2017 IEEE Symposium on Security and Privacy

Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes*

Sébastien Bardin
CEA, LIST,
91191 Gif-Sur-Yvette, France
sebastien.bardint@cea.fr

Robin David
CEA, LIST,
91191 Gif-Sur-Yvette, France
robin.david@cea.fr

Jean-Yves Marion
Université de Lorraine,
CNRS and Inria, LORIA, France
jean-yves.marion@loria.fr

***Abstract*—Software deobfuscation is a crucial activity in security analysis and especially in malware analysis. While standard static and dynamic approaches suffer from well-known shortcomings, Dynamic Symbolic Execution (DSE) has recently been proposed as an interesting alternative, more robust than static analysis and more complete than dynamic analysis. Yet, DSE addresses only certain kinds of questions encountered by a reverser, namely feasibility questions. Many issues arising during reverse, e.g., detecting protection schemes such as opaque predicates, fall into the category of infeasibility questions. We present Backward-Bounded DSE, a generic, precise, efficient and robust method for solving infeasibility questions. We demonstrate the benefit of the method for opaque predicates and call stack tampering, and give some insight for its usage for some other protection schemes. Especially, the technique has successfully been used on state-of-the-art packers as well as on the government-grade X-Tunnel malware – allowing its entire deobfuscation. Backward-Bounded DSE does not supersede existing DSE approaches, but rather complements them by addressing infeasibility questions in a scalable and precise manner. Following this line, we propose sparse disassembly, a combination of Backward-Bounded DSE and static disassembly able to enlarge dynamic disassembly in a guaranteed way, hence getting the best of dynamic and static disassembly. This work paves the way for robust, efficient and precise disassembly tools for heavily-obfuscated binaries.**

I. Introduction

Context. Obfuscation [1] is a prevalent practice aiming at protecting some functionalities or properties of a program. Yet, while its legitimate goal is intellectual property protection, obfuscation is widely used for malicious purposes. Therefore, (binary-level) software deobfuscation is a crucial task in reverse-engineering, especially for malware analysis.

A first step of deobfuscation is to recover the most accurate control-flow graph of the program (*disassembly*), i.e., to recover all instructions and branches of the program under analysis. This is already challenging for non-obfuscated codes due to tricky (but common) low-level constructs [2] like indirect control flow (computed jumps, `jmp eax`) or the interleaving of code and data. But the situation gets largely worst in the case of obfuscated codes.

* Work partially funded by ANR, grant 12-INSE-0002.

Standard disassembly approaches are essentially divided into *static methods* and *dynamic methods*. On one hand, static (syntactic) disassembly tools such as `IDA` or `Objdump` have the potential to cover the whole program. Nonetheless, they are easily fooled by obfuscations such as code overlapping [3], opaque predicates [4], opaque constants [5], call stack tampering [6] and self-modification [7]. On the other hand, dynamic analysis cover only a few executions of the program and might miss both significant parts of the code and crucial behaviors. *Dynamic Symbolic Execution* (DSE) [8], [9] (a.k.a *concolic execution*) is a recent and fruitful formal approach to automatic testing, recently proposed as an interesting approach for disassembly [10], [11], [12], [13], [14], more *robust* than static analysis and covering more instructions than dynamic analysis. *Currently, only dynamic analysis and DSE are robust enough to address heavily obfuscated codes.*

Problem. Yet, these dynamic methods only address reachability issues, namely *feasibility questions*, i.e., verifiying that certain events or setting can occur, e.g., that an instruction in the code is indeed reachable. Contrariwise, many questions encountered during reversing tasks are *infeasibility questions*, i.e., checking that certain events or settings cannot occur. It can be used either for detecting obfuscation schemes, e.g., detecting that a branch is dead, or to prove their absence, e.g., proving that a computed jump cannot lead to an improper address.

These *infeasibility issues are currently a blind spot of both standard and advanced disassembly methods*. Dynamic analysis and DSE do not answer the question because they only consider a *finite number of paths* while infeasibility is about considering *all paths*. Also, (standard) syntactic static analysis is too easily fooled by unknown patterns. Finally, while recent semantic static analysis approaches [15], [13], [16], [17] can in principle address infeasibility questions, they are currently neither scalable nor robust enough.

At first sight infeasibility is a simple mirror of feasibility, however from an algorithmic point of view they are not the same. Indeed, since solving feasibility questions on general programs is undecidable, practical approaches have to be one-sided, favoring either feasibility (i.e., answering *"feasible"* or

DOI 10.1109/SP.2017.36

"don't know") or infeasibility (i.e., answering *"don't know"* or *"infeasible"*). While there currently exist robust methods for answering feasibility questions on heavily obfuscated codes, no such method exist for infeasibility questions.

Goal and challenges. In this article, we are interested in solving automatically infeasibility questions occurring during the reversing of (heavily) obfuscated programs. The intended approach must be *precise* (low rates of false positives and false negatives) and able to *scale* on realistic codes both in terms of size (*efficient*) and protection – including self-modification (*robustness*), and *generic* enough for addressing a large panel of infeasibility issues. Achieving all these goals at the same time is particularly challenging.

Our proposal. We present *Backward-Bounded Dynamic Symbolic Execution* (BB-DSE), the first precise, efficient, robust and generic method for solving infeasibility questions. To obtain such a result, we have combined in an original and fruitful way, several state-of-the-art key features of formal software verification methods, such as deductive verification [18], bounded model checking [19] or DSE. Especially, the technique is *goal-oriented* for precision, *bounded* for efficiency and combines *dynamic information and formal reasoning* for robustness.

Contribution. The contribution of this paper are the following:

- First, we highlight the importance of infeasibility issues in reverse and the urging need for automating the investigation of such problems. Indeed, while many deobfuscation-related problems can be encoded as infeasibility questions (cf. Section V) it remains a blind spot of state-of-the-art disassembly techniques.
- Second, we propose the new *Backward-Bounded DSE* algorithm for solving infeasibility queries arising during deobfuscation (Section IV). The approach is both precise (low rates of false positives and false negatives), efficient and robust (cf. Table I), and it can address in a generic way a large range of deobfuscation-related questions – for instance opaque predicates, call stack tampering or self-modification (cf. Section V). The technique draws from several separated advances in software verification, and combines them in an original and fruitful way. We present the algorithm along with its implementation within the BINSEC open-source platform [1] [20], [21].
- Third, we perform an extensive experimental evaluation of the approach, focusing on two standard obfuscation schemes, namely *opaque predicates* and *call stack tampering*. In a set of *controlled experiments with ground truth* based on open-source obfuscators (cf. Section VI), we demonstrate that our method is very precise and efficient. Then, in a *large scale experiment with standard packers* (including self-modification and other advanced protections), the technique is shown to scale on realistic obfuscated codes, both in terms of efficiency and robustness (cf. Section VI).
- Finally, we present two practical applications of Backward-Bounded DSE. First, we describe an *in-depth case-study of the government-grade malware* X-TUNNEL [22] (cf. Section VIII), where BB-DSE allows to identify and remove all obfuscations (opaque predicates). We have been able to automatically extract a de-obfuscated version of functions – discarding almost 50% of dead and "spurious" instructions, and providing an insights into its protection schemes, laying a very good basis for further in-depth investigations. Second, we propose *sparse disassembly* (cf. Section IX), a combination of Backward-Bounded DSE, dynamic analysis and standard (recursive, syntactic) static disassembly allowing to *enlarge* dynamic disassembly *in a precise manner* – getting the best of dynamic and static techniques, together with encouraging preliminary experiments.

Discussion. Several remarks must be made about the work presented in this paper.

- First, while we essentially consider opaque predicates and call stack tampering, BB-DSE can also be useful in other obfuscation contexts, such as flattening or virtualization. Also self-modification is inherently handled by the dynamic aspect of BB-DSE.
- Second, while we present one possible combination for sparse disassembly, other combinations can be envisioned, for example by replacing the initial dynamic analysis by a (more complete) DSE [10] or by considering more advanced static disassembly techniques [2].
- Finally, some recent works target opaque predicate detection with standard forward DSE [12]. As already pointed out, DSE is not tailored to infeasibility queries, while BB-DSE is – cf. Sections VI and XI.

Impact. Backward-Bounded DSE does not supersede existing disassembly approaches, it complements them by addressing infeasibility questions. Altogether, this work paves the way for robust, precise and efficient disassembly tools for obfuscated binaries, through the careful combination of static/dynamic and forward/backward approaches.

TABLE I: Disassembly methods for obfuscated codes

	feasibility query	infeasibility query	efficiency	robustness
dynamic analysis	✓/×(†)	×	✓	✓
DSE	✓	×	×	✓
static analysis (syntactic)	✓	✓/×(††)	✓	×
static analysis (semantic)	×	✓	×	×
BB-DSE	×	✓(‡)	✓	✓

(†): follow only a few traces
(††): very limited reasoning abilities
(‡): can have false positive and false negative, yet very low in practice

[1] http://binsec.gforge.inria.fr/

II. Background

Obfuscation. These transformations [1] aim at hiding the real program behavior. While approaches such as virtualization or junk insertion make instructions more complex to understand, other approaches directly hide the legitimate instructions of the programs – making the reverser (or the disassembler) missing essential parts of the code while wasting its time in dead code. The latter category includes for example code overlapping, self-modification, opaque predicates and call stack tampering.

We are interested here in this latter category. For the sake of clarity, this paper mainly focuses on *opaque predicates* and *call stack tampering*.

- An *opaque predicate* always evaluates to the same value, and this property is ideally difficult to deduce. The infeasible branch will typically lead the reverser (or disassembler) to a large and complex portion of useless junk code. Figure 1 shows the `x86` encoding of the opaque predicate $7y^2 - 1 \neq x^2$, as generated by O-LLVM [23]. This condition is always false for any values of DS:X, DS:Y, so the conditional jump `jz <addr_trap>` is never going to be taken.
- A *(call) stack tampering*, or `call`/`ret` violation, consists in breaking the assumption that a `ret` instruction returns to the instruction following the call (*return site*), as exemplified in Figure 2. The benefit is twofold: the reverser might be lured into exploring useless code starting from the return site, while the real target of the `ret` instruction will be hidden from static analysis.

```
mov   eax, ds:x
mov   ecx, ds:y
imul  ecx, ecx
imul  ecx, 7
sub   ecx, 1
imul  eax, eax
cmp   ecx, eax
jz    <addr_trap> //false jump to junk
....  ........    //real code
```

Fig. 1: opaque predicate: $7y^2 - 1 \neq x^2$

<main>:	<fun>:
call <fun>	[...]
..... // return site	push X
..... // junk code	ret //jump to X instead
..... // junk code	//of return site

Fig. 2: Standard stack tampering

Disassembly. We call *legit* an instruction in a binary if it is executable in practice. Two expected qualities for disassembly are (1) *soundness: does the algorithm recover only legit instructions?*, (2) *completeness: does the algorithm recover all legit instructions?* Standard approaches include *linear sweep*, *recursive* disassembly and *dynamic disassembly*.

- *Recursive disassembly* statically explores the executable file from a given (list of) entry point(s), recursively following the possible successors of each instruction. This technique may miss a lot of instructions, typically due to computed jumps (`jmp eax`) or self-modification. The approach is also easily fooled into disassembling junk code obfuscated by opaque predicates or call stack tampering. As such, the approach is neither sound nor complete.
- *Linear sweep* linearly decodes all possible instructions in the code sections. The technique aims at being more complete than recursive traversal, yet it comes at the price of many additional misinterpreted instructions. Meanwhile, the technique can still miss instructions hidden by code overlapping or self-modification. Hence the technique is unsound, and incomplete on obfuscated codes.
- *Dynamic disassembly* retrieves only legit instructions and branches observed at runtime on one or several executions. The technique is sound, but potentially highly incomplete – yet, it does recover part of the instructions masked by self-modification, code overlapping, etc.

For example, while `Objdump` is solely based on linear sweep, `IDA` performs a combination of linear sweep and recursive disassembly (geared with heuristics).

Dynamic Symbolic Execution. Dynamic Symbolic Execution (DSE) [9], [8] (a.k.a *concolic execution*) is a formal technique for exploring program paths in a systematic way. For each path π, the technique computes a symbolic *path predicate* Φ_π as a set of constraints on the program input leading to follow that path at runtime. Intuitively, Φ_π is the conjunction of all the branching conditions encountered along π. This path predicate is then fed to an *automatic solver* (typically a SMT solver [24]). If a solution is found, it corresponds to an input data exercising the intended path at runtime. Path exploration is then achieved by iterating on all (user-bounded) program paths, and paths are discovered lazily thanks to an interleaving of dynamic execution and symbolic reasoning [25], [26]. Finally, *concretization* [25], [26], [27] allows to perform relevant under-approximations of the path predicate by using the concrete information available at runtime.

The main advantages of DSE are *correctness* (no false negative in theory, a bug reported is a bug found) and *robustness* (concretization does allow to handle unsupported features of the program under analysis without losing correctness). Moreover, the approach is easy to adapt to binary code, compared to other formal methods [28], [8], [29], [30]. The very main drawback of DSE is the so-called *path explosion problem*: DSE is doomed to explore only a portion of all possible execution paths. As a direct consequence, DSE is incomplete in the sense that it can only prove that a given path (or objective) is feasible (or coverable), but not that it is infeasible.

DSE is interesting for disassembly and deobfuscation since it enjoys the advantages of dynamic analysis (especially, sound disassembly and robustness to self-modification or code overlapping), while being able to explore a larger set of behaviors. Yet, while on small examples DSE can achieve

complete disassembly, it often only slightly improves coverage (w.r.t. pure dynamic analysis) on large and complex programs.

III. Motivation

Let us consider the obfuscated pseudo-code given in Figure 3. The function <main> contains an opaque predicate in ① and a call stack tampering in ②.

```
<main>:                      <fun1>:
if (C) { ①                   .....
   call <fun1>               push <X>②
   //junk ⓐ                  ret
}
else {
   call <fun2> ⓑ
}
//junk ⓒ                     <fun2>:
ret //fake end of fun        .... ⓓ
<X>:                         ret
//payload
```

Fig. 3: Motivating example

Getting the information related to the opaque predicate and the call stack tampering would allow:

- ① to know that <fun1> is always called and reciprocally that <fun2> is never called. As consequence ⓑ and ⓓ are dead instructions;
- ② to know that the ret of <fun1> is tampered and never return to the caller, but to <X>. As a consequence, ⓐ and ⓒ are dead instructions, and we discover the real payload located at <X>.

Hence the main motivation is not to be fooled by such infeasibility-based tricks that slow-down the program reverse-engineering and its global understanding.

Applications. The main application is to improve a disassembly algorithm with such information, since static disassembly will be fooled by such tricks and dynamic disassembly will only cover a partial portion of the program. Our goal is to design an efficient method for solving infeasibility questions. This approach could then passes the original code annotated with infeasibility highlights to other disassembly tools, which could take advantage of this information – for example by avoiding disassembling dead instructions. This view is depicted in Figure 4, and such a combination is discussed in Section IX.

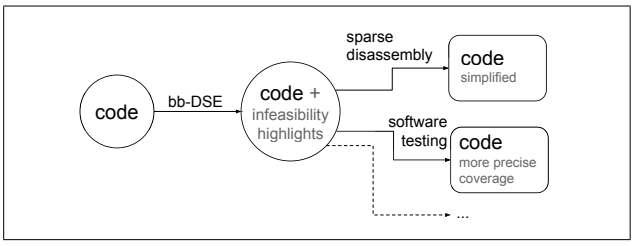


Fig. 4: motivation schema

Finally, infeasibility information could also be used in other contexts, e.g. , to obtain more accurate coverage rates in software testing, or to guide vulnerability analysis.

IV. Backward-Bounded DSE

We present in this section the new Backward-Bounded DSE technique for solving infeasibility queries on binary codes.

Preliminaries. We consider a binary-level program P with a given initial code address a_0. A state $s \triangleq (a, \sigma)$ of the program is defined by a code address a and a memory state σ, which is a mapping from registers and memory to actual values (bitvectors, typically of size 8, 32 or 64). By convention, s_0 represents an initial state, i.e., s_0 is of the form (a_0, σ). The transition from one state to another is performed by the $post$ function that executes the current instruction. An execution π is a sequence $\pi \triangleq (s_0 \cdot s_1 \cdot ... \cdot s_n)$, where s_{j+1} is obtained by applying the $post$ function to s_j (s_{j+1} is the successor of s_j).

Let us consider a predicate φ over memory states. We call *reachability condition* a pair $c \triangleq (a, \varphi)$, with a a code address. Such a condition c is *feasible* if there exists a state $s \triangleq (a, \sigma)$ and an execution $\pi_s \triangleq (s_0 \cdot s_1 \cdot ... \cdot s)$ such that σ satisfies φ, denoted $\sigma \models \varphi$. It is said *infeasible* otherwise. A *feasibility (resp. infeasibility) question* consists in trying to solve the feasibility (resp. infeasibility) of such a reachability condition.

Note that while these definitions do not take self-modification into account, they can be extended to such a setting by considering code addresses plus waves or phases [3], [31].

Principles. We build on and combine 3 key ingredients from popular software verification methods:

- backward reasoning from deductive verification, for *precise goal-oriented reasoning*;
- combination of dynamic analysis and formal methods (from DSE), for *robustness*;
- bounded reasoning from bounded model checking, for *scalability* and the ability *to perform infeasibility proofs*.

The initial idea of BB-DSE is to perform a **backward reasoning**, similar to the one of DSE but going from successors to predecessors (instead of the other way). Formally, DSE is based on the $post$ operation while BB-DSE is based on its inverse pre. Perfect backward reasoning pre^* (i.e., fixpoint iterations of relation pre, collecting all predecessors of a given state or condition) can be used to check feasibility and infeasibility questions. But this relation is not computable.

Hence, we rely on computable **bounded reasoning**, namely pre^k, i.e., collecting all the "predecessors in k steps" (k-predecessors) of a given state (or condition). Given a reachability condition c, if $pre^k(c) = \emptyset$ then c is infeasible (unreachable). Indeed, if a condition has no k-predecessor, it has no k'-predecessor for any $k' > k$ and cannot be reached. Hence, pre^k can answer *positively to infeasibility queries*. Yet, symmetry does not hold anymore, as pre^k *cannot falsify infeasibility queries* – because it could happen that a condition is infeasible for a reason beyond the bound k. The example in Figures 6 and 7 give an illustration of such a situation. In this case,

we have a ***false negative (FN)***, i.e. a reachability condition wrongly identified as feasible because of a too-small k.

In practice, *when the control-flow graph of the program (CFG) is available*, checking whether $pre^k = \emptyset$ can be easily done in a symbolic way, like it is done in DSE: the set pre^k is computed implicitly as a logical formula (typically, a quantifier-free first-order formula over bitvectors and arrays), which is unsatisfiable iff the set is empty. This formula is then passed to an automatic solver, typically a SMT solver [24] such as `Z3`. Moreover, it is efficient as the computation does not depend on the program size but on the user-chosen bound k.

Yet, backward reasoning is very fragile at binary-level, since computing a precise CFG may be highly complex because of dynamic jumps or self-modification. The last trick is to combine this pre^k reasoning with **dynamic traces**, so that the whole approach benefits from the robustness of dynamic analysis. Actually, the pre^k is now computed w.r.t. the control-flow graph *induced by a given trace* π – in a dynamic disassembly manner. We denote this *sliced* pre^k by pre^k_π.

Hence we get *robustness*, yet since some parts of pre^k may be missing from pre^k_π, we *now lose correctness* and may have ***false positive (FP)***, i.e., reachability conditions wrongly identified as infeasible, additionally to the false negative FN due to "boundedness" (because of too small k). A picture of the approach is given in Figure 5.

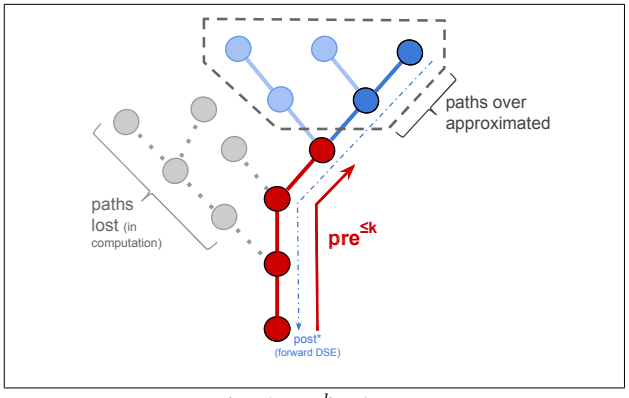


Fig. 5: pre^k schema

BB-DSE **through example.** We now illustrate BB-DSE on a toy example along with the impact of the bound k and of the (set of) dynamic traces on FP and FN. Figure 6 shows a simple pseudo-code program, where branch condition `x″ ≠ y′` always evaluate to true (*opaque predicate*) – as it encodes condition $7x^2 - 1 \neq y^2$ on the program input `x` and `y`. The two other branch conditions can evaluate to both true and false, depending on the input. Figure 7 shows the partial CFG obtained by dynamic execution on the toy example, where the call to function `even` is inlined for simplicity. We consider two traces: π_1 covers bold edges (true, true), and π_2 covers dash edges (false, false).

```
<main>:                          <even(int a)>:
x = input()                      if (a % 2 == 0) {
y = input()                        res = 1
x' = 7*(x*x)                     }
result = even(y)                 else {
y' = y*y                           res = 0
x"= x'-1                         }
if (result) { ①                  return res
  if(x" ≠ y'){ ②
    //always taken
  }
  else {//dead }
}
else {
  ...
}
```

Fig. 6: Toy example

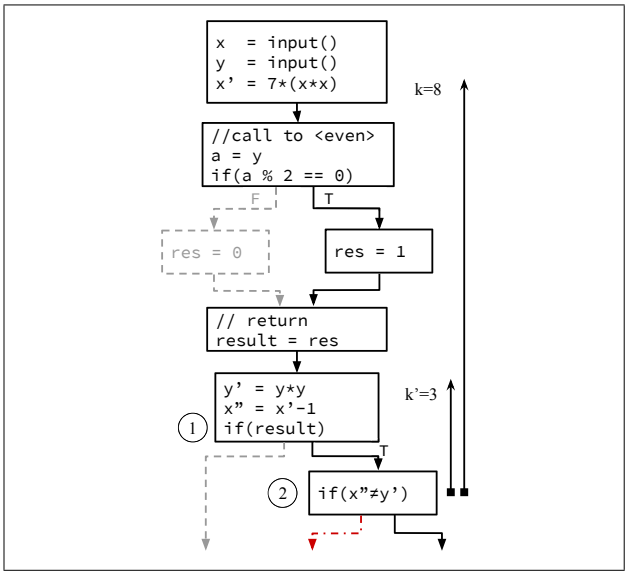


Fig. 7: Partial CFG from toy example

Suppose we want to use BB-DSE to prove that branch condition ② is indeed opaque, i.e., that `x″=y′` is infeasible at program location ②. The algorithm goes backward from program location ② and predicate $x'' = y'$, and gathers back all dynamic suffixes up to the bound k. Considering only trace π_1 (bold edges) and $k = 8$, we obtain (after substitution): $pre^k_{\pi_1} \triangleq 7x^2 - 1 = y^2 \wedge result = 1 \wedge result \neq 0 \wedge y\%2 = 0$, which is UNSAT, as $7x^2 - 1 = y^2$ is UNSAT. Hence, *branch condition ② is indeed proved opaque.* In the case where we consider also π_2, then $pre^k_{\pi_1,\pi_2} \triangleq (7x^2 - 1 = y^2) \wedge ((y\%2 = 0 \wedge result = 1 \wedge result \neq 0) \vee (y\%2 \neq 0 \wedge result = 0 \wedge result \neq 0))$, where $pre^k_{\pi_1,\pi_2}$ is obtained by simplifying the disjunction of both formulas $pre^k_{\pi_1}$ and $pre^k_{\pi_2}$. It is easy to see that $pre^k_{\pi_1,\pi_2}$ is also UNSAT. Once again, *branch condition ② is successfully proved opaque.*

We now illustrate the case where our technique misses an

infeasible condition (FN). Consider once again traces π_1, π_2 and branch condition ②, with bound $k' = 3$. Then $pre^{k'}_{\pi_1,\pi_2} \triangleq x' - 1 = y^2 \wedge result \neq 0$, which is satisfiable (with $x' = 1, y = 0, result = 1$). Hence, *branch condition ② is not proved opaque*. We miss here an unfeasible condition because of a too-small bound k', yielding a false negative (FN).

Finally, we illustrate the case where our technique can wrongly identify a condition as infeasible (FP). We are interested now in deciding whether branch condition ① can take value false, i.e., if `result` can be 0 at program location ①. We consider trace π_1 and bound $k'' = 4$ (or higher). We obtain $pre^{k''}_{\pi_1} \triangleq result = 0 \wedge \ldots \wedge result = res \wedge res = 1$, which is UNSAT, and we *wrongly conclude that branch condition ① is opaque, because of the missing path* where `res` is assigned to `0`. This corresponds to a false positive (FP). If we consider also π_2, then $pre^{k''}_{\pi_1,\pi_2} \triangleq result = 0 \wedge x'' = x' - 1 \wedge y' = y^2 \wedge result = res \wedge (res = 1 \vee res = 0)$ is satisfiable (with $y' = y = x'' = 0, x' = 1, res = 0$) and branching condition ① is now (correctly) not identified as opaque.

Algorithm. Considering a reachability condition (a, φ), BB-DSE starts with a dynamic execution π:

- if π reaches code address a, then compute $pre^k_\pi((a, \varphi))$ as a formula and solve it
  - if it is UNSAT, then the result is INFEASIBLE;
  - if it is SAT, then the result is UNKOWN;
  - if it is TO (timeout), then the result is TO;
- otherwise the result is UNKOWN.

As a summary, this algorithm enjoys the following good properties: it is efficient (depends on k, not on the trace or program length) and as robust as dynamic analysis. On the other hand, the technique may report both false negative (bound k too short) and false positive (dynamic CFG recovery not complete enough). *Yet, in practice, our experiments demonstrate that the approach performs very well, with very low rates of FP and FN.* Experiments are presented in Sections VI, VII and VIII.

We will not distinguished anymore between the predicate φ and the reachability condition (a, φ), when clear from context.

Impact of the bound on correctness and completeness. In the ideal case where the dynamic CFG recovery is perfect w.r.t. the bound k, i.e., $pre^k_\pi = pre^k$ (all suffixes of size k have been collected by the trace), the technique has no false positive FP and the effect of k is (as expected) a tradeoff between computation cost and false negatives FN: longer suffixes allow to correctly identify more infeasible conditions. Things are less intuitive when pre^k_π is incomplete, i.e. $pre^k_\pi \subset pre^k$. There, the technique yields also FP because of missing suffixes (cf. previous example). Since a larger k means more room to miss suffixes, it yields also more FP. Hence, in the general case a larger k leads to both less FN and more FP [2].

A straightforward way to decrease the number of FP is to consider more dynamic traces in order to obtain a "more complete" dynamic CFG and come closer to the ideal case above (cf. toy example in Figure 7). As such, the technique can benefit from fuzzing or standard (forward) DSE.

Implementation. This algorithm is implemented on top of BINSEC/SE [21], a forward DSE engine inside the open-source platform BINSEC [20] geared to formal analysis of binary codes. The platform currently proposes a front-end from `x86` (32bits) to a generic intermediate representation called DBA [32] (including decoding, disassembling, simplifications). It also provides several semantic analyses, including the BINSEC/SE DSE engine [21]. BINSEC/SE features a strongly optimized path predicate generation as well as highly configurable search heuristics [21], [13] and C/S policies [27]. The whole platform[3] amounts for more than 40k of OCaml line of codes (loc). BINSEC also makes use of two other components. First, the dynamic instrumentation called PINSEC, based on Pin, in charge of running the program and recording runtime values along with self-modification layers. Written in C++ it amounts for 3kloc. Second, IDASEC is an `IDA` plugin written in Python (∼13kloc) aiming at triggering analyzes and post-processing results generated by BINSEC.

The BB-DSE algorithm is tightly integrated in the BINSEC/SE component. Indeed, when solving a predicate feasibility, BINSEC/SE DSE performs a backward pruning pass aiming at removing any useless variable or constraint. BB-DSE works analogously, but takes into account the distance from the predicate to solve: any definition beyond the (user-defined) k bound is removed. In a second phase, the algorithm creates a new input variable for any variable used but never defined in the sliced formula. Actually, we do not compute a single formula for pre^k_π, but enumerate its suffixes (without repetition) – this could be optimized. For a given suffix the algorithm is standard [27]. Yet, we stay in a purely symbolic setting (no concretization) with formulas over bitvectors and arrays, making simplifications [21] important.

V. SOLVING INFEASIBILITY QUESTIONS WITH BB-DSE

We show in this section how several natural problems encountered during deobfuscation and disassembly can be thought of as infeasibility questions, and solved with BB-DSE.

A. Opaque Predicates

As already stated in Section II, an opaque predicate (OP) is a predicate always evaluating to the same value. They have successfully been used in various domains [33], [1]. Recent works [12] identify three kinds of opaque predicates:

- *invariant*: always true/false due to the structure of the predicate itself, regardless of inputs values,
- *contextual*: opaque due to the predicate and its constraints on input values,
- *dynamic*: similar to contextual, but opaqueness comes from dynamic properties on the execution (e.g., memory).

Approach with BB-DSE. Intuitively, to detect an opaque predicate the idea is to backtrack all its data dependencies

[2] cf. Figure 14 in Appendix.

[3] http://binsec.gforge.inria.fr/tools

and gather enough constraints to conclude to the infeasibility of the predicate. If the predicate is local (invariant), the distance from the predicate to its input instantiation will be short and the predicate will be relatively easy to break. Otherwise (contextual, dynamic) the distance is linear with the trace length, which does not necessarily scale.

This is a direct application of BB-DSE, where $p \triangleq (a, \varphi)$ is the pair address-predicate for which we want to check for opacity. We call π the execution trace under attention (extension to a set of traces is straightforward). Basically, the detection algorithm is the following:

- if p is dynamically covered by π, then returns FEASIBLE;
- otherwise, returns BB-DSE (p), where INFEASIBLE is interpreted as "opaque".

Results are guaranteed solely for FEASIBLE, since BB-DSE has both false positives and negatives. Yet, experiments (Sections VI-VIII) show that error ratios are very low in practice.

Concerning the choice of bound k, experiments in Section VI demonstrates that a value between 10 and 20 is a good choice for invariant opaque predicates. Interestingly, the X-TUNNEL case study (Section VIII) highlights that such rather small bound values may be sufficient to detect opaque predicates with long dependency chains (up to 230 in the study, including contextual opaque predicates), since we do not always need to recover all the information to conclude to infeasibility.

B. Call Stack Tampering

Call stack tampering consists in altering the standard compilation scheme switching from function to function by associating a `call` and a `ret` and making the `ret` return to the call next instruction (*return site*). The `ret` is tampered (a.k.a violated) if it does not return to the expected return site.

New taxonomy. In this work we refine the definition of a stack tampering in order to characterize it better.

- ***integrity***: does `ret` return to the same address as pushed by the `call`? It characterizes if the tampering takes place or not. A `ret` is then either `[genuine]` (always returns to the caller) or `[violated]`.
- ***alignment***: is the stack pointer (`esp`) identical at `call` and `ret`? If so, the stack pointer is denoted `[aligned]`, otherwise `[disaligned]`.
- ***multiplicity***: in case of violation, is there only one possible `ret` target? This case is noted `[single]`, otherwise `[multiple]`.

Approach with BB-DSE. The goal is to check several properties of the tampering using BB-DSE. We consider the following predicates on a `ret` instruction:

- $@[esp_{\{call\}}] = @[esp_{\{ret\}}]$: Compare the content of the value pushed at call $@[esp_{\{call\}}]$ with the one used to return $@[esp_{\{ret\}}]$. If it evaluates to VALID, the `ret` cannot be tampered `[genuine]`. If it evaluates to UNSAT, a violation necessarily occurs `[violated]`. Otherwise, cannot characterize integrity.
- $esp_{\{call\}} = esp_{\{ret\}}$: Compare the logical ESP value at the `call` and at `ret`. If it evaluates to VALID, the `ret` necessarily returns at the same stack offset `[aligned]`, if it evaluates to UNSAT the `ret` is `[disaligned]`. Otherwise cannot characterize alignment.
- $\mathcal{T} \neq @[esp_{\{ret\}}]$: Check if the logical `ret` jump target $@[esp_{\{ret\}}]$ can be different from the concrete value from the trace ($\mathcal{T}$). If it evaluates to UNSAT the `ret` cannot jump elsewhere and is flagged `[single]`. Otherwise cannot characterize multiplicity.

The above cases can be checked by BB-DSE (for checking VALID with some predicate ψ, we just need to query BB-DSE with predicate $\neg\psi$). Then, our detection algorithm works as follow, taking advantage of BB-DSE and dynamic analysis:

- the dynamic analysis can tag a `ret` as: `[violated]`, `[disaligned]`, `[multiple]`;
- BB-DSE can tag a `ret` as: `[genuine]`, `[aligned]`, `[single]` (`[violated]` and `[disaligned]` are already handled by dynamic analysis).

As for opaque predicates, dynamic results can be trusted, while BB-DSE results may be incorrect. Table II summarizes all the possible situations.

TABLE II: Call stack tampering detection

RT Status	integrity	alignment	multiplicity
RT Genuine	VALID: `[genuine]`	RT: KO `[disaligned]` - VALID: `[aligned]`	
RT Tampered `[violated]`		RT: KO `[disaligned]` - VALID: `[aligned]`	RT: (2+) `[multiple]` - UNSAT: `[single]`

This call stack tampering analysis uses BB-DSE, but with a slightly non-standard setting. Indeed, in this case the bound k will be different for every `call`/`ret` pair. The trace is analysed in a forward manner, keeping a formal stack of `call` instructions. Each `call` encountered is pushed to the formal stack. Upon `ret`, the first `call` on the formal stack is poped and BB-DSE is performed, where k is the distance between the `call` and the `ret`.

From an implementation point of view, we must take care of possible corruptions of the formal stack, which may happen for example in the following situations:

- Call to a non-traced function: because the function is not traced, its `ret` is not visible. In our implementation these calls are not pushed in the formal stack;
- Tail call [2] to non-traced function: tail calls consists in calling functions through a jump instruction instead of `call` to avoid stack tear-down. This is similar to the previous case, except that care must be taken in order to detect the tail call.

C. Other deobfuscation-related infeasibility issues

Opaque constant. Similar to opaque predicates, opaque constants are expressions always evaluating to a single value. Let us consider the expression e and a value v observed at runtime for e. Then, the opaqueness of e reduces to the infeasibility of $e \neq v$.

Dynamic jump closure. When dealing with dynamic jumps, switch, etc., we might be interested in knowing if all the

targets have been found. Let us consider a dynamic jump `jump eax` for which 3 values v_1, v_2, v_3 have been observed so far. Checking the jump closure can be done through checking the infeasibility of $eax \neq v_1 \wedge eax \neq v_2 \wedge eax \neq v_3$.

Virtual Machine & CFG flattening. Both VM obfuscation and CFG flattening usually use a custom instruction pointer aiming at preserving the flow of the program after obfuscation. In the case of CFG flattening, after execution of a basic block the virtual instruction pointer will be updated so that the dispatcher will know where to jump next. As such, we can check that all observed values for the virtual instruction pointer have been found for each flattened basic block. Thus, if for each basic block we know the possible value for the virtual instruction pointer and have proved it cannot take other values, we can ultimately get rid of the dispatcher.

A glimpse of conditional self-modification. Self-modification is a killer technique for blurring static analysis, since the real code is only revealed at execution time. The method is commonly found in malware and packers, either in simple forms (unpack the whole payload at once) or more advanced ones (unpack on-demand, shifting-decode schemes [34]). The example in Figure 8 (page 10) taken from `ASPack` combines an opaque predicate together with a self-modification trick turning the predicate to true in order to fool the reverser. Other examples from existing malwares have been detailed in previous studies (NetSky.aa [10]).

Dynamic analysis allows to overcome the self-modification as the new modified code will be executed as such. Yet, BB-DSE can be used as well, to *prove interesting facts about self-modification schemes*. For example, given an instruction known to perform a self-modification, we can take advantage of BB-DSE to know whether another kind of modification by the same instruction is possible or not (conditional self-modification). Let us consider an instruction $mov\ [addr],\ eax$ identified by dynamic analysis to generate some new code with value $eax = v$. Checking whether the self modification is conditional reduces to the infeasibility of predicate $eax \neq v$.

As a matter of example, this technique has been used on the example of Figure 8 to show that no other value than 1 can be written. This self-modification is thus unconditional.

VI. Evaluation: Controlled Experiments

We present a set of controlled experiments with *ground truth values* aiming at evaluating the precision of BB-DSE as well as giving hints on its efficiency and comparing it with DSE.

A. Preliminary: Comparison with Standard DSE

As already stated, forward DSE is not fit to infeasibility detection, both in terms of scalability and error rate (false positive, FP), since DSE essentially proves the infeasibility of *paths*, not of reachability conditions. The goal of this preliminary experiment is to illustrate this fact clearly, since DSE is sometimes used for detecting opaque predicates [12]. We consider a trace of 115000 instructions *without any opaque predicate*, and we check at each conditional jump if the branch not taken is proved infeasible (if so, this is a FP). We take the BB-DSE algorithm for opaque predicate from Section V, with bound $k = 20$, which is a reasonable value (cf. Section VI-B). We take the forward DSE of BINSEC/SE. Results are presented in Table III. As expected, BB-DSE is much more efficient than DSE and yields far less FP and timeouts (TO).

These results were expected, as they are direct consequences of the design choices behind DSE and BB-DSE. On the opposite, BB-DSE is not suitable for feasibility questions.

TABLE III: Benchmark DSE versus BB-DSE

	bound	Cond. branch		Total
	k	# FP	#TO	time
forward DSE	-	**7749**	**2460**	**17h43m**
BB-DSE	20	**54**	**0**	**4m14s**

total number of queries: 10784 – TO: timeout (60 seconds)
#FP: #false positive – no false negative on this example

B. Opaque Predicates evaluation

We consider here the BB-DSE-based algorithm for opaque predicate detection. We want to evaluate its precision, as well as to get insights on the choice of the bound k.

Protocol and benchmark. We consider two sets of programs: (1) all 100 `coreutils` without any obfuscation, as a genuine reference data set, and (2) 5 simple programs taken from the State-of-the-Art in DSE deobfuscation [10] and obfuscated with O-LLVM [23]. Each of the 5 simple programs was obfuscated 20 times (with different random seeds) in order to balance the numbers of obfuscated samples and genuine `coreutils`. We have added new opaque predicates, listed in Table IV, in O-LLVM (which is open-source) in order to maximize diversity.

TABLE IV: OP implemented in O-LLVM

Formulas		Comment
$\forall x, y \in \mathcal{Z}$	$y < 10 \| \| 2 \| (x \times (x-1))$	(initially present in O-LLVM)
$\forall x, y \in \mathcal{Z}$	$7y^2 - 1 \neq x^2$	
$\forall x \in \mathcal{Z}$	$2 \| (x + x^2)$	
$\forall x \in \mathcal{Z}$	$2 \| \lfloor \frac{x^2}{2} \rfloor$	(2^{nd} bit of square always 0)
$\forall x \in \mathcal{Z}$	$4 \| (x^2 + (x+1)^2)$	
$\forall x \in \mathcal{Z}$	$2 \| (x \times (x+1))$	

In total, 200 binary programs were used. For each of them a dynamic execution trace was generated with a maximum length of 20.000 instructions. By tracking where opaque predicates were added in the obfuscated files, we are able a priori to know if a given predicate is opaque or not, ensuring a *ground truth evaluation*. Note that we consider all predicates in `coreutils` to be genuine. The 200 samples sums up a total of 1,091,986 instructions trace length and 11,725 conditional jumps with 6,170 genuine and 5,556 opaque predicates. Finally, experiments were carried using different values for the bound k, and with a 5 second timeout per query.

Results. Among the 11,725 predicates, 987 were fully covered by the trace and were excluded from these results, keeping

10,739 predicates (and 5,183 genuine predicates). Table V (and Figure 14 in Appendix) shows the relation between the number of predicates detected as opaque (OP) or genuine, false positive (*FP, here: classify a genuine predicate as opaque*) and false negatives (*FN, here: classify an opaque predicate as genuine*) depending of the bound value k. The experiment shows a tremendous peak of opaque detection with $k = 12$. Alongside, the number of false negative steadily decreases as the number of false positive grows. An optimum is reached for $k = 16$, with *no false negative*, no timeout and a small number of false positive (372), representing an error rate of 3.46%, while the smallest error rate (2.83%) is achieved with $k = 12$. Results are still very precise up to k = 30, and very acceptable for k = 50.

TABLE V: Opaque predicate detection results

k	OP (5556)		Genuine (5183)		TO	Error rate (FP+FN)/Tot (%)	Time (s)	avg/query (s)
	ok	miss (FN)	ok	miss (FP)				
2	0	5556	5182	1	0	51.75	89	0.008
4	903	4653	5153	30	0	43.61	96	0.009
8	4561	995	4987	196	0	11.09	120	0.011
12	**5545**	**11**	**4890**	**293**	**0**	**2.83**	**152**	**0.014**
16	**5556**	**0**	**4811**	**372**	**0**	**3.46**	**197**	**0.018**
20	**5556**	**0**	**4715**	**468**	**2**	**4.36**	**272**	**0.025**
24	**5556**	**0**	**4658**	**525**	**7**	**4.89**	**384**	**0.036**
32	5552	4	4579	604	25	5.66	699	0.065
40	5548	8	4523	660	39	6.22	1145	0.107
50	5544	12	4458	725	79	6.86	2025	0.189

. 10,739 predicates, 5,556 opaque predicates, 5,183 genuine predicates
. ok: well-classified predicate – miss: badly-classified predicate
. Timeout (TO): 5 sec, counts UNKNOWN (classifies the predicate as genuine)

A glimpse at efficiency. Taking the same detection algorithm and queries, we want to evaluate predicate solving time. Each predicate formula is solved using Z3[4] with a 5 seconds timeout. Table V gives for each value of k the total time taken for solving and the average time per query. For k = 16 the average time per query is 0.018s which proves that the technique scales. Previous works based on forward DSE aiming at solving invariant opaque predicates [12] reports an average of 0.49s per queries (min:0.09, max:0.79). As expected, BB-DSE provides a very significant speed up.

Conclusion. BB-DSE is very precise at detecting opaque predicates, and while the choice of bound is important, the technique still performs very well if the bound is not too far from the optimal one (k between 10 and 30). Query solving time is also very low.

C. Call Stack Tampering evaluation

We want to assess the precision BB-DSE for stack tampering detection (Section VI-C). Especially, we would like the technique to identify as *genuine* or *single-target tampered* (cf. Section VI-C) as many `ret` as possible, so that latter analyses (human or automated) do not have to reason about them anymore. We consider two sets of programs: (1) the 5 samples used Section VI-B, obfuscated with the source-to-source obfuscator Tigress [35]; (2) 5 (randomly selected) `coreutils` without any obfuscation. The 5 samples from Section VI-B have been obfuscated with the AntiBranchAnalysis transformation that replaces all conditional branches with calls and rets. The two schemes used by the obfuscator are:

- push; call; ret; ret (the last `ret` jumps on pushed value)
- push; ret (idem)

Results. Results are given in Table VI. The 5 obfuscated samples sums up 218 different `ret`. Among them 77 are genuine and 141 violated (including 132 single-target). Our detection method achieves here a *perfect score*, with no false positive nor false negative. On the 5 `coreutils`, BB-DSE does not yield *any false positive* and most of the `ret` are proved genuine (149/156). The few remaining unproved `ret` come from unhandled libc side-effects.

TABLE VI: Stack tampering results

Sample	runtime genuine			runtime violation		
	#`ret` †	proved genuine	proved a/d	#`ret` †	proved a/d	proved single
obfuscated programs						
simple-if	6	6	6/0	9	0/0	8
bin-search	15	15	15/0	25	0/0	24
bubble-sort	6	6	6/0	15	0/1	13
mat-mult	31	31	31/0	69	0/0	68
huffman	19	19	19/0	23	0/3	19
non-obfuscated programs						
ls	30	30	30/0	0	-	-
dir	35	35	35/0	0	-	-
mktemp	21	20	20/0	0	-	-
od	21	21	21/0	0	-	-
vdir	49	43	43/0	0	-	-

†each `ret` is counted only once – a: aligned, d: disaligned (cf. Sect. VI-C)

Conclusion. BB-DSE performs very well here, with no false positive and a perfect score on obfuscated samples. The technique recovers both genuine `ret` and single-target tampered `ret`. Interestingly, no tampered `ret` were found on the few (randomly selected) `coreutils`, supporting the idea that such tampering is not meant to occur in legitimate programs.

D. Conclusion

These different controlled experiments demonstrate clearly that BB-DSE is a very precise approach for solving different kinds of infeasibility questions. They also demonstrate that finding a suitable bound k is not a problem in practice. Finally, the approach seems to be scalable. This last point will be definitely proved in Sections VII and VIII.

VII. Large-scale Evaluation on Packers

To validate the scalability of BB-DSE on representative codes, in terms of both size and protection, we perform a large scale experiment on packers with the two detection algorithms already used in Section VI.

Context. Packers are programs embedding other programs and decompressing/deciphering them at runtime. Since packers are

[4]http://github.com/Z3Prover/z3

used for software protection, most of them contain several obfuscation schemes (including *self-modification*). As a matter of fact, packers are also widely used by malware, and actually in many cases they are the only line of defense. Hence, *packers are very representative* for our study, both in terms of malware protections and size, as packed programs tend to have huge execution traces.

Protocol. We want to check if BB-DSE is able to detect opaque predicates or call stack tampering on packed programs. For that, a large and representative set of packers was chosen, ranging from free to commercial tools. Then a stub binary (`hostname`) was packed by each packer. Analyses are then triggered on these packed programs in a black-box manner, that is to say, without any prior knowledge of the internal working of the packers – we do not know which obfuscation are used. For homogeneity, trace length are limited to 10M instructions and packers reaching this limit were not analysed.

A. Results

Table VII shows the partial results on 10 packers. The complete results are given in Table XVI in Appendix. First, BB-DSE is efficient and robust enough to pass on most of the packed programs, involving very long traces ($\geq$ million of instructions) and advanced protections such as self-modification. Second, over the 32 packers, 420 opaque predicates and 149 call/stack tampering have been found, and many `ret` have been proved genuine. All the results that have been manually checked appeared to be true positive (we did not checked them all because of time constraints).

B. Other Discoveries

Opaque predicates. Results revealed interesting patterns, for instance ACProtect tends to add opaque predicates by chaining conditional jumps that are mutually exclusive like: `jl 0x100404c ; jge 0x100404c`. In this example the second jump is necessarily opaque since the first jump strengthens the path predicate, enforcing the value to be lower. This example shows that our approach can detect both invariant and contextual opaque predicates. Many other variants of this pattern were found: `jp/jnp`, `jo/jno`, etc. Similarly, the well-known opaque predicate pattern `xor ecx, ecx; jnz` was detected in ARMADILLO. Because of the `xor`, the non-zero branch of `jnz` is never taken.

The dynamic aspect of BB-DSE allowed to bypass some tricks that would misled a reverser into flagging a predicate as opaque. A good example is a predicate found in `ASPack` seemingly opaque but that turned not to be opaque *due to a self-modification* (Figure. 8). Statically, the predicate is opaque since BL is necessarily 0 but it turns out that the second opcode bytes of the MOV BL, 0X0 is being patched to 1 in one branch in order to take the other branch when looping back later on.

Call/stack tampering. According to the taxonomy of Section V, many different kinds of violations are detected. For instance, the two patterns found in ACProtect (Figures 9 and 10) are detected as `[violated]`, `[disaligned]`, `[single]` and

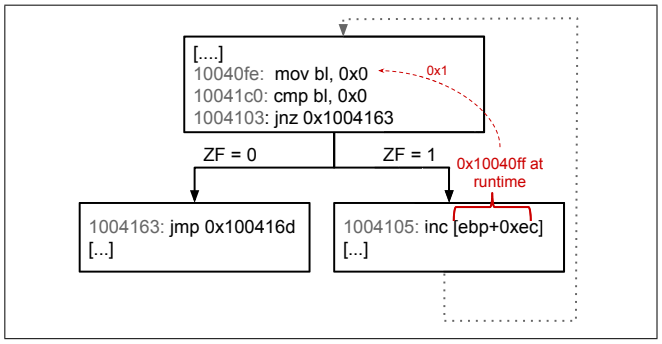


Fig. 8: `ASPack` opaque predicate decoy

`[violated]`, `[aligned]`, `[single]`. More details can be found in Appendix. Especially, in Aspack, stack tampering detection allows to find precisely that moment in the trace, where the packer payload (i.e., the original unpacked program) is very likely decompressed in memory.

address	mnemonic	comment
1004328	call 0x1004318	//push 0x100432d as return
1004318	add [esp], 9	//tamper the value in place
100431c	ret	//return to 0x1004n336

Fig. 9: ACProtect violation 1/2

address	mnemonic	comment
1001000	push 0x1004000	
1001005	push 0x100100b	
100100a	ret	jump on the ret below
100100b	ret	jump on 0x1004000

Fig. 10: ACProtect violation 2/2

C. Conclusion

By detecting opaque predicates and call/stack tampering on packers with very long trace length, this experiment clearly demonstrates both the ability of BB-DSE to scale to realistic obfuscated examples (without any prior-knowledge of the protection schemes) and its usefulness. This study yields also a few unexpected and valuable insights on the inner working on the considered packers, such as some kinds of protections or the location of the jump to the entrypoint of the original unpacked program.

VIII. REAL-WORLD MALWARE: X-TUNNEL

A. Context & Goal

Context. As an application of the previous techniques we focus in this section on the heavily obfuscated X-TUNNEL malware. X-TUNNEL is a ciphering proxy component allowing the X-AGENT malware to reach the command and control (CC) if it cannot reach it directly [22]. It is usually the case for machines

TABLE VII: Packer experiment, OP & Stack tampering

Packers	Static	Dynamic information					Obfuscation detection				
	size					(self-mod.)	Opaque Pred.			Stack tampering	
	prog	#tr.len	(tr.ok/host)	#proc	#th	#layers	Unk	OP	TO	RT_{ok} (a/d/g)	RT_{ko} (a/d/s)
ACProtect v2.0	101K	1.8M	(✓,×)	1	1	4	74	159	0	0 (0/0/0)	48 (45/1/45)
ASPack v2.12	10K	377K	(✓,✓)	1	1	2	32	24	0	11 (7/0/7)	6 (1/4/1)
Crypter v1.12	45K	1.1M	(✓,×)	1	1	0	263	24	0	125 (94/0/94)	78 (0/30/32)
Expressor	13K	635K	(✓,✓)	1	1	1	42	8	0	14 (10/0/10)	0 (0/0/0)
nPack v1.1.300	11K	138K	(✓,✓)	1	1	1	41	2	0	21 (14/0/14)	1 (0/0/0)
PE Lock	21K	2.3M	(✓,✓)	1	1	6	53	90	0	4 (3/0/3)	3 (0/1/0)
RLPack	6K	941K	(✓,✓)	1	1	1	21	2	0	14 (8/0/8)	0 (0/0/0)
TELock v0.51	12K	406K	(×,✓)	1	1	5	0	2	0	3 (3/0/3)	1 (0/1/0)
Upack v0.39	4K	711K	(✓,✓)	1	1	2	11	1	0	7 (5/0/5)	1 (0/0/0)
UPX v2.90	5K	62K	(✓,✓)	1	1	1	11	1	0	4 (2/0/2)	0 (0/0/0)

. opaque pred.: bound $k = 16$ – OP: **proved** opaque – Unk: query returns unknown – TO: timeout (5 sec.)
. stack tampering: RT_{ok}: `#ret` runtime genuine - RT_{ko}: `#ret` runtime tampered - a/d/g/s: **proved** aligned/disaligned/genuine/single target
. dynamic information: tr.ok: whether the executed trace was successfully gathered without exception/detection - host: whether the payload was successfully executed - #proc: #process spawned - #th: #threads spawned - #layers: #self-modification layers

not connected to internet but reachable from an internal network. These two malwares are being used as part of target attack campaigns (APT) from the APT28 group also known as Sednit, Fancy Bear, Sofacy or Pawn Storm. This group, active since 2006, targets geopolitical entities and is supposedly highly tight to Russian foreign intelligence. Among alleged attacks, noteworthy targets are NATO [36], EU institutions [37], the White House [38], the German parliaments [39] and more recently the American Democrate National Comittee DNC [40] that affected the running of elections. This group also makes use of many 0-days [41] in Windows, Flash, Office, Java and also operate other malwares like rootkits, bootkits, droppers, Mac 0SX malwares [42] as part of its ecosystem.

Goal. This use-case is based on 3 X-TUNNEL samples[5] covering a 5 month period (according to timestamps). While Sample #0 is not obfuscated and can be straightforwardly analyzed, Samples #1 and #2 are, and they are also much larger than Sample #0 (cf. Table VIII). The main issue here is:

G1: *Are there new functionalities in the obfuscated samples?*

Answering this question requires first to be able to analyse the obfuscated binaries. Hence we focus here on a second goal:

G2: *Recover a de-obfuscated version of Samples #1 and #2.*

We show in the latter how BB-DSE can solve goal G2, and we give hints on what is to be done to solve G1.

Analysis context. Obfuscated samples appeared to contain a tremendous amount of opaque predicates. As a consequence, our goal is to detect and remove all opaque predicates in order to remove the dead-code and meaningless instructions to hopefully obtain a de-obfuscated CFG. This deobfuscation step is a prerequisite for later new functionality finding. The analysis here has to be performed *statically*:

[5]We warmly thank Joan Calvet for providing the samples.

TABLE VIII: Samples infos

	Sample #0 42DEE3[...]	Sample #1 C637E0[...]	Sample #2 99B454[...]
obfuscated	No	Yes	Yes
size	1.1 Mo	2.1 Mo	1.8 Mo
creation date	25/06/2015	02/07/2015	02/11/2015
#functions	3039	3775	3488
#instructions	231907	505008	434143

- as the malware is a network component, it requires to connect to the CC server, which is *truly* not desirable;
- moreover, many branching conditions are network-event based, thus unreliable and more hardly reproducible.

Fortunately, a quick inspection (dynamic run skipping server connexion) confirms that X-TUNNEL does not seem to use any self-modification or neatly tricks to hamper static disassembly. Thus, we proceed as follows: we take the CFG recovered by IDA, and from that we compute the pre^k of each conditional branch (IDASEC). This is a *realistic* reverse scenario when dynamic recovery is not desirable, IDA being the *de facto* static disassembly standard. Correctness of the analysis depends on the quality of the CFG recovered by IDA, so we cannot have *absolute* guarantees. Our goal here is to *improve over state-of-the-practice on a realistic scenario*.

B. Analysis

OP detection. The analysis performs a BB-DSE on every conditional jumps of the program, testing systematically both branches. Taking advantage of previous experiments, we set the bound k to 16. The solver used is `Z3` with a 6s timeout. If both branches are UNSAT, the predicate is considered dead, as the unsatisfiability is necessarily due to path constraints indicating that the predicate is not reachable.

Code simplification. We perform three additional computations in complement to the opaque predicate detection:

- **predicate synthesis** recovers the high-level predicate of an opaque predicate by backtracking on its logical operations. The goal of this analysis is twofold: (1) indexing the different kind of predicates used and (2) identifying instruction involved in the computation of an OP denoted *spurious instructions* (in order to remove them);
- **liveness propagation** based on obfuscation-related data aims at marking instruction by theirs status, namely *alive, dead, spurious*;
- **reduced CFG extraction** extracts the de-obfuscated CFG based on the liveness analysis.

C. Results

Execution time. Table IX reports the execution time of the the BB-DSE and predicate synthesis. The predicate synthesis takes a non-negligible amount of time, yet it is still very affordable, and moreover our implementation is far from optimal.

TABLE IX: Execution time

	#preds	DSE	Synthesis	Total
Sample #1	34505	57m36	48m33	1h46m
Sample #2	30147	50m59	40m54	1h31m

OP diversity. Each sample presents a very low diversity of opaque predicates. Indeed, solely $7x^2 - 1 \neq x^2$ and $\frac{2}{x^2+1} \neq y^2 + 3$ were found. Table X sums up the distribution of the different predicates. The amount of predicates and their distribution supports the idea that they were inserted automatically and picked randomly.

TABLE X: Opaque predicates variety

	$7y^2 - 1 \neq x^2$	$\frac{2}{x^2+1} \neq y^2 + 3$
Sample #1	6016 (49.02%)	6257 (50.98%)
Sample #2	4618 (45.37%)	5560 (54.62%)

Detection results. As the diversity of opaque predicates is very low, we are able to determine, with quite a good precision, the amount of false negatives and false positives based on the predicate synthesized. If a predicates matches one (resp. do not match any) of the two identified opaque predicates and is classified as genuine (resp. opaque), then we considered it a false negative (respectively false positive). Results are given in Table XI and Figure 11. The detection rate is satisfactory, with 3% of false negative and 8.4 to 8.6% of false positive. A few conditions are classified as *unknown*, since both branches are proved infeasible due to some unhandled syscalls.

Dependency evaluation. While the average distance between an opaque predicate and its variable definitions is here 8.7 (less than the bound $k = 16$), the maximum distances are 230 (Sample #1) and 148 (Sample #2). Fortunately, we dot not need all this information to prove infeasibility.

TABLE XI: Opaque predicates evaluation

	#pred	Genuine (syntactic)		OP (syntactic)		Unknown
		Genuine	FN	OP	FP	
Sample #1	34505	17197 (49.8%)	1046 (3.0%)	11973 (34.7%)	2968 (8.6%)	1321 (3.8%)
Sample #2	30147	16148 (53.7%)	914 (3.0%)	9790 (32.5%)	2543 (8.4%)	652 (2.5%)

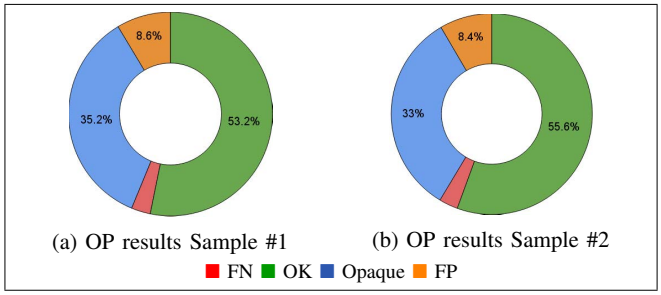


Fig. 11: Graph of opacity distribution

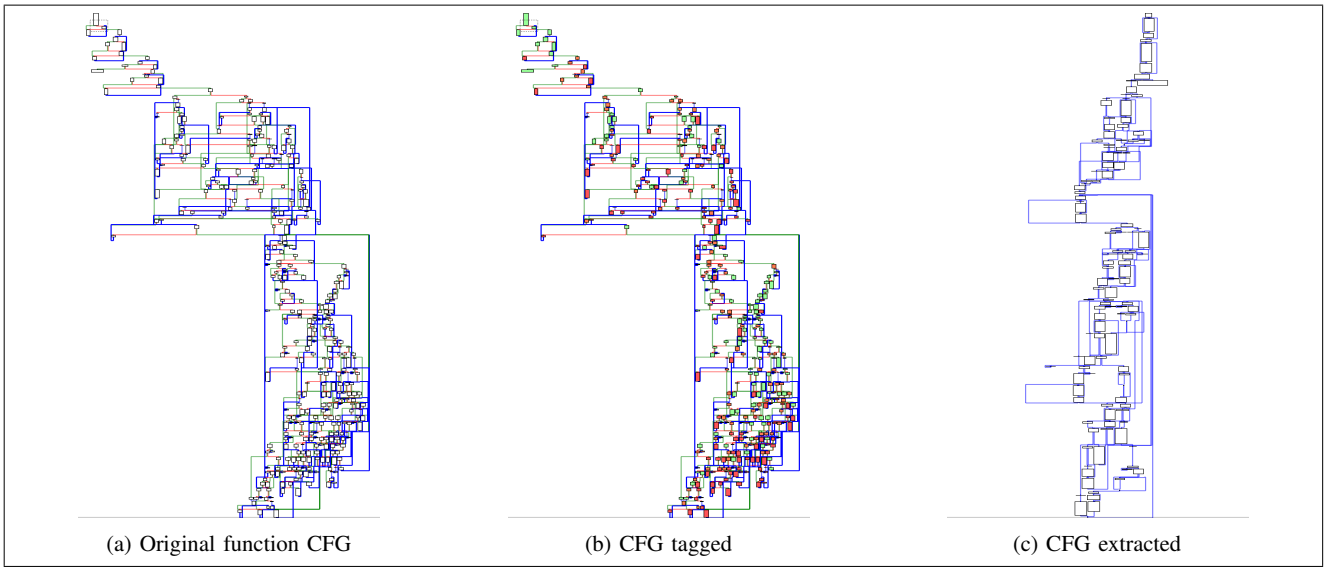
Difference with O-LLVM. Interesting differences with OP found in O-LLVM are to be emphasized. First, there is more interleaving between the payload and the OPs computation. Some meaningful instructions are often encountered within the predicate computation. Second, while O-LLVM OPs are really local to the basic block, there are here some code sharing between predicates, and predicates are not fully independent from one another. Also, the obfuscator uses local function variables to store temporary results at the beginning of the function for later usage in opaque predicates. This increases the depth of the dependency chain and complicates the detection.

Code simplification, Reduced CFG extraction. Table XII shows the number of instructions re-classified based on their status. The dead code represents 1/4 of all program instructions. Computing the difference with the original non-obfuscated program shows a very low difference. Therefore, the simplification pass allowed to retrieve a program which is roughly the size of the original one. The difference is highly likely to be due to the false negatives or missed *spurious* instructions. Finally, Figure 12 shows a function originally (a), with the status tags (b), and the result after extraction (c) using tags (red:dead, orange:spurious, green:alive). Although the CFG extracted still containing noise, it allows a far better understanding of the function behavior. A demo video showing the deobfuscation of a X-TUNNEL function with BINSEC and IDASEC is available as material for this paper[6].

D. Conclusion

About the case-study. We have been able to automatically detect opaque predicates in the two obfuscated samples

[6]https://youtu.be/Z14ab_rzjfA



(a) Original function CFG (b) CFG tagged (c) CFG extracted

Fig. 12: Examples of CFG extraction

TABLE XII: Code simplification results

	#instr	#alive	#dead	#spurious	diff sample #0†
Sample #1	507,206	279,483 (55%)	121,794 (24%)	103,731 (20%)	**47,576**
Sample #2	436,598	241,177 (55%)	113.764 (26%)	79,202 (18%)	**9,270**

† Sample #0: 231,907 instrs

of the X-TUNNEL malware, leading to a significant (and automatic) simplification of these codes – removing all spurious and dead instructions. Moreover, we have gained insights (both strengths and weaknesses) into the inner working of X-TUNNEL protections. Hence, we consider that goal G2 has been largely achieved. In order to answer to the initial question (G1), some similarity algorithms should be computed between the non-obfuscated and simplified samples. This second step is left as future work.

About X-TUNNEL **protections.** The obfuscations found here are quite sophisticated compared with existing opaque predicates found in the state-of-the-art. They successfully manage to spread the data dependency across a function so that some predicates cannot be solved locally at the basic block level. Thankfully, this is not a general practice across predicates so that BB-DSE works very well in the general case. The main issue of the obfuscation scheme is the low diversity of opaque predicates, allowing for example pattern matching techniques to come in relay of symbolic approaches.

IX. APPLICATION: SPARSE DISASSEMBLY

A. Principles

As already explained, static and dynamic disassembly methods tend to have complementary strengths and weaknesses, and BB-DSE is the only robust approach targeting infeasibility questions. Hence, we propose *sparse disassembly*, an algorithm based on recursive disasssembly reinforced with a dynamic trace and complementary information about obfuscation (computed by BB-DSE) in order to provide a more precise disassembly of obfuscated codes. The basic idea is to *enlarge* and initial dynamic disassembly by a cheap syntactic disassembly *in a guaranteed way*, following information from BB-DSE, hence getting the best of dynamic and static approaches.

The approach takes advantage of the two analyses presented in Sections VI-B and VI-C as follows (cf. Figure 13):

- use dynamic values found in the trace to keep disassembling after indirect jump instructions;
- use opaque predicates found by BB-DSE to avoid disassembling dead branches (thus limiting the number of recovered non legit instructions);
- use stack tampering information found by BB-DSE to disassemble the return site of the `call` only in the genuine case, and the real `ret` targets in case of violation.

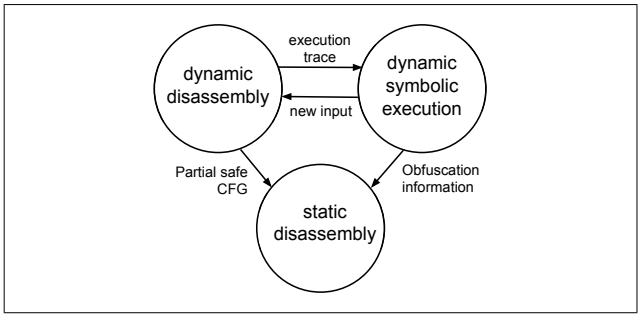


Fig. 13: Sparse disassembly combination

Implementation. A preliminary version of this algorithm has

been integrated in BINSEC, taking advantage of the existing recursive disassembly algorithm. The BB-DSE procedure sends OP and `ret` information to the modified recursive disassembler, which takes the information into account.

B. Preliminary Evaluation

We report two sets of experiments, designed to assess the precision of the approach and its ability to enlarge an initial dynamic trace. We compare our method mainly to the well-known disassembly tools `IDA` and `Objdump`. `IDA` relies on a combination of recursive disassembly, linear sweep and dedicated heuristics. `Objdump` performs only liner sweep.

Precision. In the first evaluation, we compare these different tools on simple programs obfuscated either by O-LLVM (opaque predicates) or Tigress (stack tampering). In each experiment, we compare the set of disassembled instructions with the set of legitimate instructions of the obfuscated program (i.e., those instructions which can be part of a real execution). It turns out on these small examples that all methods are able to find all the legitimate instructions, yet they may be lured into dead instructions introduced by obfuscation.

Tables XIII and XIV present our results. We report for each program and each disassembly method the number of recovered instructions. It turns out that this information is representative of the quality of the disassembly (the less instruction, the better), given the considered obfuscations and the fact that here all methods recover all legitimate instructions (actually, all results have been checked manually).

TABLE XIII: Sparse disassembly opaque predicates

sample	no obf.	Obfuscated				gain vs IDA (sparse)
		perfect	IDA	Objdump	BINSEC **sparse**	
simple-if	37	**185**	240	244	**185**	23,23%
huffman	558	**3226**	3594	3602	**3226**	10,26%
mat_mult	249	**854**	1075	1080	**854**	20,67%
bin_search	105	**833**	1110	1115	**833**	24,95%
bubble_sort	121	**1026**	1531	1537	**1026**	32,98%

TABLE XIV: Sparse disassembly stack tampering

sample	no obf.	Obfuscated				gain vs IDA (sparse)
		perfect	IDA	Objdump	BINSEC **sparse**	
simple-if	37	**83**	95	98	**83**	14.45%
huffman	558	**659**	678	683	**659**	2.80%
mat_mult	249	**461**	524	533	**461**	12.0%
bin_search	105	**207**	231	238	**207**	10.39%
bubble_sort	121	**170**	182	185	**170**	6.6%

In both cases, sparse disassembly achieves a *perfect score* – recovering all but only legitimate instructions, performing better than `IDA` and `Objdump`. Especially, when opaque predicates are considered, sparse disassembly recovers up to 32% less instructions than `IDA`.

Improvement over dynamic analysis. We now seek to assess whether sparse disassembly can indeed *enlarge a dynamic analysis in a significant yet guaranteed way*, i.e., without adding dead instructions. We consider 5 larger `coreutils` programs obfuscated with O-LLVM. We compare sparse disassembly to dynamic analysis (starting from the same trace). The number of recovered instructions is again a good metric of precision (the bigger, the better), since both methods *report only legitimate instructions* on these examples (we checked that BB-DSE was able to find all inserted opaque predicates). Results are reported in Table XV. We also report the output of `IDA` and `Objdump` for the sake of information, yet recall that these tools systematically get fooled by opaque predicates and recover many dead instructions. The important metric here is the *differential between dynamic disassembly and sparse disassembly*. Moreover, note that the absolute coverage of both dynamic and sparse disassembly can naturally be improved using more dynamic traces.

TABLE XV: Sparse disassembly `coreutils`

sample	Tr.len	Obfuscated			
		Objdump	IDA	Dynamic disas.	BINSEC **sparse**
basename	1,783	20,776	20,507	**1,159**	**7,894**
env	3,692	19,714	19,460	**477**	**6,743**
head	17,682	32,840	32,406	**1,299**	**19,807**
mkdir	1,436	57,238	56,767	**1,407**	**10,428**
mv	14,346	115,278	114,067	**5,261**	**81,596**

Actually, these experiments demonstrate that sparse disassembly is an effective way to *enlarge a dynamic disassembly*, in a both *significant and guaranteed manner*. Indeed, sparse disassembly recovers between 6x and 16x more instructions than dynamic disassembly, yet it still recovers much less than linear sweep – due to the focused approach of dynamic disassembly and the guidance of BB-DSE. Hence, sparse disassembly stays close to the original trace.

Conclusion. The carried experiments showed very good and accurate results on controlled samples, achieving perfect disassembly. From this stand-point, sparse disassembly performs better than combination of both recursive and linear like in `IDA`, with up to 30% less recovered instructions than `IDA`. The `coreutils` experiments showed that sparse disassembly is also an effective way to enlarge a dynamic disassembly in a both significant and guaranteed manner. In the end, this is a clear demonstration of infeasibility-based information used in the context of disassembly.

Yet, our sparse disassembly algorithm is still very preliminary. It is currently limited by the inherent weaknesses of recursive disassembly (rather than sparse disassembly shortcomings), for example the handling of computed jumps would require advanced pattern techniques.

X. DISCUSSION: SECURITY ANALYSIS

From the attacker point of view, three main counter-measures can be employed to hinder our approach. We present them as well as some possible mitigation.

The first counter-measure is to artificially spread the computation of the obfuscation scheme over a long sequence of code, hoping either to evade the "k" bound of the analysis (false negatives) or to force a too high value for k (false positives or timeouts). Nevertheless, it is often not necessary to backtrack all the dependencies to prove infeasibility. An example is given in X-TUNNEL were many predicates have a dependency chain longer than the chosen bound (k=16, chain up to 230) but this value was most of the time sufficient to gather enough constraints to prove predicate opacity. Moreover, a very good mitigation for these "predicates with far dependencies" is to rely on a more generic notion of the k bound, based for example on def-use chain length or some formula complexity criterias rather than a strict number of instructions.

The second counter-measure is to introduce hard-to-solve predicates (based for example on Mixed-Boolean Arithmetic [43] or cryptographic hashing functions) in order to lead to inconclusive solver responses (timeout). As we cannot directly influence the solving mechanism of SMT solvers, there is no clear mitigation from the defender perspective. Nonetheless, solving such hard formula is an active research topic and some progress can be expected in a middle-term on particular forms of formulas [44]. Moreover, certain simplifications typically used in symbolic execution (e.g., constant propagation or tainting) already allow to bypass simple cases of *a priori* difficult-to-solve predicates Additionally, *triggering a timeout is already a valuable information*, since BB-DSE with reasonable k bound usually does not timeout. The defender can take advantage of it by manually inspecting the timeout root cause and deduce *(in-)feasible patterns*, which can now be detected through mere syntactic matching. In the same vein, timeout may pinpoint to the reverser the most important parts of the code, unless hard predicates are used everywhere, with a possibly very significant runtime overhead. Finally, such counter-measures would greatly complicate the malware design (and its cost!) and a careless insertion of such complex patterns could lead to atypical code structures prone to relevant malware signatures.

Actually, our experiments show that symbolic methods are quite efficient for deobfuscation. Yet, it is clear that dedicated protections could be used, and indeed such anti-DSE protections have been recently proposed [45], [10]. We are in the middle of a cat-and-mouse game, and our objective is to push it further in order to significantly raise the bar for malware creators.

The third counter-measure is to add anti-dynamic tricks, in order to evade the first step of dynamic disassembly. Yet, since our technique works with any tracer technology, the dynamic instrumentation can be strengthened with appropriate mitigations. Interestingly, certain dynamic tricks can be easily mitigated in a symbolic setting, e.g., detection based on timing can be defeated by symbolizing adequat syscalls.

XI. RELATED WORK

DSE and deobfuscation. Dynamic Symbolic Execution has been used in multiple situations to address obfuscation, generally for discovering new paths in the code to analyze. Recently, Debray at al. [10], [11] used DSE against conditional and indirect jumps, VM and return-oriented programming on various packers and malware in order to prune the obfuscation from the CFG. Mizuhito *et al.* also addressed exception-based obfuscation using such techniques [46]. Recent work from Ming *et al.* [12] used (forward) DSE to detect different classes of opaque predicates. Yet, their technique has difficulties to scale due to the trace length (this is consistent with experiments in Section VI-A). Indeed, by doing it in a forward manner they needlessly have to deal with the whole path predicate for each predicate to check. As consequence they make use of taint to counterbalance which far from being perfect brings additional problems (under-tainting/over-tainting).

DSE is designed to prove the reachability of certain parts of code (such as path, branches or instructions). It is complementary to BB-DSE in that it addresses feasibility queries rather than infeasibility queries. Moreover, BB-DSE scales very well, since it does not depend on the trace length but on the user-defined parameter k. Thus, while backward-bounded DSE seems to be the most appropriate way to solve infeasibility problems no researches have used this technique.

Backward reasoning. Backward reasoning is well-known in infinite-state model checking, for example for Petri Nets [47]. It is less developed in formal software verification, where forward approaches are prevalent, at the notable exception of deductive verification based on weakest precondition calculi [18]. Interestingly, Charreteur *et al.* have proposed (unbounded) backward symbolic execution for goal-oriented testing [48]. Forward and backward approaches are well-known to be complementary, and can often be combined with benefit [49].

Yet, purely backward approaches seem nearly impossible to implement at binary level, because of the lack of *a priori* information on computed jumps. We solve this problem in BB-DSE by performing backward reasoning along some dynamic execution paths observed at runtime, yet at the price of (a low-rate of) false positives.

Disassembly. Standard disassembly techniques have already been discussed in Section IX. Advanced static techniques include recursive-like approaches extended with patterns dedicated to difficult constructs [2]. Advanced dynamic techniques take advantage of DSE in order to discover more parts of the code [14], [28]. Binary-level semantic program analysis methods [15], [16], [17], [13], [50] does allow in principle a guaranteed exhaustive disassembly. Even if some interesting case-studies have been conducted, these methods still face big issues in terms of scaling and robustness. Especially, self-modification is very hard to deal with. The domain is recent, and only very few work exist in that direction [51], [52]. Several works attempt to combine static analysis and dynamic analysis in order to get better disassembly. Especially, CODISASM [3] take advantage of the dynamic trace to perform syntactic static disassembly of self-modifying programs.

Again, our method is complementary to all these approaches which are mainly based on forward reasoning [53].

Obfuscations. Opaque predicates were introduced by Collberg [4] giving a detailed theoretical description and possible usages [54], [55] like watermarking. In order to detect them various methods have been proposed [56], notably by abstract interpretation [52] and in recent work with DSE [12]. Issues raised by stack tampering and most notably non-returning functions are discussed by Miller [2]. Lakhotia [6] proposes a method based on abstract interpretation [6]. None of the above solutions address the problem in such a scalable and robust way as BB-DSE does.

XII. Conclusion

Many problems arising during the reverse of obfuscated codes come down to solve infeasibility questions. Yet, this class of problem is mostly a blind spot of both standard and advanced disassembly tools. We propose Backward-Bounded DSE, a precise, efficient, robust and generic method for solving infeasibility questions related to deobfuscation. We have demonstrated the benefit of the method for several realistic classes of obfuscations such as opaque predicate and call stack tampering, and given insights for other protection schemes. Backward-Bounded DSE does not supersede existing disassembly approaches, but rather complements them by addressing infeasibility questions. Following this line, we showed how these techniques can be used to address state-sponsored malware (X-TUNNEL) and how to merge the technique with standard static disassembly and dynamic analysis, in order to enlarge a dynamic analysis in a precise and guaranteed way. This work paves the way for precise, efficient and disassembly tools for obfuscated binaries.

References

[1] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.

[2] B. P. Miller and X. Meng, "Binary code is not easy," in *ISSTA 2016*. ACM, 2016.

[3] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "Codisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *CCS 2015*. ACM, 2015.

[4] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *POPL 1998*. ACM, 1998. [Online]. Available: http://doi.acm.org/10.1145/268946.268962

[5] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *ACSAC 2007*, Dec 2007.

[6] A. Lakhotia, E. U. Kumar, and M. Venable, "A Method for Detecting Obfuscated Calls in Malicious Binaries," *IEEE Trans. Softw. Eng.*, vol. 31, no. 11, Nov. 2005.

[7] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Comput. Surv.*, vol. 46, no. 1, Jul. 2013.

[8] P. Godefroid, M. Y. Levin, and D. A. Molnar, "SAGE: whitebox fuzzing for security testing," *Commun. ACM*, vol. 55, no. 3, 2012. [Online]. Available: http://doi.acm.org/10.1145/2093548.2093564

[9] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, no. 2, 2013. [Online]. Available: http://doi.acm.org/10.1145/2408776.2408795

[10] B. Yadegari and S. Debray, "Symbolic execution of obfuscated code," in *CCS 2015*. ACM, 2015.

[11] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *SP 2015*, May 2015.

[12] J. Ming, D. Xu, L. Wang, and D. Wu, "Loop: Logic-oriented opaque predicate detection in obfuscated binary code," in *CCS 2015*. ACM, 2015.

[13] S. Bardin, P. Herrmann, and F. Védrine, "Refinement- based CFG reconstruction from unstructured programs," in *VMCAI 2011*, 2011.

[14] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and D. Song, "BitScope: Automatically dissecting malicious binaries," School of Computer Science, Carnegie Mellon University, Tech. Rep. CS-07-133, Mar. 2007.

[15] G. Balakrishnan and T. W. Reps, "WYSINWYX: what you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, 2010.

[16] J. Kinder and H. Veith, "Precise static analysis of untrusted driver binaries," in *FMCAD 2010*. Springer, 2010.

[17] A. Sepp, B. Mihaila, and A. Simon, "Precise static analysis of binaries by extracting relational information," in *18th Working Conference on Reverse Engineering, WCRE 2011*. IEEE, 2011. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2011.50

[18] K. R. M. Leino, "Efficient weakest preconditions," *Inf. Process. Lett.*, vol. 93, no. 6, 2005.

[19] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without bdds," in *TACAS 1999*. Springer, 1999.

[20] A. Djoudi and S. Bardin, "Binsec: Binary code analysis with low-level regions," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015.

[21] R. David, S. Bardin, T. Thanh Dinh, J. Feist, L. Mounier, M.-L. Potet, and J.-Y. Marion, "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis," in *SANER 2016*. IEEE, 2016.

[22] J. Calvet, J. Campos, and T. Dupuy, "Visiting The Bear Den, A Journey in the Land of (Cyber-)Espionage," RECON 2016, Montreal, 17/06/16.

[23] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm: Software protection for the masses," in *SPRO 2015*. IEEE Press, 2015.

[24] J. Vanegue and S. Heelan, "SMT solvers in software security," in *WOOT 2012*. Usenix Association, 2012, pp. 85–96. [Online]. Available: http://www.usenix.org/conference/woot12/smt-solvers-software-security

[25] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, vol. 40, no. 6, 2005.

[26] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for C," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, 2005.

[27] R. David, S. Bardin, J. Feist, J.-Y. Marion, L. Mounier, M.-L. Potet, and T. D. Ta, "Specification of concretization and symbolization policies in symbolic execution," in *ISSTA 2016*. ACM, July 2016.

[28] S. Bardin and P. Herrmann, "OSMOSE: automatic structural testing of executables," *Softw. Test., Verif. Reliab.*, vol. 21, no. 1, 2011.

[29] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, Feb. 2012.

[30] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *SP 2012*. IEEE, 2012.

[31] M. D. Preda, R. Giacobazzi, S. K. Debray, K. Coogan, and G. M. Townsend, "Modelling metamorphism by abstract interpretation," in *SAS 2010*. Springer, 2010.

[32] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent, "The Bincoa Framework for Binary Code Analysis," in *CAV 2011*, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22110-1_13

[33] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *SP 2014*, May 2014.

[34] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *SP 2015*, 2015. [Online]. Available: http://dx.doi.org/10.1109/SP.2015.46

[35] C. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *ACSAC 2012*. ACM, 2012.

[36] Trend Micro, "Operation Pawn Storm, Using Decoys to Evade Detection," Tech. Rep., 2014.

[37] ESET Research, "Sednit APT Group Meets Hacking Team," http://www.welivesecurity.com/2015/07/10/sednit-apt-group-meets-hacking-team/, Oct. 2015.

[38] Trend Micro, "Operation Pawn Storm Ramps Up its Activities; Targets NATO, White House," Apr. 2015.

[39] von Gastbeitrag, "Digital Attack on German Parliament: Investigative Report on the Hack of the Left Party Infrastructure in Bundestag," Jun. 2015.

[40] D. Alperovitch, "Bears in the Midst: Intrusion into the Democratic National Committee," https://www.crowdstrike.com/blog/bears-midst-intrusion-democratic-national-committee/, Jun. 2016.
[41] N. Mehta and B. Leonard, "CVE-2016-7855: Chromium Win32k system call lockdown," Tech. Rep., 2016.
[42] D. Creus, T. Halfpop, and R. Falcone, "Sofacy's 'Komplex' OS X Trojan," http://researchcenter.paloaltonetworks.com/2016/09/unit42-sofacys-komplex-os-x-trojan/, Sep. 2016.
[43] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information Hiding in Software with Mixed Boolean-Arithmetic Transforms," in *Information Security Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4867, pp. 61–75.
[44] N. Eyrolles, L. Goubin, and M. Videau, "Defeating mba-based obfuscation," in *SPRO 2016 (CCS workshop)*, ACM, Ed., 2016.
[45] S. Banescu, C. S. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *ACSAC 2016*. ACM, 2016.
[46] N. M. Hai, M. Ogawa, and Q. T. Tho, *Foundations and Practice of Security: 8th International Symposium, FPS 2015, Revised Selected Papers*. Springer, 2016, ch. Obfuscation Code Localization Based on CFG Generation of Malware. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-30303-1_14
[47] A. Finkel and P. Schnoebelen, "Well-structured transition systems everywhere!" *Theor. Comput. Sci.*, vol. 256, no. 1-2, 2001.
[48] F. Charreteur and A. Gotlieb, "Constraint-based test input generation for java bytecode," in *ISSRE 2010*. IEEE, 2010.
[49] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. L. Traon, and J. Marion, "Sound and quasi-complete detection of infeasible test requirements," in *ICST 2015*. IEEE, 2015.
[50] T. Reinbacher and J. Brauer, "Precise control flow reconstruction using boolean logic," in *EMSOFT 2011*. ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2038642.2038662
[51] S. Blazy, V. Laporte, and D. Pichardie, "Verified abstract interpretation techniques for disassembling low-level self-modifying code," in *ITP 2014*. Springer, 2014.
[52] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi, "Opaque predicates detection by abstract interpretation," in *AMAST 2006*. Springer-Verlag, 2006. [Online]. Available: http://dx.doi.org/10.1007/11784180_9
[53] M. H. Nguyen, T. B. Nguyen, T. T. Quan, and M. Ogawa, "A hybrid approach for control flow graph construction from binary code," in *APSEC 2013*, vol. 2, Dec 2013.
[54] G. Myles and C. Collberg, "Software watermarking via opaque predicates: Implementation, analysis, and attacks," *Electronic Commerce Research*, vol. 6, no. 2, 2006. [Online]. Available: http://dx.doi.org/10.1007/s10660-006-6955-z
[55] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang, "Experience with software watermarking," in *ACSAC 2000*, 2000. [Online]. Available: http://dx.doi.org/10.1109/ACSAC.2000.898885
[56] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *WCRE 2005*, 2005.

APPENDIX

(Section VI-B, extended). Figure 14 shows a graphical representation of results from Table V. The x-axis represents the value of the bound k, and the y-axis represents the numbers of predicates identified as opaque, genuine, plus the number of timeouts (TO), false positive (FP) and false negative (FN). When k increases, #FN strongly decreases while #FP slowly increases. Here, #TO is kept very low.

(Section VII-B, extended) Findings on call/stack tampering. From the call/stack tampering perspective and according to the taxonomy defined in Section V, many different kinds of violations were detected. The first two patterns found in ACProtect shown in Figures 15 and 16 are respectively

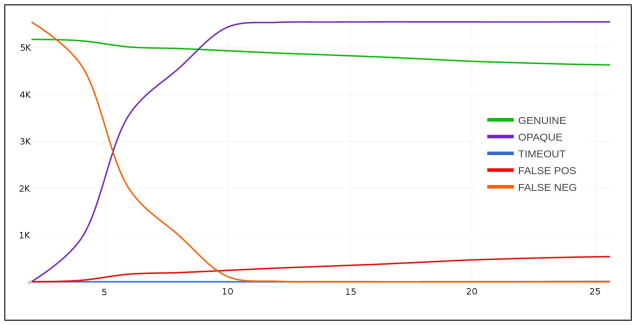


Fig. 14: OP detection: tradeoff between k, FN and FP

detected as `[violated], [single], [aligned]` and `[violated], [single], [disaligned]`. Figures 18, 17 and 19 show three different kinds of violation found in `ASPack`. In the first example (cf. Figure 18) the tampering is detected with labels `[violated], [disaligned]` since the stack pointer read the `ret` address at the wrong offset. In the second example (cf. Figure 17), the return value is modified in place. The tampering is detected with the `[violated], [aligned], [single]` tags. The last example (cf. Figure 19), takes place between the transition of two self-modification layers and the ret is used for tail-transitioning to the packer payload (i.e., the original unpacked program). This violation is detected with `[violated], [disaligned], [single]` since the analysis matches a `call` far upper in the trace which is disaligned. Note that instruction `push 0x10011d7` at address 10043ba is originally a `push 0`, but it is patched by instruction at address 10043a9, triggering the entrance in a new auto-modification layer when executing it. This pattern reflects a broader phenomenon found in many packers like `nPack`, `TELock` or `Upack` having a single `ret` tampered: these packers perform their tail transition to the entrypoint of the original (packed) program with `push; ret`. Thus, *such analysis allows to find precisely that moment in the execution trace, where the payload is very likely decompressed in memory*.

address	mnemonic	comment
1004328	call 0x1004318	//push 0x100432d as return
1004318	add [esp], 9	//tamper the value in place
100431c	ret	//return to 0x1004n336

Fig. 15: ACProtect violation 1/2

address	mnemonic	comment
1001000	push 0x1004000	
1001005	push 0x100100b	
100100a	ret	jump on the ret below
100100b	ret	jump on 0x1004000

Fig. 16: ACProtect violation 2/2

address	len	mnemonic	comment
1004a3a	5	call 0x1004c96	//push 0x1004a3f as return site
1004c96	5	call 0x1004c9c	//push 0x1004c9b as return site
1004c9c	1	pop esi	//pop return address in esi
1004c9d	5	sub esi, 4474311	
1004ca3	1	ret	//return to 0x1004a3f

Fig. 17: ASPack violation 1/3

address	mnemonic	comment
1004002	call 0x100400a	//push 0x1004007 as return
1004007	.byteinvalid	//invalid byte (cannot disassemble)
1004008	[...]	//not disassembled
100400a	pop ebp	//pop return address in ebp
100400b	inc ebp	//increment ebp
100400c	push ebp	//push back the value
100400d	ret	//jump on 0x1004008

Fig. 18: ASPack violation 2/3

address	mnemonic	layer	comment
10043a9	mov [ebp+0x3a8], eax	0	//Patch push value at 10043ba*
10043af	popa	0	//restore initial program context
10043b0	jnz 0x10043ba	0	//enter last SM layer (payload)
	Enter SMC Layer 1		
10043ba	push 0x10011d7	1	//push the address of the entrypoint
10043bf	ret	0	//use ret to jump on it
10011d7	[...]	1	//start executing payload

*(at runtime eax=10011d7 and ebp+0x3a8=10043bb)

Fig. 19: ASPack violation 3/3

(Section VII-A, extended) Detailed packer experiments. Table XVI presents a complete view of the experiments presented in Table VII.

TABLE XVI: Packer experiment: Opaque Predicates & Call stack tampering

Packers	Static	Dynamic					Obfuscation detection					
	size					self-mod.	Opaque Predicates (k_{16})				Stack tampering	
	prog	#tr.len	(tr.ok/host)	#proc	#th	#layers	OK	OP	To	Covered	OK (a/d)	Viol (a/d/s)
ACProtect v2.0	101K	1.813.598	(✓,×)	1	1	4	74	159	0	9	0 (0/0)	48 (45/1/45)
Armadillo v3.78	460K	150.014	(×,×)	2	11	1	1	20	0	1	2 (2/0)	0 (0/0/0)
Aspack v2.12	10K	377.349	(✓,✓)	1	1	2	32	24	0	136	11 (7/0)	6 (1/4/1)
BoxedApp v3.2	903K	/	(×,×)*	1	15	-	-	-	-	-	-	-
Crypter v1.12	45K	1.170.108	(✓,×)	1	1	0	263	24	0	136	125 (94/0)	78 (0/30/32)
Enigma v3.1	1,1M	10.000.000	(×,×)†	-	-	1	-	-	-	-	-	-
EP Protector v0.3	8,6K	250	(✓,✓)	1	1	1	10	1	0	2	4 (2/0)	0 (0/0/0)
Expressor	13K	635.356	(✓,✓)	1	1	1	42	8	0	39	14 (10/0)	0 (0/0/0)
FSG v2.0	3,9K	68.987	(✓,✓)	1	1	1	11	1	0	14	6 (4/0)	0 (0/0/0)
JD Pack v2.0	53K	42	(×,✓)	1	1	0	2	0	0	0	0 (0/0)	0 (0/0/0)
Mew	2,8K	59.320	(✓,✓)	-	-	1	11	1	0	18	6 (4/0)	1 (0/0/0)
MoleBox	70K	5.288.567	(✓,✓)‡	1	1	2	307	60	0	128	X	X
Mystic	50K	4.569.154	(✓,✓)‡	1	1	1	X	X	X	X	X	X
Neolite v2.0	14K	42.335	(✓,✓)	1	1	1	95	1	0	42	9 (3/0)	0 (0/0/0)
nPack v1.1.300	11K	138.231	(✓,✓)	1	1	1	41	2	0	34	21 (14/0)	1 (0/0/0)
Obsidium v1364	116K	21	(×,✓)	-	-	0	1	0	0	0	0 (0/0)	0 (0/0/0)
Packman v1.0	5,9K	130.174	(✓,✓)	1	1	1	12	1	0	21	7 (4/0)	0 (0/0/0)
PE Compact v2.20	7,0K	202	(✓,✓)	1	1	1	11	1	0	1	4 (2/0)	0 (0/0/0)
PE Lock	21K	2.389.260	(✓,✓)	1	1	6	53	90	0	42	4 (3/0)	3 (0/1/0)
PE Spin v1.1	26K	/	(×,×)*	1	1	-	-	-	-	-	-	-
Petite v2.2	12K	260.025	(×,×)	1	1	0	60	19	0	45	4 (1/0)	0 (0/0/0)
RLPack	6,4K	941.291	(✓,✓)	1	1	1	21	2	0	25	14 (8/0)	0 (0/0/0)
Setisoft v2.7.1	378K	4.040.403	(×,×)‡	1	5	4	X	X	X	X	X	X
svk 1.43	137K	10.000.000	(×,✓)†	-	-	0	-	-	-	-	-	-
TELock v0.51	12K	406.580	(×,✓)	1	1	5	0	2	0	5	3 (3/0)	1 (0/1/0)
Themida v1.8	1,2M	10.000.000	(×,✓)†	1	28	0	-	-	-	-	-	-
Upack v0.39	4,1K	711.447	(✓,✓)	1	1	2	11	1	0	30	7 (5/0)	1 (0/0/0)
UPX v2.90	5,5K	62.091	(✓,✓)	1	1	1	11	1	0	26	4 (2/0)	0 (0/0/0)
VM Protect v1.50	13K	/	(×,✓)*	1	1	0	-	-	-	-	-	-
WinUPack	4,0K	657.473	(✓,✓)	1	1	2	12	1	0	33	7 (5/0)	1 (0/0/0)
Yoda's Crypter v1.3	12K	240.900	(×,✓)	1	1	3	38	1	0	16	4 (3/0)	9 (0/1/0)
Yoda's Protector v1.02	18K	17	(×,✓)	1	1	0	1	0	0	0	0 (0/0)	0 (0/0/0)

- **size prog**: size of the program
- **#tr.len**: execution trace length
- **tr.ok**: whether the executed trace was successfully gathered without exception/detection
- **host**: whether the payload was successfully executed *(printing the hostname of the machine)*
- **#proc**: number of process spawned
- **#th**: number of threads spawned
- **#layers**: number of self-modification layers recorded
- **OK, OP, To, Covered**: predicate ok, opaque predicate, timeout, predicate fully covered (both branches)
- **(a/d/s)**: (aligned/disaligned/single)
- $*$ failed to record the trace
- † maximum trace length reached *(thus packer not analyzed)*
- ‡ analysis failed (due to lack of memory)