How to Kill Symbolic Deobfuscation for Free

(or: Unleashing the Potential of Path-Oriented Protections)

Anonymous Author(s)

ABSTRACT

Code obfuscation is a major tool for protecting software intellectual property from attacks such as reverse engineering or code tampering. Yet, recently proposed (automated) attacks based on Dynamic Symbolic Execution (DSE) shows very promising results, hence threatening software integrity. Current defenses are not fully satisfactory, being either not efficient against symbolic reasoning, or affecting runtime performance too much, or being too easy to spot. We present and study a new class of anti-DSE protections coined as path-oriented protections targeting the weakest spot of DSE, namely path exploration. We propose a lightweight, efficient, resistant and analytically proved class of obfuscation algorithms designed to hinder DSE-based attacks. Extensive evaluation demonstrates that these approaches critically counter symbolic deobfuscation while yielding only a very slight overhead.

ACM Reference Format:
Anonymous Author(s). 2019. How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-Oriented Protections) . In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Context. Reverse engineering and code tampering are widely used to extract proprietary assets (e.g., algorithms or cryptographic keys) or bypass security checks from software. *Code protection* techniques precisely seek to prevent, or at least make difficult, such *man-at-the-end* attacks, where the attacker has total control of the environment running the software under attack. Obfuscation [19, 20] aims at hiding a program's behavior by transforming its executable code in such a way that the behavior is conserved but the program becomes much harder to understand. Even though obfuscation techniques are quite resilient against basic automatic reverse engineering (including static attacks, e.g. disassembly, and dynamic attacks, e.g. monitoring), code analysis improves quickly [37]. Recent attacks based on *Dynamic Symbolic Execution* (DSE, a.k.a. *concolic execution*) [17, 28, 38] use logical formulas to represent input constraints along an execution path, and then automatically solve these constraints to discover new execution paths. DSE appears to be very efficient against existing obfuscations [5, 8, 22, 35, 49], combining the best of dynamic and semantic analysis.

Problem. *The current state of symbolic deobfuscation is actually pretty unclear.* Dedicated protections have been proposed, mainly based on hard-to-solve predicates, like Mixed Boolean Arithmetic formulas (MBA) [50] or cryptographic hash functions [40]. Yet the effect of complexified constraints on automatic solvers is hard to predict [6], while cryptographic hash functions are easy to spot, may induce significant overhead and are amenable to key extraction attacks (possibly by DSE).

On the other hand, DSE has been fruitfully applied on malware and legit codes protected by state-of-the-art tools and methods, including virtualization, self-modification, hashing or MBA [8, 35, 49]. The recent systematic experimental evaluation of symbolic deobfuscation by Banescu et al. [5] shows that most standard obfuscation techniques do not seriously impact DSE. Only nested virtualization seems to provide a good protection, assuming the defender is ready to pay a high cost in terms of runtime and code size [35]. Also, while the experimental evaluation of Banescu et al. sets the stage for a systematic study of deobfuscation methods, the study consider only obfuscation strength − while cost is also crucial − and small synthesized functions.

Goals and Challenges. We want to propose a new class of dedicated anti-DSE obfuscation techniques to render automated attacks based on symbolic execution inefficient. These techniques should be *strong* – making DSE intractable in practice, *lightweight* – with very low overhead in both code size and runtime performance.

While most anti-DSE defenses try to break the symbolic reasoning part of DSE (constraint solver), we instead target its real weak spot, namely path exploration. Banescu et al. [5] present one such specific obfuscation scheme but with a large space overhead and no experimental evaluation. We aim at proposing a general framework to understand such obfuscations and to define new schemes *both strong and lightweight.*

Contribution. We study *path-oriented* protections, a class of protections seeking to hinder DSE by substantially increasing the number of feasible paths within a program.

- We detail a formal framework describing *path-oriented* protections (Sec. 4). We characterize their desirable properties — namely *tractability*, *strength*, and the key criterion of *single value path* (SVP). The framework is *predictive*, in the sense that our classification is confirmed by experimental evaluation (Sec. 8), allowing both to shed new light on the few existing path-oriented protections and to provide guidelines to design new ones. In particular, no existing protection [5] achieves both tractability and optimal strength (SVP). As a remedy, we propose *the first two obfuscation schemes* achieving both *tractability and optimal strength* (Sec. 5).
- We highlight the importance of the *anchorage policy*, i.e. the way to choose where to insert protection in the code, in terms of protection efficiency and robustness. Especially, we identify a way of achieving *optimal composition* of path-oriented protections

---

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

(Sec. 6.1), and also a way to completely prevent taint-based, pattern-matching-based and slicing-based attacks (two powerful code-level attacks against obfuscation), coined as *resistance by design* (Sec. 6.2).

- We conduct extensive experiments (Sec. 8.3) with two different attack scenarios — *exhaustive path coverage* (Sec. 8.3) and *secret finding*. Results confirm that path-oriented protections are much stronger against DSE attacks than standard code protections (including nested virtualization) for only a slight overhead. Moreover, while existing (non optimal) techniques [5] can still be weak in some scenarios (very long timeout or secret finding), our *new optimal schemes cripple symbolic deobfuscation at essentially no cost in any setting*. Finally, experiments against slice, pattern-matching and taint attacks confirm the quality of our robust-by-design mechanism.

As a practical outcome, we propose a new *hardened deobfuscation benchmark* (Sec. 9), currently out-of-reach of symbolic engines, in order to extend existing obfuscation challenges [1] and benchmarks [5, 35].

Discussion. We study a powerful class of protections against symbolic deobfuscation, based on a careful analysis of DSE – we target its weakest point (path exploration) when other dedicated methods usually aim at its strongest point (constraint solving and ever-evolving SMT solvers). We propose a predictive framework allowing to understand these protections, as well as several concrete protections impacting DSE more than three levels of virtualization at essentially no cost. We expect them to be also efficient against other semantic attacks [10, 29] (cf. Sec. 10). From a methodological point of view, this work extends recent attempts at rigorous evaluation of obfuscation methods. We provide both an analytical evaluation, as Bruni et al. [14] for anti-abstract model checking, and a refinement of the experimental setup initiated by Banescu et al. [5].

2 MOTIVATION

2.1 Attacker model

Goal. We consider man-at-the-end scenarios where the attacker has full access to a potentially protected code under attack. The attacker has just the binary code and no access to the source code. The attacker model and the methodology follows closely the survey by Schrittwieser et al. [37]. In order to be more concrete, we will focus on the following (intermediate) goals: (1) *Exhaustive Path Exploration.* Covering every feasible path of the binary allows the attacker to retrieve a consistent Control Flow Graph and understand what the original program performs. (2) *Secret Finding.* Focusing on a specific part of the code (*e.g.* license checks) and try to understand or retrieve a secret (*e.g.* a key).

Capacity. we assume an *all-powerful symbolic adversary*, that is, this adversary can run a correct and *complete* Dynamic Symbolic Execution (DSE). In practice, symbolic engines are correct — every path discovered is actually feasible — but incomplete — they can be tricked into missing feasible paths [48] or the underlying solver may timeout. This adversary may also perform additional code-analysis based attack steps, such as *slicing*- [42], pattern-matching or *tainting*-based [38] code simplifications.

Caveat. In the remains, we always consider this attacker model. There is one caveat that it is worth mentioning now. In this model, the attacker has no access to the source code. *However, part of our experimental evaluations are done from source codes. That is, in our experiments the attacker has sometimes the source code. The reason is that state-of-the-art source-level DSE tools are much more efficient than binary-level ones, and that there is no good state-of-the-art tools to perform slice or taint attacks on binary codes.* ***Our experimental conditions are much more in favor of the attacker****, and as a result they show that our approach is all the more effective.*

```
int check_char_0(char chr){
  char ch = chr;
  ch ^= 97;
  return (ch == 31);
}

/* ... 9 other checks ... */

int check_char_10(char chr){ /* ... */ }

int check(char* buf) {
  int retval = 1;
  retval *= check_char_0(buf[0]);
  /* ... check buf[1] to buf[9] ... */
  retval *= check_char_10(buf[10]);
  return retval;
}

int main(int argc, char** argv) {
  char* buf = argv[1];
  if (check(buf)) puts("win");
  else puts("lose");
}
```

Figure 1: Manticore crackme code structure

2.2 Motivating example

Let us illustrate anti-symbolic path-oriented protections on a toy crackme program[1]. Fig. 1 displays a skeleton of its source code. `main` calls `check` to verify each character of the 11 bytes input. It then outputs `"win"` for a correct guess, `"lose"` otherwise. Each sub-function `check_char_i`$_{i \in [0,10]}$ hides a secret character value behind bitwise transformations, like xor or shift. *Such a challenge can be easily solved, completely automatically, by symbolic execution tools. KLEE [16] needs 0.03s (on C code) and Binsec [24] 0.3s (on binary code) to both find a winning input and explore all paths.*

Standard protections. Let us now protect the program with standard obfuscations to measure their impact on symbolic deobfuscation. We will rely on Tigress [21], a widely used tool for systematic evaluation of deobfuscation methods [5, 8, 35], to apply (nested) virtualization, a most effective obfuscation [5]. Yet, Table 1 clearly shows that virtualization does not prevent KLEE from finding the winning output, though it can thwart path exploration – but with a high runtime overhead (40×).

The case for (new) path-oriented protections. To defend against symbolic attackers, we thus need better anti-DSE obfuscation: *path-oriented protections*. Such protections aim at exponentially increasing the number of paths that a DSE-based deobfuscation tool, like KLEE, must explore. Two such protections are Split and For, illustrated in Fig. 2 on function `check_char_0` of the example. *For the*

---

[1] https://github.com/trailofbits/manticore

Table 1: DSE Attack on the Crackme Example (KLEE)

	Obfuscation type		**Slowdown** *Symbolic Execution*		**Over-head**
			Coverage	Secret	runtime
Standard	Virt		××	××	×1.1
	Virt ×2		×	××	×1.3
	Virt ×3		✓	×	×40
Path-Oriented	SPLIT [5]	$k = 11$	××	××	×1.0
		$k = 15$	✓	××	×1.0
		$k = 19$	✓	××	×1.0
	FOR (new)	$k = 1$	✓	×	×1.0
		$k = 2$	✓	✓	×1.0
		$k = 3$	✓	✓	×1.0

×× t ≤ 1s   × 30s < t < 5min   ✓ time out (≥ 1h30)

Unobfuscated case: KLEE succeeds in 0.03s

For

```
int func(char chr){


  char ch = 0;
  for (int i=0; i<chr; i++)
     ch++;
  ch ^= 97;
  return (ch == 31);
}
```

Split

```
int func(char chr,ch1,ch2) {
// new input char ch1 and ch2
  char garb = 0 // junk
  char ch = chr;
  if (ch1 > 60) garb++; else garb−−;
  if (ch2 > 20) garb++; else garb−−;
  ch ^= 97;
  return (ch == 31);
}
```

Figure 2: Unoptimized obfuscation of check_char_0

sake of simplicity, the protections are implemented in a naive form, sensitive to slicing or compiler optimizations. Robustness is discussed afterwards. In a nutshell, Split— an instance of Range Divider [5] — adds a number k of conditional statements depending on new fresh inputs, increasing the number of paths to explore by a factor of 2^k. Also, in *this implementation* we use a junk variable garb and two additional inputs ch1 and ch2 unrelated to the original code. The *novel* obfuscation For (Sec. 5) adds k loops whose upper bound depends on distinct input bytes and which recompute a value that will be used later, expanding the number of paths to explore by a factor of $2^{8 \cdot k}$ – assuming a 8-bit **char** type. *This implementation* does not introduce any junk variable nor additional input. In both cases, the obfuscated code relies on the input, forcing DSE to explore *a priori* all paths. Table 1 summarizes the performance of Split and For. Both Split and For do not induce any overhead, Split is highly efficient (timeout) against coverage but not against secret finding, while For is highly efficient for both. For ($k = 2$) performs already better than Split ($k = 19$) and further experiments (Sec. 8) shows For to be a much more effective path protection than Split.

Question: *How to distinguish a priori between mildly effective and very strong path-oriented protections?*

Note that gcc -Ofast is able to remove this simple Split, as it is not related to the output *(slicing attack)*. The basic For resists such attack, but clang -Ofast is able to remove it by an analysis akin to a *pattern attack*. However, a sightly modified For (Fig. 3) overcomes such optimizations.

Question: *How to protect path-oriented protections against code analysis-based attacks (slicing, tainting, patterns)?*

```
int func(char chr) {
  int ch = 0; // prevent char overflows
  for (int i=0; i< (int) chr; i++){
     if (i % 2 == 0) ch += 3;
     if (i % 2 != 0) ch−−;
  }
  if (i % 2 != 0) ch -= 2; // adjust for odd values
  ch ^= 97;
  return (ch == 31);
}
```

Figure 3: Enhanced For – check_char_0

The goal of this paper is to define, analyze and explore in a systematic way the potential of path-oriented transformations as anti-DSE protections. We define a *predictive* framework (Sec. 4) and propose several new *concrete protections* (Sec. 5). In particular, our framework allows to precisely explain why For is experimentally better than Split. We also discuss how path-oriented protections can be made resistant to several types of attacks (Sec. 6 and 7).

3 BACKGROUND

Obfuscation. Obfuscation [20] 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 $\mathcal{P}'$ and $\mathcal{P}$ are semantically equivalent, $\mathcal{P}'$ is roughly as efficient as $\mathcal{P}$ and $\mathcal{P}'$ is *harder to understand.* While it is still unknown whether applicable theoretical criteria of obfuscation exist [7], practical obfuscation techniques and tools do.

Let us touch briefly on three such important techniques. *Mixed Boolean-Arithmetic* [27, 50] transforms an arithmetic and/or Boolean equation into another using a combination of Boolean and arithmetic operands with the goal to be more complex to understand and more difficult to solve by SMT solvers [9, 44]. *Virtualization* and *Flattening* [45] 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 original program semantic. 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. *Self-modifying code* and *Packing* insert instructions that dynamically modify the flow of executed instructions. These techniques seriously damage static analyses by hiding the real instructions. However, extracting the hidden code can be done by dynamic approaches [26, 30], including DSE [49].

Dynamic Symbolic Execution (DSE). Symbolic execution [17] 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. The symbolic execution engine 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 the constraints automatically (typically with off-the-shelf SMT solvers [44]) then allows to generate *new input values leading to new paths*, progressively covering all paths of the program – up to a user-defined bound. The technique has seen strong renewed interest in the last decade to become a prominent bug finding technique [17, 18, 28].

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* [28]. Typically, (concrete) runtime values are used to complete missing part of path constraints that are then fed to the solver through *concretization* [23]. This feature makes the approach especially robust against complicated constructs found in obfuscated binary codes, typically packing or self-modification, making DSE a strong candidate for automated deobfuscation – *symbolic deobfuscation*: it is as robust as dynamic analysis, with the additional ability to infer *trigger-based conditions*.

4 A FRAMEWORK FOR PATH-ORIENTED PROTECTIONS

This section presents a framework to evaluate *path-oriented* obfuscations, i.e. protections aiming at hindering symbolic deobfuscation by taking advantage of path explosion.

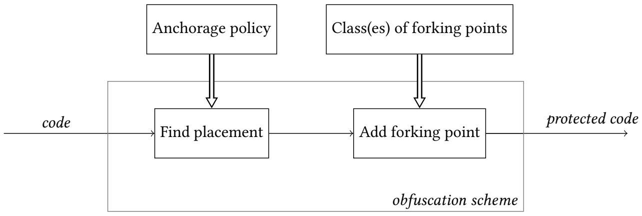

Figure 4: Path-Oriented Obfuscation Framework

4.1 Basic definitions

This paper deals with a specific kind of protections targeting DSE: *path-oriented* protections. Transforming a program $\mathcal{P}$ into $\mathcal{P}'$ using *path-oriented* protections ensures that $\mathcal{P}'$ is functionally equivalent to $\mathcal{P}$ and aims to guarantee $\#\Pi' \gg \#\Pi$, that is the number of paths in $\mathcal{P}'$ is much greater that the ones in $\mathcal{P}$.

The most basic path-oriented protection consists in one *forking point* inserted in the original code of $\mathcal{P}$.

Definition 1 (Forking Point) *A forking point* $\mathcal{F}$ *is a location in the code that creates at most* γ *new paths.* $\mathcal{F}$ *is defined by: an address a, a variable x and a capacity* γ*. It is written* $\mathcal{F}(a, x, \gamma)$

To illustrate this definition, see the snippet of Split in Figure 2: both if-statements define each a forking point of capacity 2 based on the variable `ch1` and `ch2` respectively.

Now, to obtain a complete path-oriented obfuscation $\mathcal{P}'$ of a program $\mathcal{P}$, we need to insert *n* forking points throughout the code of $\mathcal{P}$, hence the notion of *obfuscation scheme* (Fig. 4).

Definition 2 (Obfuscation scheme) *A* (path-oriented protection) obfuscation scheme *is a function* $f(\mathcal{P}, n)$ *that, for every program* $\mathcal{P}$*, inserts n forking points in* $\mathcal{P}$*. It comprises a set of forking points and an anchorage policy, i.e. the placement method of the forking points.*

4.2 Desirable obfuscation scheme properties

An ideal obfuscation scheme is both strong (high cost for the attacker) and cheap (low cost for the defender). Let us define these properties more precisely.

The **strength** of an obfuscation scheme is intuitively the expected increase of the search space for the attacker. Given an obfuscation scheme f, it is defined as $\Gamma_f(\mathcal{P}, n) = \#\Pi_{f(\mathcal{P},n)}$, for a program $\mathcal{P}$ and n forking points to insert.

The **cost** is intuitively the *maximal* runtime overhead the defender should worry about. Given an obfuscation scheme f, the **cost** is defined by $\Omega_f(\mathcal{P}, n)$, as the *maximum* trace size of the obfuscated program $f(\mathcal{P}, n)$. Formally, $\Omega_f(\mathcal{P}, n) = max_i\{|\tau_i'|\}$ where $\{\tau_i'\}$ is the set of execution traces of $f(\mathcal{P}, n)$ and $|\tau_i'|$ is the size of the trace.

We seek strong tractable obfuscations, i.e., yielding enough added paths to get a substantial slowdown, with a low runtime overhead.

Definition 3 (Strong scheme) *An obfuscation scheme f is* **strong** *if for any program* $\mathcal{P}$*, we have* $\Gamma_f(\mathcal{P}, n) \geq 2^{O(n)} \cdot \#\Pi_{\mathcal{P}}$*, where* $\Pi_{\mathcal{P}}$ *is the set of paths of mathcalP. Putting things quickly, it means that the number of paths to explore is multiplied by* 2^n*.*

Definition 4 (Tractable scheme) *An obfuscation scheme f is* **tractable** *if for any program* $\mathcal{P}$*,* $\Omega_f(\mathcal{P}, n) \leq max_i\{|\tau_i|\} + O(n)$*, where* $max_i\{|\tau_i|\}$ *is the size of the longest trace of* $\mathcal{P}$*. In other words, it is tractable only if the overhead runtime is linear on n.*

Combining schemes. Scheme composition preserves tractability (the definition involves an upper bound) but not necessarily strength (the definition involves a lower bound). Hence, we need *optimal composition rules* (Sec. 6.1).

4.3 Building stronger schemes

Strong path-oriented protections, can rely on composition but we saw in Sec. 4.2 that it is not straightforward. But since path-oriented protections first lean on forking execution into many paths, we should also investigate whether some forking points are better that others. The best case is to insert k forking points $(\mathcal{F}(a_i, x_i, \gamma_i))_i$ such that it would ensure that each path created by a forking point $(\mathcal{F}(a_i, x_i, \gamma_i))_i$ corresponds to only one possible value of the variable x_i. This leads us to define this type of forking point as a *Single Value Path* (SVP) protection.

Definition 5 (Single Value Path) *A forking point based on variable x is* Single Value Path *(SVP) if and only if x has only one possible value in each path created by the protection.*

A SVP forking point creates a new path for each possible value of variable x (*e.g.*, i.e., 2^{32} new paths are created for an unconstrained C **int** variable). For example, the For obfuscation shown in Figure 2 produces 2^{32} paths, that should be a priori explored since it depends on an input variable.

SVP forking points is key to ensure that DSE will need to enumerate all possible input values of the program under analysis (thus *boiling down to brute forcing*) – see Sec. 6.

5 CONCRETE PATH-ORIENTED PROTECTIONS

This section reviews existing path-oriented protection schemes within the framework of Sec. 4, but also details new such schemes achieving both tractability and optimal strength (SVP).

Range Divider [5]. Range Divider is an anti-symbolic execution obfuscation proposed by Banescu et al.. Branch conditions are inserted in basic blocks to divide the input value range into multiple

```
int main (int argc, char** argv){
  char* input = argv[1];

  char chr = *input; // inserted by obfuscation

  switch (chr) {  // inserted by obfuscation
    case 1:  ...  // original code
      break;
    case 2:  // obfuscated version of case 1
      break;
    ...
    default: // another obfuscated version of case 1
      break;
  }
  return (*input >= 100);
}
```

Figure 5: Range Divider obfuscation

sets. The code inside each branch of the conditional statement is an obfuscated version of the original code. We distinguish two cases, depending on whether the branch condition uses a **switch** or a **if** statement. *In the remaining part of this paper, Split will denote the Range Divider obfuscation with **if** statement, and Range Divider the Range Divider obfuscation with **switch** statement.*

The Range Divider (**switch**) scheme introduces an exhaustive **switch** statement over all possible values of a given variable – see example in Fig. 5, thus yielding 2^S extra-paths, with S the bit size of the variable. This scheme enjoys the SVP property as in each branch of the **switch** the target variable can have only one value, and it is also tractable in time provided the **switch** is efficiently compiled into a binary search tree or a jump table, as usual. Yet, while not pointed out by Banescu et al., *this scheme is not tractable in space* (code size) as it leads to *huge* amount of code duplication – the byte case may be fine, but not above.

```
int main (int argc, char** argv){
  char* input = argv[1];

  char chr = *input; // inserted by obfuscation

  if (chr < 30) {  // inserted by obfuscation
      ...          // original code
   }
  else  ...        // obfuscated version of case true

  return (*input >= 100);
}
```

Figure 6: Split obfuscation

Split [5]. This transformation (Fig. 6) is similar to Range Divider, but the control-flow is split by a condition triggered by a variable. This protection is tractable in both time (only one additional condition check per forking point) and space (only one block duplication per forking point). Yet, the protection is not SVP.

For (new). The For scheme (Fig. 2) replaces assignments `ch := chr` by loops `ch = 0; for (i = 0; i <= chr; i++) ch++;` where `chr` is an input-dependent variable. Intuitively, such **for** loops can be unrolled n times, for any value n that `chr` can take at runtime. Hence, a loop controlled by a variable defined over a bit size S generates up to 2^S extra-paths, with additional path length of 2^S. While the achieved protection is excellent, it is *intractable* when $S = 32$ or $S = 64$. **The trick** is to restrict this scheme to byte-size variables, and then chain such forking points on each byte of the variable of interest. Indeed, For over a byte-size variable generates up to 2^8 additional paths with an additional path length at most of 2^8. Chaining k forking points such For loops leads up to 2^{8k} extra-paths with an extra-length of only $k \cdot 2^8$, *keeping strength while making runtime overhead reasonable.* (More precisely with a constant time overhead wrt inputs.)

Write (new). The Write obfuscation adds self-modification operations to the code. It replaces an assignment `a := input` with a non input-dependent operation `a := k` (with `k` an arbitrary constant value) and replaces at runtime this instruction by `a := i` where `i` is the runtime value of `input` (self-modification). This is illustrated in Fig. 7, where the offset move at label `L1` actually rewrites the constant `0` at `L2` to the value contained at the address of the input.

```
 L: mov [a], [input]  ⇒  L1: mov L2+off, [input]
                         L2: mov [a], 0
```

Figure 7: Write obfuscation

Symbolic execution engines are not likely to relate `a` and `input`, thus thinking that `a` is constant across all executions. If the dynamic part of the engine spots that `a` may have different values, it will iterate over every possible values of `input`, creating new paths each time. The scheme is SVP, and its overhead is negligible (2 additional instructions, independent of the bit size of the targeted variable as long as it can be handled natively by the underlying architecture). Write has yet two drawbacks: it can be spotted by a dynamic analysis and needs the section containing the code to be writable.

Table 2: Classification of obfuscation schemes

		Tractable		SVP	Stealth
		Time	Space		
Range Divider [5]	switch	✓	✗	✓	✗
Split [5]	if	✓	✓	✗	✓
For	word	✗	✓	✓	✓
	byte	✓	✓	✓	✓
Write		✓	✓	✓	~

Summary. The properties of every scheme presented so far are summarized in Table 2 – stealth is discussed in Sec. 7.2. *Obfuscation schemes from the literature are not fully satisfactory*: Range Divider is space expensive and easy to spot, Split is not strong enough (not SVP). On the other hand, *the new schemes* For (at byte-level) and Write *are both strong (they satisfy SVP) and tractable, making them perfect anti-DSE protections.*

As a consequence, we suggest using variations of For as the main protection layer, with Write deployed only when self-modification and unpacking are already used (so that the scheme remains hidden). Range Divider can be used *occasionally* but only on byte variables to mitigate space explosion. Split can add further diversity and code complexity, but it should not be the main defense line. All these protections must be inserted in a *resistant-by-design* manner (Sec. 6.2) together with *diversity of implementation* (Sec. 7.1).

6 ANCHORAGE POLICY

We need to ensure that inserting path-oriented protections into a program gives real protection against DSE and will not be circumvented by attackers.

6.1 Optimal composition

We show how to combine the forking points in order to obtain *strong* obfuscation schemes. The issue with obfuscation scheme combination is that some forking points could hinder the efficiency of other forking points – imagine a `if`$(x \geq 100)$ split followed by a `if`$(x \leq 10)$ split: we will end up with 3 *feasible* paths rather than the expected $2 \times 2 = 4$, as one of the path if infeasible ($x > 100 \wedge x \leq 10$).

Intuitively, we would like the forking points to be independent from each other, in the sense that their efficiency combine perfectly (the number of extra-paths is $\gamma_1 \times \gamma_2$, with γ_i the capacities of two such independent forking points). An easy way to obtain forking point independence is to consider forking points built on independent variables – variables are *independent* if their values are not computed from the same input values. Actually, if independent forking points are well placed in the program, path-oriented protections ensure an exponential increase in the number of paths.

Property 1 (Optimal Composition) *Suppose that $\mathcal{P}'$ is obtained by obfuscating the program $\mathcal{P}$. If every original path of $\mathcal{P}$ goes through at least k independent forking points of $\mathcal{P}'$ of capacity γ, then $\#\Pi_{\mathcal{P}'} \geq \#\Pi_{\mathcal{P}} \cdot \gamma^k$*

By choosing enough independent SVP forking points (one for each input variable), we can even ensure that DSE will have to enumerate over all possible input values of the program under analysis, hence performing as bad as mere *brute forcing*.

Implementation. Ensuring that each path will go through at least k forking points can be achieved by carefully selecting the points in the code where the forking points are inserted: a control flow graph analysis provides information about where and how many forking points are needed to cover all paths. The easiest way to impact all paths at once is to select points in the code that are not influenced by any conditional statement. Dataflow analysis can be used further in order to ensure that the selected variables do not share dependencies with the same input (independent variables).

6.2 Resistance-by-design to taint and slice

Taint analysis [38] and (backward) slicing [42] are two advanced code simplification methods built on the notion of *data flow relations* through a program. These data flow relations can be defined as *Definition-Use* (Def-Use) chains – as used in compilers. Data are *defined* when variables are assigned values or declared, and *used* in expressions. Taint (resp. Slice) uses Def-Use chains to replace input-independent by its constant effect (resp. remove code not impacting the output). If there exists a Def-Use chain linking data x to data y, we write: x ⤷ y. *Relevant* variables are defined as having both a Def-Use chain with an input and one with an output:

Definition 6 (Relevant Variable) *x is* ***relevant*** *if there exists at least two Def-Use chains such that input ⤷ x and x ⤷ output.*

The use of a relevant variable x to implement a forking point makes the protection resistant against attacks by both slice (x impacts output) and taint (x is input dependent).

Property 2 (Resistance by design) *A forking point $\mathcal{F}(x, a, \gamma)$ is slice and taint resistant iff variable x is relevant.*

Implementation. Relevant variables can be identified in at least two ways. First, one can modify standard compiler algorithms computing *possible* Def-Use chains in order to compute *real* Def-Use chains – technically, going from a *may analysis* to a *must analysis*. A more original solution observes at runtime a set of real Def-Use chains and deduces accordingly a set of relevant variables. This method does not require any advanced static analysis, only basic dynamic tracing features.

7 THREATS

In this section we discuss possible threats to path-oriented protections and propose adequate mitigations. Indeed, when weaving the forking points within the code of a program, we need to ensure that they are hard to discover or remove. Three main attacks appear to seem effective against path-oriented protections: (1) taint analysis, (2) backward slicing, (3) and pattern attacks. We showed how path-oriented protections can be made *resistant by-design* to Taint and Slice in Sec. 6.2. We discuss now pattern attacks, as well as stealth issues and the specific unfriendly case of programs with a small input space.

```
① for (int i = 0; i++; i < input) a++;
② for (int i = 0; i++; i < input)
     a = (a ^ 1) + 2 * (a & 1);
③ int i = 0;
   while (i < input) { i++; a++; }
④ int f(int x){ return (x <= 0 ? 0 : f(x − 1) + 1); }

   a = f(input);
⑤ #define A // arbitrary value

   int f(int x){ return x <= 0 ? 0 : A + g(x − 1); }

   int g(int x){ return !x ? 1 − A : 2 − A + f(−−x); }

   a = f(input);
```

Figure 8: Several encodings of protection For

7.1 Pattern attacks

Pattern attacks search for specific patterns in the code of a program to identify, and remove, known obfuscations. This kind of analysis assumes more or less similar constructions across all implementations of an obfuscation scheme. A general defense against pattern attacks is *diversity*. It works well in the case of path-oriented protections: on the one hand the schemes we provide can be implemented in many ways, and on the other hand our framework provides guidelines to design new schemes – altogether, it should be enough to defeat pattern attacks. Regarding diversity of implementations, the standard For forking point can be for example replaced by a **while** loop, (mutually) recursive function(s), the loop body can be masked through MBAs, etc. These variants can be combined as in Fig. 8, and we can imagine many other variations.

The other schemes as well can be implemented in many ways, and we could also think of ROP-based encoding [39] or other diversification techniques. Altogether, it should provide a powerful enough mitigation against pattern attacks.

7.2 Stealth

In general, code protections are better when hard to identify, in order to prevent human-level attacks like stubbing parts of the code or designing targeted methods. Let us evaluate the stealthiness of path-oriented protections (summary in Table 2). Split and For do not use rare operators or exotic control-flow structures, only some additional conditions and loops scattered through the program. Hence Split and For are considered hard to detect on binary code, though For especially may be visible at source level. Range Divider is easy to spot at source level: `switch` statements with hundreds of branches are indeed distinctive. Compilation makes it harder to find but the height of the produced binary search trees or the size of the generated jump table are easily perceptible. Write stands somewhere in between. It cannot be easily discovered statically, but is trivial to detect dynamically. However, since self-modification and unpacking are common in obfuscated codes, Write could well be mistaken for one of these more standard (and less damaging) protections.

7.3 Beware: programs with small input space

Resistance by design (Sec. 6.2) relies on *relevant variables*, so we only have limited room for forking points. In practice it should not be problematic as Sec. 8 shows that we already get very strong protection with only 3 input bytes – assuming a SVP scheme. Yet, for programs with very limited input space, we may need to add *(fake) crafted inputs* for the input space to become (apparently) larger – see Split example in Fig. 2. In this case, our technique still ensures resistance against tainting attacks, but slicing attacks may now succeed. The defender must then rely on well-known (but imperfect) anti-slicing protections to blur code analysis through hard-to-reason-about constructs, such as pointer aliasing, arithmetic and bit-level identities, etc.

8 EXPERIMENTAL EVALUATION

The experiments below seek to answer four Research Questions[2]:

- **RQ1** What is the impact of path-oriented protections on semantic attackers? Especially, we consider DSE attack and two different attacker goals: Path Exploration **(Goal 1)** and Secret Finding **(Goal 2)**.
- **RQ2** What is the cost of path-oriented protections for the defender in runtime overhead and code size increase?
- **RQ3** What is the effectiveness of our resistance-by-design mechanism against taint and slice attacks?
- **RQ4** What is the difference between standard protections, path-oriented protections and SVP protections?

8.1 Experimental setup

Tools. Our attacker mainly comprises the state-of-the-art source-level DSE tool KLEE (version 1.4.0.0 with LLVM 3.4, POSIX runtime and STP solver). KLEE is highly optimized [16] and works from source code, so *it is arguably the worst case DSE-attacker we can face* [5]. We have considered standard search heuristics (DFS, BFS, Non-Uniform Random Search) but report only about DFS as others perform slightly worse (*see Appendix*). Also, we used two other *binary-level* DSE tools, Binsec [24] and Triton [36], with similar results.

Regarding standard defense, we use Tigress [21], a freely available state-of-the-art obfuscator featuring many standard obfuscations and allowing to precisely control which ones are used – making Tigress a tool of choice for the systematic evaluation of deobfuscation methods [5, 8, 35].

Protections. We only consider tractable path-oriented protections and select both a new SVP scheme (For) and an existing non-SVP scheme (Split), inserted in a robust-by-design way. We vary the number of forking points per path (parameter k).

We also consider standard protections: *Virtualization* (up to 3 levels), *arithmetic encoding* and *flattening* [46]. Previous work [5] has shown that nested virtualization is the sole standard protection useful against DSE. Our results confirm that, so we report only results about virtualization *(other results partly in Appendix).*

8.2 Datasets

We select small and medium programs for experiments as they represent the worst case for program protection. If path-oriented protections can slow down DSE analysis substantially on smaller programs, then those protections can only give better results for larger programs.

Dataset #1. This synthetic dataset from Banescu *et al* [3] [5] offers a valuable diversity of functions and has already been used to assess resilience against DSE. It has 48 C programs (between 11 and 24 lines of code) including control-flow statements, integer arithmetic and system calls to `printf`. We exclude 2 programs because reaching full coverage took considerably longer than for the other 46 programs and blurred the overall results. Also, some programs have only a 1-byte input space, making them too easy to brute force (Goal 2). We turn them into equivalent 8-byte input programs with same number of paths – additional input are not used by the program, but latter protections will rely on them. The maximum time to obtain full coverage on the **46** programs with KLEE is 33s, mean time is 2.34s (Appendix A).

Dataset #2. The second dataset comprises 7 larger realistic programs, representative of real-life protection scenarios: 4 hash functions (City, Fast, Spooky, md5), 2 cryptographic encoding functions (AES, DES) and a snippet from the GRUB bootloader. Unobfuscated programs have between 101 and 934 LOCs: KLEE needs at most 33.31s to explore all paths, mean time is 8s (Appendix A).

8.3 Impact on Dynamic Symbolic Execution

Protocol. To assess the impact of protections against DSE, we consider the induced *slowdown* (time) of symbolic execution on an obfuscated program w.r.t. its original version. Fore more readable results, we only report whether DSE achieves its goal or times out.

For Path Exploration (**Goal 1**), we use programs from Datasets #1 and #2, add the protections and launch KLEE untils it reports full coverage or times out – 3h for Dataset #1, or a 5,400× average slowdown, 24h for Dataset #2, or a 10,000x average slowdown. For Secret finding (**Goal 2**), we modify the programs from both datasets into "secret finding" oriented code (e.g.,*win / lose*) and set up KLEE to stop execution as soon as the secret is found. We take the whole

---

[2] Download at https://bit.ly/2wYSEDG – deanonymation

[3] https://github.com/tum-i22/obfuscation-benchmarks

Dataset #2, but restrict Dataset #1 to the 15 programs with 16-byte input space. We set smaller timeouts (1h for Dataset #1, 3h and 8h for Dataset #2) as the time to find the secret with KLEE on the original programs is substantially lower (0.3*s* average).

Table 3: Impact of obfuscations on DSE

Transformation (#TO/#Samples)	**Dataset #1**		**Dataset #2**		
	Goal 1 *3h TO*	*Goal 2* *1h TO*	*Goal 1* *24h TO*	*Goal 2* *3h TO*	*Goal 2* *8h TO*
Virt	0/46	0/15	0/7	0/7	0/7
Virt ×2	1/46	0/15	0/7	0/7	0/7
Virt ×3	5/46	2/15	1/7	0/7	0/7
SPLIT ($k = 10$)	1/46	0/15	0/7	0/7	0/7
SPLIT ($k = 13$)	4/46	0/15	1/7	1/7	0/7
SPLIT ($k = 17$)	18/46	2/15	3/7	2/7	1/7
FOR ($k = 1$)	2/46	0/15	0/7	0/7	0/7
FOR ($k = 3$)	30/46	8/15	3/7	2/7	1/7
FOR ($k = 5$)	46/46	15/15	7/7	7/7	7/7

Results & Observations. Table 3 shows the number of timeouts during symbolic execution for each obfuscation and goal. For example, KLEE is always able to cover all paths on Dataset #1 against simple Virtualization (0/46 TO) – the protection is useless here, while it fails on ≈ 40% of the programs with Split ($k = 17$), and never succeeds with For ($k = 5$).

As expected, higher levels of protections (more virtualization layers or more forking points) result in better protection. Yet, results of Sec. 8.4 will show that while increasing forking points is cheap, increasing levels of virtualization is quickly prohibitive.

Virtualization is rather weak for both goals – only 3 levels of virtualization manage some protection. For performs very well for both goals: with $k = 3$ and Dataset #1, For induces a timeout for more than half the programs for both goals, which is significantly better than **Virt**×3. With $k = 5$, all programs timeout. In between, Split is efficient for Goal 1 (even though it requires much higher k than For) but not for Goal 2 – see for example Dataset #1 and $k = 17$: 39% timeouts (18/46) for Goal 1, only 13% (2/15) for Goal 2.

Other (unreported) results. All standard protections from Tigress we used turns out to be ineffective against DSE – for example *Flattening* and *EncodeArithmetic* on Dataset#1 slows path exploration by a maximum factor of 10, which is far from critical. Search heuristics obviously do not make any difference in the case of Path Exploration (Goal 1). Still, DFS tends to perform slightly better than BFS and NURS against Split in the case of Secret Finding (Goal 2). No other difference is visible. Experiments with two binary-level DSE engines supported by different SMT solvers (Binsec [24] with Boolector[13], and Triton [36] with Z3[25]) are in line with those reported here. Actually, as expected, both engines perform worse than KLEE. Part of these results can be found in Appendix.

Conclusion. As already stated in the literature, standard protections such as nested virtualization are mostly inefficient against DSE attacks. Path-oriented protections are shown here to offer a stronger protection. Yet, care must be taken. Non-SVP path protections such as Split do improve over nested virtualization (Split with $k = 13$ is roughly equivalent to **Virt** ×3, with $k = 17$ it is clearly superior), but they provide only a weak-to-mild protection in the cases of Secret Finding (Goal 2) or large time outs. On the other hand, SVP protections (represented here by For) are able to discard all DSE attacks on our benchmarks for both Path Exploration and Secret Finding with only $k = 5$, demonstrating a protection power against DSE far above those of standard protections and non-SVP path protections.

To conclude, path-oriented protections are indeed a tool of choice against DSE attacks (**RQ1**), much stronger than standard ones (**RQ4**). In addition, SVP allows to predict the strength difference of these protections (**RQ4**), against Coverage or Secret Finding.

8.4 Impact on Runtime Performance

Protocol. We evaluate the cost of path-oriented protections by measuring the *runtime overhead* (RO) and the (binary-level) *code size increase* (CI) of an obfuscated program w.r.t. its original version. We consider also two variants of For– its recursive encoding REC (Sec. 7.1) and the more robust P2 encoding (Sec. 8.5), as well as the untractable word-level For scheme (Sec. 5), coined WORD.

Table 4: Impact of obfuscations on runtime performance

Transformation	**Dataset #1**		**Dataset #2**	
	RO	*CI*	*RO*	*CI*
Virt	×1.5	×1.5	×1.7	×1.4
Virt ×2	×15	×2.5	×5.1	×2.1
Virt ×3	$\times 1.6 \cdot 10^3$	×4	×362	×3.0
SPLIT ($k = 10$)	×1.2	×1.0	×1.0	×1.0
SPLIT ($k = 13$)	×1.2	×1.0	×1.0	×1.0
SPLIT ($k = 50$)	×1.5	×1.5	×1.1	×1.0
FOR ($k = 1$)	×1.0	×1.0	×1.0	×1.0
FOR ($k = 3$)	×1.1	×1.0	×1.0	×1.0
FOR ($k = 5$)	×1.3	×1.0	×1.1	×1.0
FOR ($k = 50$)	×1.5	×1.5	×1.2	×1.1
FOR ($k = 5$) **P2**	×1.3	×1.0	×1.1	×1.0
FOR ($k = 5$) **REC**	×3.0	×1.0	×2.7	×1.0
FOR ($k = 1$) **WORD**	$\times 2.6 \cdot 10^3$	×1.0	$\times 2.1 \cdot 10^3$	×1.0

Results & Observations. Results are shown in Table 4 as average values over all programs in the datasets. As expected, nested virtualization introduces a significant and prohibitive runtime overhead (three layers: $\times 1.6 \cdot 10^3$ for Dataset #1 and ×362 for Dataset #2), and each new layer comes at a high price (from 1 to 2: between ×3 and ×10; from 2 to 3: between ×70 and ×100). Moreover, the code size is also increased, but in a more manageable way (still, at least ×3 for three layers). On the other hand, Split and For introduce only very low runtime overhead (at most ×1.3 on Dataset #1 and ×1.1 on Dataset #2), and no noticeable code size increase is reported even for $k = 50$. Regarding variants of For, P2 does not show any overhead w.r.t. For, while the recursive encoding REC comes at a higher price. Finally, *as predicted by our framework*, WORD is intractable.

Conclusion. As expected, tractable path-oriented protections indeed yield only a very slight overhead, both in terms of time or code size (**RQ2**), and improving the level of protection (k) is rather cheap, while nested virtualization comes at a high price (**RQ4**). Coupled with results of Sec. 8.3, it turns out that path-oriented protections offer a much better anti-DSE protection than nested virtualization at a runtime cost several orders of magnitude lower. Also, the code size increase due to path-oriented protections seems compatible with strict memory requirements (e.g., embedded systems) where it is not the case for nested virtualization.

Table 5: Robustness of path-oriented protections

Tool	**Robust ?**		
	P1 (basic)	P2 (obfuscated)	P3 (weak)
GCC -Ofast	✓	✓	×
clang -Ofast	×	✓	×
Frama-C Slice	✓	✓	×
Frama-C Taint	✓	✓	✓
Triton (taint)	✓	✓	✓
KLEE	✓	✓	✓

✓: no protection simplified ×: ≥ 1 protection simplified

8.5 Robustness to taint and slice attacks

Protocol. We consider the heavily optimized compilers Clang & GCC (many simplifications including slicing), the industrial-strength Frama-C static code analyzer (both its Taint and Slice plugins together with precise interprocedural range analysis) as well as Triton (which features tainting) and KLEE. We take 6 programs from dataset #1 (with 16-byte input space) and all programs from dataset #2. We use the For scheme (k=3) weaved into the code following our robust-by-design method (Sec. 6.2). Actually we consider 3 variants of the scheme: **P1**, **P2** and **P3**. P1 is the simple version of For presented in Fig. 8, P2 is a mildly obfuscated version (adds a `if` statement always evaluating to `true` in the loop – opaque predicate) and P3 relies on fake inputs (a dangerous construction discussed in Sec. 7.3). A protection will be said to be *simplified* when the number of explored paths for full coverage is much lower than expected (DSE tools), no protection code is marked by the analysis tool (Frama-C) or running KLEE on the produced code does not show any difference (compilers).

Results & Observations. Results in Table 5 confirm our expectations. No analyzer but clang is able to simplify our robust-by-design protections (P1 and P2), whereas the weaker P3 is broken by slicing (GCC, clang, Frama-C) but not by tainting – exactly as pointed out in Sec. 7.3. Interestingly, clang -Ofast simplifies scheme P1, *not due to slicing* (this is resistant by design), but thanks to some loop simplification more akin to a pattern attack, relying on finding an affine relation between variables and loop counters. The slightly obfuscated version P2 is immune to this particular attack.

Conclusion. It turns out that our robust-by-design method indeed works as expected against taint and slice (**RQ3**). Yet, care must be taken to avoid pattern-like simplifications. Note that in a real scenario, the attacker must work on binary code, making static analysis much more complicated. Also, virtualization, unpacking or self-modification can be used in addition to path-oriented protections to completely hinder static analysis.

9 APPLICATION: HARDENED BENCHMARK

We propose a set of benchmarks containing 4 programs from Banescu's dataset and our 6 real-world programs (GRUB excluded) from Sec. 8.2 in order to help advance the state of the art of symbolic deobfuscation. Each program comes with two setups, Path Exploration and Secret Finding, obfuscated with both a path-oriented protection (For k=5, taint- and slice- resistant) and a virtualization layer against human and static attacks[4]. Table 6 shows the performance of KLEE, Triton and Binsec (Secret Finding, 24h timeout). Unprotected and virtualized codes are easily solved, but hardened versions remain unsolved within the timeout, for every tool.

Table 6: Results on 10 hardened examples (secret finding)

	Unprotected (TO = 10 sec)	Virt ×1 (TO = 5 min)	**Hardened** – For (k=5) **(TO = 24h)**
KLEE	10/10	10/10	**0/10** ✓
Binsec	10/10	10/10	**0/10** ✓
Triton	10/10	10/10	**0/10** ✓

10 DISCUSSION

10.1 On the methodology

We discuss biases our experimental evaluation could suffer from.

Metrics. We add overhead metrics (runtime, code size) to the commonly used "DSE slowdown" measure [5, 35], giving a better account of the pros and cons of obfuscation methods.

Obfuscation techniques & tools. We consider the strongest standard obfuscation methods known against DSE, as identified in previous systematic studies [5, 35]. We restrict ourselves to their implementation in Tigress, a widely respected and freely available obfuscation tool considered state-of-the-art – studies including Tigress along packers and protected malware [8, 49] do not report serious deficiencies about its protections. Anyway, the evaluation of the path-oriented protections is independent of Tigress.

DSE engines. We use 3 symbolic execution engines (mostly KLEE, also Binsec and Triton) working on different program representations (C source, binary), with very similar final results. Moreover, KLEE is a highly respected tool, implementing advanced path pruning methods (*path merging*) and solving strategies. It also benefits from *code-level optimizations* of Clang as it operates on LLVM bitcode. Previous work [5] considers KLEE as the *worst-case attackers*, in front of Triton [36] and angr [41].

Benchmarks. Our benchmarks include Banescu et al.'s synthetic benchmarks [5], *enriched by 7 larger real-life programs* consisting essentially of hash functions (a typical software asset one has to protect) [35]. We also work both on source and binary code to add another level of variability. As already said, the considered programs are rather small, *on purpose*, to embody *the defender worst case*. Note that this case still represents real life situations, e.g., protecting small critical assets from *targeted* DSE attacks.

10.2 Generality of path-oriented protections

Path-oriented protections should be effective on a larger class of attacks besides DSE – actually, all major semantic program analysis techniques. Indeed, all path-unrolling methods will suffer from path explosion, including Bounded model checking [10], backward bounded DSE [8] and abstract interpretation with aggressive trace partitioning [31]. Model checking based on counter-example guided

---

[4] Sources available at https://bit.ly/2GNxNv9

refinement [29] will suffer both from path explosion and Single Value Path protections – yielding ineffective refinements in the vein of [14]. Finally, standard abstract interpretation [4] will suffer from significant precision loss due to the many introduced *merge points* – anyway purely static techniques cannot currently cope with self-modification or packing.

10.3 Countermeasures and mitigations

We can think of three possible mitigations a DSE attacker could use against our new defenses: (1) remove the protection through tainting and slicing; (2) detect our defenses via pattern attacks and (3) directly handle the protection through advanced semantic techniques for loops. *Slicing, tainting* and *pattern attacks*, are thoroughly discussed in Sec. 6.2 and 7.

Advanced program analysis techniques for loops is a very hot research topic, still largely open in the case of under-approximation methods such as DSE. The best methods for DSE are based on path merging [3], but they lack a generalization step allowing to completely capture loop semantics. Even though KLEE implements such path merging, it still fails against our protections. Widening in abstract interpretation [4] over-approximates loop semantics, but the result is often very crude: using such over-approximations inside DSE is still an open question. Anti-implicit flow techniques [32, 33] may identify dataflow hidden as control-flow (it identified for instance a For forking point), yet they do not recover any precise loop semantics and thus cannot reduce path explosion.

Finally, note that: (1) obfuscation schemes can easily be scattered along several functions (see alternative For encodings in Sec. 7.1) to bar expensive but targeted intra-procedural attacks – attackers will need (costly) precise inter-procedural methods, (2) real-life attacks are performed on binary code – binary-level static analysis is known to be extremely hard to get precise; and (3) static analysis is completely broken by packing or self-modification.

11 RELATED WORK

We have already discsussed obfuscation, symbolic execution and symbolic deobfuscation at length throughout the paper, including successful applications of DSE-related techniques to deobfuscation [8, 22, 35, 49]. In addition, Schrittwieser et al. [37] give an exhaustive survey about program analysis-based deobfuscation, while Schwartz et al. [38] review DSE, tainting and their applications in security.

Limits of symbolic execution. Anand et al. [2] describe, in the setting of automatic testing, the three major weaknesses of DSE: *Path explosion*, *Path divergence* and *Complex constraints*. Cadar [15] shows that compiler optimizations can sensibly alter the performance of a symbolic analyzer like KLEE, confirming the folklore knowledge that strong enough compiler optimizations resemble code obfuscations. That said, the performance penalty is far from offering a strong defense against symbolic deobfuscation.

Constraint-based anti-DSE protections. Most anti-DSE techniques target the constraint solving engine through hard-to-solve predicates. The impact on symbolic deobfuscation through the complexification of constraints has been studied by Banescu et al. [6]. Biondi et al. [11] propose an obfuscation based on *Mixed Boolean-Arithmetic* expressions [50] to complexify *points-to functions*, making it harder for solvers to determine the trigger. Eyrolles et al. [27] present a similar obfuscation together with a MBA expression simplifier based on pattern matching and arithmetic simplifications. Cryptographic hash functions hinder current solvers and can replace MBA [40]. In general, formula hardness is difficult to predict, and solving such formulas is a hot research topic. Though cryptographic functions resist solvers up to now, promising attempts [34] exist. More importantly, private keys must also be protected against symbolic attacks, yielding a potentially easier deobfuscation subgoal – a standard whitebox cryptography issue.

Other anti-DSE protections. Yadegari and Debray [48] describe obfuscations thwarting standard byte-level taint analysis, possibly resulting in missing legitimate paths for DSE engines using taint analysis (Triton does, KLEE and Binsec do not). It can be circumvented in the case of taint-based DSE by bit-level tainting [48]. *Symbolic Code* combines this idea with *input-dependent trigger-based self modifications*. Here, the dynamic analysis part of DSE must be able to detect these input-dependent self-modifications. Solutions exist but must be carefully integrated [8, 12]. Wang et al. [47] propose an obfuscation based on mathematical conjectures in the vein of the Collatz conjecture. This transformation increases the number of (symbolic) paths through an input-dependent loop, while the conjecture (should) ensure that the loop always converges to the same result. Banescu et al. [5] propose an anti-DSE technique based on encryption and proved to be highly effective, but it requires some form of secret sharing (the key) and thus falls outside the strict scope of MATE attacks that we consider here. Stephens et al. [43] recently proposed an obfuscation based on covert channels (timing, etc.) to hide data flow within invisible states. Current tools do not handle correctly this kind of protections. However, the method ensures only probabilistic correctness and thus cannot be applied in every context.

Systematic evaluation of anti-DSE techniques. Banescu et al. [5] set the ground for the experimental evaluation of symbolic deobfuscation techniques. Our own experimental evaluation extends and refines their method in several ways: new metrics, different DSE settings, larger examples. Bruni et al. [14] propose a mathematically proven obfuscation against Abstract Model Checking attacks.

12 CONCLUSION

Code obfuscation intends to protect proprietary software assets against attacks such as reverse engineering or code tampering. Yet, recently proposed (automated) attacks based on symbolic execution (DSE) and semantic reasoning have shown a great potential against traditional obfuscation methods. We explore a new class of anti-DSE techniques targeting the very weak spot of these approaches, namely path exploration. We propose a predictive framework for understanding such path-oriented protections, and we propose new lightweight, efficient and resistant obfuscations. Experimental evaluation indicates that our method critically damages symbolic deobfuscation while yielding only a very small overhead.

REFERENCES

[1] [n.d.]. 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. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013).

[3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2016. Enhancing symbolic execution with veritesting. *Commun. ACM* 59, 6 (2016).

[4] Gogul Balakrishnan and Thomas W. Reps. 2010. WYSINWYX: What you see is not what you eXecute. *ACM Trans. Program. Lang. Syst.* 32 (2010).

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

[6] Sebastian Banescu, Christian S. Collberg, and Alexander Pretschner. 2017. Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning. In *USENIX Security Symposium*.

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

[8] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *2017 IEEE Symposium on Security and Privacy, SP*. IEEE Computer Society.

[9] Clark Barrett and Cesare Tinelli. 2018. *Satisfiability Modulo Theories*. Springer International Publishing.

[10] Armin Biere. 2009. Bounded Model Checking. In *Handbook of Satisfiability*.

[11] Fabrizio Biondi, Sébastien Josse, Axel Legay, and Thomas Sirvent. 2017. Effectiveness of synthesis in concolic deobfuscation. *Computers & Security* 70 (2017).

[12] Guillaume Bonfante, José M. Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. 2015. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *Conference on Computer and Communications Security*.

[13] Robert Brummayer and Armin Biere. 2009. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS (Lecture Notes in Computer Science)*, Stefan Kowalewski and Anna Philippou (Eds.), Vol. 5505. Springer.

[14] Roberto Bruni, Roberto Giacobazzi, and Roberta Gori. 2018. Code Obfuscation Against Abstract Model Checking Attacks. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI*.

[15] Cristian Cadar. 2015. Targeted program transformations for symbolic execution. In *Joint Meeting on Foundations of Software Engineering, ESEC/FSE*.

[16] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association.

[17] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013).

[18] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Symposium on Security and Privacy, SP*. IEEE Computer Society.

[19] Christian Collberg and Jasvir Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection* (1st ed.). Addison-Wesley Professional.

[20] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. A Taxonomy of Obfuscating Transformations.

[21] Christian S. Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. 2012. Distributed application tamper detection via continuous software updates. In *Annual Computer Security Applications Conference, ACSAC*.

[22] Kevin Coogan, Gen Lu, and Saumya K. Debray. 2011. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Conference on Computer and Communications Security, CCS*.

[23] Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. 2016. Specification of concretization and symbolization policies in symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM.

[24] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER*. IEEE Computer Society.

[25] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.), Vol. 4963. Springer.

[26] Saumya K. Debray and Jay Patel. 2010. Reverse Engineering Self-Modifying Code: Unpacker Extraction. In *Working Conference on Reverse Engineering, WCRE*.

[27] Ninon Eyrolles, Louis Goubin, and Marion Videau. 2016. Defeating MBA-based Obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection, SPRO@CCS 2016*.

[28] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012).

[29] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. 2002. Lazy abstraction. In *The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[30] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: a hidden code extractor for packed executables. In *ACM Workshop Recurring Malcode (WORM)*. ACM.

[31] Johannes Kinder. 2012. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *19th Working Conference on Reverse Engineering, WCRE*.

[32] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. 2008. Implicit Flows: Can’t Live with ’Em, Can’t Live without ’Em. In *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*.

[33] Yin Liu and Ana Milanova. 2010. Static Information Flow Analysis with Handling of Implicit Flows and a Study on Effects of Implicit Flows vs Explicit Flows. In *14th European Conference on Software Maintenance and Reengineering, CSMR*.

[34] Saeed Nejati, Jia Hui Liang, Catherine H. Gebotys, Krzysztof Czarnecki, and Vijay Ganesh. 2017. Adaptive Restart and CEGAR-Based Solver for Inverting Cryptographic Hash Functions. In *Verified Software. Theories, Tools, and Experiments - 9th International Conference, VSTTE 2017*.

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

[36] Florent Saudel and Jonathan Salwan. 2015. Triton : Framework d’exÃľcution concolique. In *SSTIC*.

[37] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.* 49, 1, Article 4 (2016), 37 pages.

[38] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. 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*.

[39] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Conference on Computer and Communications Security, CCS*, Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson (Eds.). ACM.

[40] Monirul I. Sharif, Andrea Lanzi, Jonathon T. Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Network and Distributed System Security Symposium, NDSS*.

[41] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society.

[42] Venkatesh Srinivasan and Thomas W. Reps. [n.d.]. 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.

[43] Jon Stephens, Babak Yadegari, Christian S. Collberg, Saumya Debray, and Carlos Scheidegger. 2018. Probabilistic Obfuscation Through Covert Channels. In *European Symposium on Security and Privacy, EuroS&P*.

[44] Julien Vanegue and Sean Heelan. 2012. SMT Solvers in Software Security. In *6th USENIX Workshop on Offensive Technologies, WOOT’12*.

[45] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. 2000. *Software Tamper Resistance: Obstructing Static Analysis of Programs*. Technical Report. Charlottesville, VA, USA.

[46] Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. 2001. Protection of Software-Based Survivability Mechanisms. In *International Conference on Dependable Systems and Networks (DSN)*.

[47] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. 2011. Linear Obfuscation to Combat Symbolic Execution. In *European Symposium on Research in Computer Security, ESORICS*.

[48] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *Conference on Computer and Communications Security (CCS)*.

[49] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Symposium on Security and Privacy, SP*.

[50] Yongxin Zhou, Alec Main, Yuan Xiang Gu, and Harold Johnson. 2007. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *Information Security Applications, WISA*.

A STATISTICS ON DATASETS

We present additional statistics on Dataset #1 (Appendix Table 7) and Dataset #2 (Appendix Table 8). For Dataset #1, recall that 1-byte input programs from the original dataset from Banescu et al. [5] are automatically turned into equivalent 8-byte input programs with same number of paths: additional input are not used by the program, but latter protections will rely on them. We must do so as they are otherwise too easy to enumerate.

Table 7: Statistics on Dataset #1 (46 programs)

‘

Entry size	**#LOC**		**KLEE exec. (s)**	
	average	***StdDev.***	***average***	***StdDev.***
16 bytes	21	1.9	2.6s	6.2
1 byte (*)	17	2.2	1.8s	6.2

loc: line of code

(*) 1-byte input programs are automatically turned into equivalent 8-byte input programs with same number of paths. We report KLEE execution time on the modified versions.

Table 8: Statistics on Dataset #2 (7 programs)

Program	**locs**	**KLEE exec. (s)**
City hash	547	7.41
Fast hash	934	7.74
Spooky hash	625	7.12
MD5 hash	157	33.31
AES	571	1.42
DES	424	0.15
GRUB	101	0.06

B ADDITIONAL EXPERIMENTS

Search heuristics. Results in Appendix Table 9 shows that DSE search heuristics does not impact that much overall results (cf. Table 3). Depth-first search appears to be slightly better than the two other ones for Split, and non-uniform random search appears to be slightly worse than the two other ones for For. Nothing dramatic yet.

Table 9: Impact of search heuristics – Dataset #1 – secret finding – 1h TO

	Timeouts			
	NURS	**BFS**	**DFS**	**allpath**
Virt	0/15	0/15	0/15	0/15
Virt $\times 2$	0/15	0/15	0/15	0/15
Virt $\times 3$	1/15	1/15	1/15	2/15
Flat-Virt	0/15	0/15	0/15	0/15
Flat-MBA	0/15	0/15	0/15	0/15
Split ($\times 10$)	0/15	0/15	0/15	0/15
Split ($\times 13$)	1/15	1/15	0/15	1/15
For ($k = 1$)	0/15	0/15	0/15	0/15
For ($k = 2$)	1/15	1/15	1/15	4/15
For ($k = 3$)	10/15	8/15	8/15	13/15
For ($k = 4$)	**15/15**	**15/15**	**15/15**	**15/15**

Runtime overhead. We evaluate how the performance penalty evolved for protection For on very high values of k. We take the 15 examples of Dataset #1 with large input space, and we vary the size of the input string from 3 to 100000, increasing the number of forking points accordingly (k between 3 and 100000), one forking point (loop) per byte of the input string. We run 15 random inputs 15 times for each size and measure the average runtime overhead. Fig. 9 shows the evolution of runtime overhead w.r.t. the number of For loops.

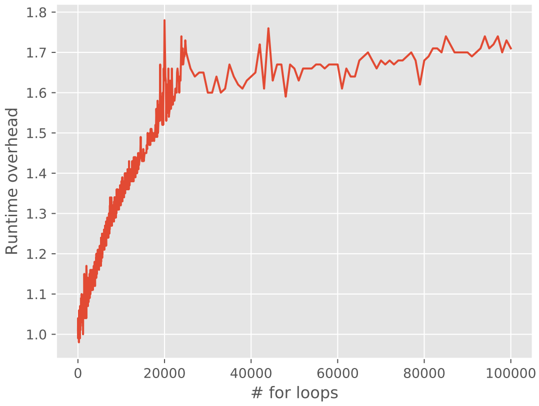

Figure 9: Runtime overhead w.r.t. to the number of For obfuscation loops

The runtime overhead stays below 5% ($\times 1.05$) for fewer than $k = 250$. This means in particular that one can significantly boost For-based protections without incurring big runtime penalties.

C MORE DETAILS ON EXPERIMENTS

We give here more detailed results on:

- the motivating example (Appendix Table 10),
- Path Exploration Dataset #1 (Appendix Table 11),
- Secret Finding DataSet #1 (Appendix Table 12).

Table 10: Benchmarking obfuscations on the crackme example – tool KLEE– 1h30 timeout.

	Obfuscation type		**Slowdown** *Symbolic Execution*		**Overhead** *Runtime*	**Overhead** *Code Size*
			Coverage	Secret		
Tigress	Virt ×2		$\times 5.4 \cdot 10^3$	×11	×1.3	×1.2
	Virt ×3		**TO**	$\times 1.1 \cdot 10^3$	×41	×3.0
	Virt ×4		**TO**	$\times 96 \cdot 10^3$	$\times 4.5 \cdot 10^3$	×4.0
	Virt ×5		**TO**	**TO**	$\times 449 \cdot 10^3$	×5.2
	Virt-Flat		×1.0	×1.8	×1.1	×1.5
	Flat ×2		×276	×1.8	×1.1	×1.3
	Flat-EncA		×83	×1.2	×1.0	×1.0
Our approach	For	$k = 1$	**TO**	$\times 3.3 \cdot 10^3$	×1.0	×1.0
		$k = 2$	**TO**	**TO**	×1.0	×1.0
		$k = 3$	**TO**	**TO**	×1.0	×1.0
	Split	$k = 11$	$\times 3.4 \cdot 10^3$	×2.8	×1.0	×1.0
		$k = 15$	TO	×3.9	×1.0	×1.0
		$k = 19$	**TO**	×5.1	×1.0	×1.0

Table 11: Obfuscations on Dataset #1 — allpath coverage — 3h timeout

	DSE Slowdown			**Runtime overhead**			**Code Size increase**			**#TO**
Transformation	min	max	avg	min	max	avg	min	max	avg	
Virt	×1.0	×17	×2.8	×1.2	×5.6	×1.5	×1.5	×1.5	×1.5	0/46
Virt ×2	×1.0	×402	×47	×1.3	×432	×15	×2.3	×2.8	×2.5	1/46
Virt ×3	×10	$\times 35 \cdot 10^3$	$\times 3.0 \cdot 10^3$	×3.2	$\times 52 \cdot 10^3$	$\times 1.6 \cdot 10^3$	×3.5	×4.6	×4	5/46
Flattening	×1.0	×1.3	×1.0	×1.0	×2.0	×1.8	×1.0	×1.0	×1.0	0/46
EncodeArithmetic	×1.0	×10	×3.9	×1.0	×2.0	×1.8	×1.0	×1.0	×1.0	0/46
Split ($k = 10$)	×10	$\times 1.2 \cdot 10^3$	×107	×1.0	×1.3	×1.2	×1.0	×1.0	×1.0	1/46
Split ($k = 13$)	×10	$\times 15 \cdot 10^3$	×862	×1.0	×1.3	×1.2	×1.0	×1.0	×1.0	4/46
For ($k = 1$)	×10	×476	×209	×1.0	×1.4	×1.2	×1.0	×1.0	×1.0	2/46
For ($k = 2$)	×10	$\times 33 \cdot 10^3$	$\times 3.7 \cdot 10^3$	×1.0	×1.4	×1.0	×1.0	×1.0	×1.0	10/46
For ($k = 3$)	×10	$\times 1.1 \cdot 10^6$	$\times 2.2 \cdot 10^5$	×1.0	×1.4	×1.3	×1.0	×1.0	×1.0	30/46
For ($k = 4$)	×10	$\times 2.2 \cdot 10^6$	$\times 5.1 \cdot 10^5$	×1.0	×1.4	×1.3	×1.0	×1.0	×1.0	**46/46**
Virt + For ($k = 2$)	×10	$\times 5.4 \cdot 10^5$	$\times 33 \cdot 10^3$	×1.0	×3.8	×1.2	×1.5	×1.6	×1.6	23/46

Table 12: Obfuscations on Dataset #1 — — Secret Finding — DFS heuristics, 1h timeout

	DSE slowdown			**Runtime overhead**			**#TO**
Transformation	min	max	avg	min	max	avg	
Virt	×1.0	×4.0	×1.6	×1.2	×1.4	×1.3	0/15
Virt ×2	×6	×268	×33	×1.3	×6.3	×2.5	0/15
Virt ×3	×557	$\times 4.7 \cdot 10^3$	$\times 1.7 \cdot 10^3$	×5.5	×513	×126	2/15
Flat-Virt	×1.0	×8.3	×2.3	×1.2	×1.5	×1.3	0/15
Flat-MBA	×2.0	×878	×59	×1.2	×1.3	×1.3	0/15
Split ($k = 10$)	×1.1	×9	×6	×1.0	×1.3	×1.2	0/15
Split ($k = 13$)	×1.1	×12	×8	×1.3	×1.8	×1.6	0/15
For ($k = 1$)	×7	$\times 1.1 \cdot 10^3$	×169	×1.0	×1.1	×1.0	0/15
For ($k = 2$)	×841	$\times 1.7 \cdot 10^5$	$\times 17 \cdot 10^3$	×1.0	×1.1	×1.0	1/15
For ($k = 3$)	$\times 2.3 \cdot 10^3$	$\times 3.6 \cdot 10^5$	$\times 1.6 \cdot 10^5$	×1.0	×1.1	×1.0	8/15
For ($k = 4$)	$\times 2.1 \cdot 10^5$	$\times 4.2 \cdot 10^5$	$\times 3.2 \cdot 10^5$	×1.0	×1.1	×1.0	**15/15**