inserts a few instructions. Yet, it is not clear how this would impact runtime execution.

Finally, note that this obfuscation is not correct *per se*, as covert channels are not perfectly reliable. The authors coin their technique as *probabilistically correct*, arguing that the vast majority of the executions over the protected code will indeed have the expected correct behavior.

Stealth and Mitigation. This protection is sensitive to covert channels detection techniques such as clock perturbation and system call-based anomaly detection. Clock perturbation significantly increase the runtime overhead but is not able to defeat the obfuscation whereas system call-based anomaly detection would mark a *dummy* version of the protection as anomalous. However, some modifications to the implementation of the primitives would reduce the risk of the obfuscation to be discovered using this technique.

7 PATH EXPLOSION

These techniques create a path explosin in order to make DSE-based exploration intractable.

7.1 Path-Oriented Protections

Method. The first category of protections based on path-explosion is *path-oriented protections*. A first primitive was proposed by Banescu *et al.* [4], before the general principle was formalized by Ollivier *et al.* [31], with new primitives. The goal of path-oriented protections is to introduce input-dependent conditional statement. Three principal primitives have been proposed: (1) Range Divider [4], which consists of a conditional `if` or `switch` statement with obfuscated versions of the original code in each conditional branches. (2) For [31], which replaces an input-dependent assignment with a `for` statement containing simple arithmetic operations. (3) Write [31] replaces the same expressions as For but with self-modifying code thus combining the strengths of self-modification and path-oriented protections.

Strength and Cost. Ollivier *et al.* [31] present several results about the strength and cost of path-oriented protections. A formal analysis indicates that these methods can offer a strength exponential to the size in bits of the input space.

Some primitives — Range Divider with `switch` — increase the size of the code as these conditional statements take up a lot of space. However, other primitives — such as For or Split— do not substantially modify the code size nor the execution time for large real-world programs.

Stealth and Mitigation. Path-oriented protections do not introduce any exotic operators or control-flow structures as conditional statements are common in genuine code. Some primitives however — such as Range Divider with `switch` — may be easier to spot within a code as they use jump tables stored in specific sections of the executable code. The Write primitive cannot be detected statically but will be very easily discovered using dynamic analysis. Furthermore, the technique may be amenable to pattern attacks, hence several variants and diversity are required.

Path explosion is a long-standing issue in DSE. Actually, some solutions have been proposed, such as path merging, but experiments show that state of the art implementation are useless here. It would be important to evaluate the difficulty needed to customize existing symbolic execution tools to simplify this type of obfuscations.

7.2 Linear Obfuscation

Method. Linear obuscation, proposed by Wang *et al.* [43], aims at concealing a trigger value by inserting a very specific form of input-dependent conditional loop based on a mathematical conjecture about a convergent sequence (typically, Syracuse). The conjecture ensures that the loop always finishes and converges to the same value, but DSE will have a hard time finding the right path.

Strength and Cost. The strength of linear obfuscation is based ont the input-dependant loop. The condition of the loop contains a *spurious* variable crafted from a real user input. The use of an unsolved conjecture that always converges to the same value ensures the obfuscation will not change the semantics of the program while symbolic execution engines will not be able to determine beforehand how many iterations of the loop are needed. Thus, DSE has to create one new path for each iteration, quickly increasing the time needed to explore.

This protection can introduce a runtime overhead as the number of loop iterations at execution may vary depending on the input value. This number will vary depending on the exact mathematical conjecture used. Syracuse seems fine, for it converges in less than 1000 iterations for all integers up to 10^{58}, but what about other conjectures?

A last issue, not discussed in the original paper, is that the Syracuse conjecture deals with mathematical integers, while computers works with machine integers. It is not clear how the conjecture behaves in that case.

Stealth and Mitigation. The authors argue that the code inserted by their technique has a common control-flow structures, thus making the code hard to spot. Still, Syracuse-like conjectures requires unusual arithmetic operations (modulo 3 or 5 for some extensions, possibly arithmetic over arbitrary integers) which are easy to spot.

The main threat to linear obfuscation is pattern attacks, as there exist only a few mathematical conjectures well suited for this method. The authors proposed to counter it using diversity. Also, note that while recovering the exact trigger by DSE only is indeed difficult, a simple local domain-based reasoning does allow to recover a very good approximation of the range of possible values for the trigger. In other words, the technique leaks information about the trigger.

8 SUMMARY

All information discussed in Sections 5 to 7 are summarized in Table 1. Some information is added: the availability of an implementation and experimental evaluation. Strength, cost and stealth are rated with a three-level scale. The value is either decided with theoretical reasonning or validated by (more or less extensive) experiments when the † or †† symbols are present. When in doubt, we add the ? symbol.

In general, anti-DSE articles focus mainly on strength, hence strength is easy to assess – even though some proposal are not backed with experimental evaluation. The situation is much less satisfactory regarding cost and stealth: these characteristics are not always discussed, and only very rarely experimentally evaluated. Especially, evaluating the cost of complex constraints is hard to determine beforehand, even though it is likely to be significant for cryptographic hash functions. Note that we write symbolic code as stealthier than standard self-modification because it is hidden with a trigger. Regarding correctness, we point out that covert channels are *probabilistically* correct.

Finally, we can note that, unfortunately, most obfuscation techniques lack an available reference implementation.

Overview of experimental evaluations. Table 2 gives an overview of experimental evaluations done in anti-DSE papers *conducting experimental evaluation*. We mainly report the characteristics of the benchmark programs (number of programs, synthetic or real, size), as well as whether or not they reuse existing benchmarks, and the DSE attackers these experiments consider. Experiments for linear obfuscation only concerns cost (code size increase) and security concerns. Note that we add the work by Banescu *et al.* [4] as a baseline: even though the corresponding experiments are not primarily about anti-DSE protections, this article clearly contributes to level up the expectations in terms of benchmarks for DSE-based attacks.

9 DISCUSSION

While several defenses have been proposed – and some of them clearly shown highly powerful in practice – we can highlight a few drawbacks common to many of these studies:

- Many protection implementations are not available, causing issues in terms of reproducibility and comparisons;
- Too many studies do not propose strong enough experimental evaluation;
- Many studies consider only protection strength, while cost and stealth are also highly relevant.

Call for action. We propose the following agenda to quickly improve the research on anti-DSE protections:

- Make the implementation of protections available, or at least some protected samples;
- Use common benchmarks, or at least try to reuse benchmark programs from other teams;
- Include discussions and experimental evaluations of cost and stealth.

Directions for future research. It seems that current anti-DSE research focuses on general attack scenarios. We feel it is time for the community to try to tackle more precise attack scenarios, taking for example into consideration: (1) a *precise model of the attacker*: including the precise goal (CFG recovery, data localization, etc.), the degree of precision that the goal must achieve, the kind(s) of attack (computing power, off-the-shelf tools or crafted ones, fully automated or interactive) and (2) a *precise model of the defender*, including the constraints in terms of cost, stealth and correctness.

10 CONCLUSION

In the context of obfuscation, we propose a systematic review of anti-DSE techniques. Especially, we propose a classification and identify strengths and weaknesses of the current lines of research, as well as promising future directions. In the light of our study, it appears that several promising techniques have been proposed, but only few of them are available in obfuscation tool. Moreover, many studies focus only on strength of protection, while other crucial points such as cost and stealth are only rarely taken into account. Finally, too many papers come with no or weak experimental evaluation. As a *call for action*, we feel that the community should investigate some of the following points: availability of the different protections, standardized benchmarks, a general reflexion on goals of obfuscation and constraints of attackers and defenders.

REFERENCES

[1] Tigress challenge. http://tigress.cs.arizona.edu/challenges.html.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 2013.

[3] Domagoj Babic, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 12–22, 2011.

[4] Sebastian Banescu, Christian S. Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Annual Conference on Computer Security Applications, ACSAC 2016*, 2016.

[5] Sebastian Banescu, Christian S. Collberg, and Alexander Pretschner. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *USENIX Security Symposium*, 2017.

[6] Boaz Barak. Hopes, fears, and software obfuscation. *Commun. ACM*, 59(3):88–96, 2016.

[7] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Advances in Cryptology - CRYPTO*, 2001.

[8] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-bounded DSE: targeting infeasibility questions on obfuscated codes. In *2017 IEEE Symposium on Security and Privacy, SP*, 2017.

[9] Sébastien Bardin and Philippe Herrmann. OSMOSE: automatic structural testing of executables. *Softw. Test., Verif. Reliab.*, 21(1), 2011.

[10] M. Beunardeau, A. Connolly, R. Géraud, and D. Naccache. White-box cryptography: Security in an insecure environment. *IEEE Security Privacy*, 14(5):88–92, Sep. 2016.

[11] Guillaume Bonfante, José M. Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions. In *Conference on Computer and Communications Security*, 2015.

[12] Ella Bounimova, Patrice Godefroid, and David A. Molnar. Billions and billions of constraints: whitebox fuzz testing in production. In *35th International Conference*

Protections	Strength	Cost	Correctness	Stealth	Implementation available	Mitigation known
Complex Constraints						
MBA [46]	░	░ ?	✓	×	✓	[25]
Crypto Hash Functions [36]	▓ ɫ	?	✓	×	×	
Path Divergence						
Self-modification	░	▓	✓	×	✓	[11, 45]
Symbolic Code [44]	░	▓	✓	░	×	[44]
Covert Channel [39]	▓ ɫ	▓	×	▓	✓	
Path Explosion						
Linear Obfuscation [43]	░	░ ɫ	✓	░ ɫ	×	
Path-oriented Protections [4, 31]	▓ ɫɫ	▓ ɫɫ	✓	░	×	

×	Bad / No	?	unknown
░	Medium	ɫ	some experimental evaluation
▓	Good	ɫɫ	large experimental evaluation

Table 1: Comparative properties of anti-DSE obfuscations

Protection	#programs (size) (synthetic)	#programs (size) (realistic)	Available benchmarks	Reuse benchmarks	Attack tools
Hash [36] [5]	0	7 (NA) 11 (20 loc)	No Yes[1]	No No	- KLEE [14], Angr [37]
Covert Channel [39]	1 (10 loc)	0	No	No	S2E [17], Triton [33] Angr, Fuzzball [3]
Linear Obfuscation [43]	0	3 (30KBytes)	No	No	-
Path-Oriented Protections [31]	46 (25 loc)	7 (900 loc)	Yes[2]	Yes [4, 32]	KLEE, Angr Binsec [23], Triton
DSE vs. Standard Protections [4]	48 (25 loc) 5000 (100 loc)	0	Yes[3]	No	KLEE, Angr Triton

[1] http://www.partow.net/programming/hashfunctions/
[2] https://github.com/binsec/hade
[3] https://github.com/tum-i22/obfuscation-benchmarks

Table 2: Summary of experimental evaluations

on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013. IEEE Computer Society, 2013.
[13] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Xiaodong Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In Wenke Lee, Cliff Wang, and David Dagon, editors, *Botnet Detection: Countering the Largest Security Threat*, volume 36 of *Advances in Information Security*, pages 65–88. Springer, 2008.
[14] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2008.
[15] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2), 2013.
[16] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Symposium on Security and Privacy, SP*, 2012.
[17] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, 2012.
[18] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.
[19] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations, 1997.
[20] Christian S. Collberg, Clark D. Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In , *Symposium on Principles of Programming Languages, POPL*, 1998.
[21] Kevin Coogan, Gen Lu, and Saumya K. Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Conference on Computer and Communications Security, CCS*, 2011.
[22] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specification of concretization and symbolization policies in symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2016*, 2016.
[23] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER*, 2016.
[24] Saumya K. Debray and Jay Patel. Reverse engineering self-modifying code: Unpacker extraction. In *Working Conference on Reverse Engineering, WCRE*, 2010.
[25] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating mba-based obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection, SPRO@CCS 2016*, 2016.
[26] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3), 2012.
[27] Nguyen Minh Hai, Mizuhito Ogawa, and Quan Thanh Tho. Obfuscation code localization based on CFG generation of malware. In *Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers*. Springer, 2015.
[28] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: a hidden code extractor for packed executables. In *ACM Workshop Recurring Malcode (WORM)*. ACM, 2007.
[29] Johannes Kinder. Towards static analysis of virtualization-obfuscated binaries. In *19th Working Conference on Reverse Engineering, WCRE*, 2012.

[30] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *23rd Annual Computer Security Applications Conference (ACSAC 2007), December 10-14, 2007, Miami Beach, Florida, USA*. IEEE Computer Society, 2007.

[31] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to kill symbolic deobfuscation for free (or: Unleashing the potential of path-oriented protections). In *Annual Conference on Computer Security Applications, ACSAC 2019*. ACM, 2019.

[32] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic deobfuscation: from virtualized code back to the original. In *5th Conference on Detection of Intrusions and malware & Vulnerability Assessment (DIMVA)*, 2018.

[33] Florent Saudel and Jonathan Salwan. Triton : Framework d'exécution concolique. In *SSTIC*, 2015.

[34] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1), 2016.

[35] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Symposium on Security and Privacy, S&P*, 2010.

[36] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *Network and Distributed System Security Symposium, NDSS*, 2008.

[37] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy, SP*, 2016.

[38] Venkatesh Srinivasan and Thomas W. Reps. An improved algorithm for slicing machine code. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016*. ACM.

[39] Jon Stephens, Babak n, Christian S. Collberg, Saumya Debray, and Carlos Scheidegger. Probabilistic obfuscation through covert channels. In *European Symposium on Security and Privacy, EuroS&P*, 2018.

[40] Ramtine Tofighi-Shirazi, Maria Christofi, Philippe Elbaz-Vincent, and Thanh Ha Le. Dose: Deobfuscation based on semantic equivalence. In *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop, San Juan, PR, USA, December 3-4, 2018*. ACM, 2018.

[41] Julien Vanegue and Sean Heelan. SMT solvers in software security. In *6th USENIX Workshop on Offensive Technologies, WOOT'12*, 2012.

[42] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, Charlottesville, VA, USA, 2000.

[43] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear obfuscation to combat symbolic execution. In *European Symposium on Research in Computer Security, ESORICS*, 2011.

[44] Babak Yadegari and Saumya Debray. Symbolic execution of obfuscated code. In *Conference on Computer and Communications Security (CCS)*, 2015.

[45] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *Symposium on Security and Privacy, SP*, 2015.

[46] Yongxin Zhou, Alec Main, Yuan Xiang Gu, and Harold Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In *Information Security Applications, WISA*, 2007.