International Journal of Information Security (2021) 20:73–101
https://doi.org/10.1007/s10207-020-00494-8

REGULAR CONTRIBUTION

Obfuscated integration of software protections

Jens Van den Broeck[1] · Bart Coppens[1] · Bjorn De Sutter[1]

Published online: 18 March 2020
© Springer-Verlag GmbH Germany, part of Springer Nature 2020

Abstract

To counter man-at-the-end attacks such as reverse engineering and tampering, software is often protected with techniques that require support modules to be linked into the application. It is well known, however, that attackers can exploit the modular nature of applications and their protections to speed up the identification and comprehension process of the relevant code, the assets, and the applied protections. To counter that exploitation of modularity at different levels of granularity, the boundaries between the modules in the program need to be obfuscated. We propose to do so by combining three cross-boundary protection techniques that thwart the disassembly process and in particular the reconstruction of functions: code layout randomization, interprocedurally coupled opaque predicates, and code factoring with intraprocedural control flow idioms. By means of an experimental evaluation on realistic use cases and state-of-the-art tools, we demonstrate our technique's potency and resilience to advanced attacks. All relevant code is publicly available online.

Keywords Man-at-the-end attacks · Control flow graph reconstruction · Reverse engineering · Resilience · Potency

1 Introduction

Software protection techniques such as code obfuscation and remote attestation aim to mitigate man-at-the-end (MATE) attacks that target software assets that come with confidentiality and integrity requirements.

The protections typically do not aim to prevent attacks completely. Because MATE attackers have white-box access to the software in their laboratories, the protections aim to raise the costs of (i) identifying successful attack vectors in the attacker's laboratory, and (ii) scaling up the attacks to exploit them outside the laboratory. The protections are in many cases best-effort rather than providing well-defined security, and part of their protection comes from security through obscurity. In practice, their effectiveness decreases when attackers gain more knowledge about their inner workings.

✉ Bjorn De Sutter
bjorn.desutter@ugent.be

Jens Van den Broeck
jens.vandenbroeck@ugent.be

Bart Coppens
bart.coppens@ugent.be

1 Department of Electronics and Information Systems, Ghent University, Technologiepark-Zwijnaarde 126, 9052 Zwijnaarde, Belgium

To be effective, protections should provide resistance against many of the possible methods with which they can be overcome, worked around, bypassed, and undone [1]. Multiple protections defending against different attack methods hence need to be layered upon each other, ideally to the point where attackers consider the attack path of least resistance not profitable enough to attack the software.

Advanced protections, such as code mobility [2], barrier-slicing with server-side execution [3], remote attestation [4], anti-debugging by self-debugging [5], and instruction set randomization [6], are deployed by means of two forms of adaptations to that software. First, components implementing functionality of the protections are linked into the software. Secondly, the original code is transformed.

To delay an attacker in overcoming protections, it is useful to embed the linked-in protection components stealthily, meaning hard to identify. For that reason, protections components are always linked statically into native software to be protected, whether that software is itself a main binary or a dynamically linked library. Static linking does not offer very strong protection, however. In experiments with both professional penetration testers and amateur hackers [1], we have observed that once attackers identify a small part of a statically linked-in protection, they can all too easily expand their reverse engineering to the most vulnerable program points. Beyond static linking, a number of design obfuscations (such

as function merging, inlining, and outlining) are available to obfuscate the design of the interfaces between the application and the protection components, but those can only be deployed when all source code is available. In practice, this is not the case: Vendors of protection tools do not make the source code of their protections available to their customers, because of the practical security-by-obscurity reasons. There hence exist few practical techniques to obfuscate how protection components are integrated into the software they help to protect. While some security vendors have post-processing tools to secure their components after they are integrated into their customers' software, both the inner workings of those tools and their effectiveness are tightly protected secrets.

In this paper, we present novel techniques and combine them with adaptations of existing techniques to hide the location and boundaries of software components that are linked together, including linked-in protection components, with the goal of hampering MATE attacks.

All our techniques are based on post-link-time binary rewriting. It hence does not suffer from module boundaries and separate compilation the way compile-time or source code techniques do. It offers an additional advantage over source-to-source code rewriting, as our techniques are not limited to the expressiveness of the used source language(s).

Our combined techniques are (1) whole-program code layout randomization, (2) insertion of fake direct control flow transfers between procedures, and (3) factorization of code fragments common to multiple components without embedding them in separate functions. Together, they make it much harder for attackers and their tools to identify and structure the relevant code and the control flow in the program. Moreover, as our evaluation will demonstrate, the combination of techniques is resilient against a number of commonly used and academic state-of-the-art manual and automated deobfuscation techniques.

This paper offers the following main contributions:

- We present new forms of code factoring to serve as module boundary obfuscations.
- We discuss how to combine them with code layout randomization and (existing) opaque predicates to resist automated and manual attacks.
- We present an extended open-source tool chain that implements the presented techniques.
- We analyze and evaluate the presented techniques on use cases of real-world complexity, using popular tools in attacker tool boxes.

This paper is structured as follows. Section 2 discusses our attack model. Sections 4–6 discuss the three forms of obfuscations we combine. Section 7 presents an elaborate quantitative experimental evaluation including an extensive sensitivity analysis, after which Sect. 8 discusses related work and Sect. 9 draws conclusions and looks forward.

2 Attack model

We protect native software from man-at-the-end (MATE) attacks. MATE attackers have full access to, and full control over, the software under attack and over the end systems on which the software runs. They can use static analysis tools, emulators, debuggers, and all kinds of other hacking tools. The attacks are looking to break integrity and confidentiality requirements of assets embedded in the software, e.g., to steal keys or IP, or to break license checks and anti-copy protections. They do so mainly by means of reverse engineering and by tampering with the code and its execution.

MATE protections mostly aim at economically driven attackers [7]. They are considered effective when the provider's cost of deploying the protections is compensated by a resulting reduction in the loss of income due to successful attacks. This reduction can result simply from delaying attacks. The protection is maximally effective if it stops attackers before they reach their goal, or even before they start an attack, e.g., because the (supposed or observed) presence of protection lowers the attackers' perceived return-on-investment to the extent that they give up.

MATE attackers execute an attack strategy in which they execute a series of attack steps. The strategy is adapted on the fly, based on the results obtained with previous attack steps. These include the testing of hypotheses regarding assets and protections. We refer to the literature for more information on models of MATE attack processes on protected software [1].

To be effective, protections deployed on software and assets should cover as many as possible relevant attack paths, i.e., paths that might be paths-of-least-resistance for certain attackers. It is commonly accepted that this can only be achieved by combining many protections in a layered fashion. The deployed protections then become assets themselves, which protect each other just like they protect the original assets.

In this section, we focus on the attack processes and attack activities that are impacted by the protections presented in this paper. These are the essential processes of:

- identifying and structuring the code components and their functionality at different levels of granularity and abstraction;
- identifying relevant relations between components;
- determining their features based on the relations;
- browsing through those elements to locate and identify the relevant fragments on which to execute additional attack steps.

Attackers use tools, techniques, and heuristics to build structured program representations such as control flow graphs (CFGs), call graphs, execution traces, data-dependency graphs, etc. of disassembled binary code. For static attack activities, i.e., activities that do no involve executing the code, most attackers rely on disassemblers such as IDA Pro, Binary Ninja, GHIDRA, Radare2, and DynInst to start their attacks. The disassemblers lift the representation of the binary program from the concrete level of bits to a more abstract level of assembler code structured in functions, CFGs, and call graphs.

They then build mental models of the software in which they assign meaning (i.e., some higher-level semantics) to the different components (typically the functions) and derive relevant features thereof. They do so in terms of all the concepts they know as relevant from past experience [1].

This assignment process and the derivation of features are typically an iterative process that starts from easily identified elements such as API calls and system calls, XOR-operations, references to strings, known patterns or fingerprints of certain algorithms, etc. That leads the attacker toward the specific components of interest, such as the data or code he wants to lift from the software, or those parts of protections he wants to tamper with to overcome the protections. Table 1 lists some of the relations between components that attackers exploit.

Table 1 Relations between structured software components and uses thereof by attackers

Relation exploited by attackers	Examples of exploitation in concrete attack
Control flow transfers	Disassemblers such as IDA Pro deploy recursive-descent algorithms to identify code bytes to be disassembled and to be partitioned into functions
Data flow dependencies	If an attacker has observed that values are XOR'ed before they are output, he often assumes they are being encrypted. The code that produces the mask used in the XOR then draws the attention of the attacker if he is after the embedded encryption key
Spatial proximity of code fragments in code sections	If an attacker has identified a code guard function, e.g., because it reads from the code sections as it hashes the code bytes, he looks in the proximity of that function for other functionality related to tamper detection, such as the functions that check the final hash value. This is based on the assumption that related functionality is linked into the program together
Temporal proximity of code fragments in an execution trace	When an attacker tampers with the code of a program, and as a reaction the program halts almost immediately, the attacker will focus on the code executed right before the halting to find the code fragment that checks the integrity of the code
Spatial proximity of data stored in memory	When attackers know that structs on the heap hold values with known patterns as well as unknown values they want to steal, they search for the known patterns to find the locations of the values to steal

3 Protection strategy rationale

It is clear that if we can prevent tools from correctly identifying the relevant relations and structures, we can make the attacks harder to execute. From conversations with professional reverse engineers at Dagstuhl Seminar 17281 in July 2017, we also learned that if tools present incorrect relations and structure, this hampers attackers even more because they then waste additional time performing activities based on incorrect assumptions and data.

In this paper, we target the disassemblers that attackers rely on as discussed in the attack model. From the field of software engineering, we know that code comprehension benefits from well-structuredness of the code [8,9] and a separation of concerns, with each fragment having a single responsibility. It then follows that attackers have a harder time comprehending code that does not adhere to structures and concepts they are familiar with, or that is structured incorrectly. In this paper, we build on the hypothesis that attackers have a harder time handling code fragments that each individually implement multiple parts of multiple, unrelated high-level functions in a program, in particular when those code fragments are not structured correspondingly.

3.1 Rationale for code factoring

Concretely, consider the procedures in a program. Attackers recognize procedures by their prologues and epilogues, and by the fact that they are invoked through function calls. It is a natural assumption that procedures can be invoked from within different contexts. As long as the semantics of the function in the multiple contexts are somewhat related, i.e., it performs roughly the same functionality in those contexts, the process of assigning a meaning to the function can require little effort.

It becomes much harder, however, to comprehend a code if a function implements multiple completely unrelated functionalities, depending on the context from which they are called. This is exploited by obfuscation techniques called function merging and fusion [7]. The fused function is then invoked from completely unrelated contexts, to perform completely unrelated computations, i.e., to implement very different semantics.

Comprehending the code becomes even harder if the code fragment that implements those different functionalities in different contexts is not even recognizable as such, i.e., if it does not look like a procedure in the first place. In software obfuscation, it is also a well-known technique to hide calls, returns, epilogues and prologues by replacing their standard assembler idioms by alternative instruction sequences with the same semantics but with different looks [10]. This thwarts disassemblers that do not recognize the replacements, and it slows down human reverse engineers.

The obfuscations proposed in this work explicitly build on this observation about the challenges that human attackers face when they try to attack and reverse engineer software. The obfuscations do so by factoring out code (outlining code) from unrelated contexts without putting the factored code in separate procedures, instead using control flow idioms typically used for intraprocedural transfers.

Not only humans are challenged when facing such factored out code fragments. Automated attack steps, such as de-obfuscating transformations and data flow analysis on which attackers rely, are also hampered.

First, it is well known that many data flow analyses return more precise results when their sensitivity is improved. Higher sensitivity, e.g., in the form of flow sensitivity, path sensitivity, or context sensitivity comes at the cost of rapidly increasing running times and resource consumption, however, so attackers need to compromise between more precision and faster analyses. While context sensitivity has been shown to be both useful and practical in the context of multiple whole-program binary analyses such as liveness analysis and constant propagation [11–13], we know of no path-sensitive variants that are practical. As context-sensitive analyses do not consider separate contexts for factored code fragments that do not look like procedures, they are of little help to attackers that aim to recover the same information they would on unprotected code.

Secondly, powerful, automated deobfuscation approaches are available that build on the detection of quasi-invariant behavior in obfuscated code. In essence, those techniques iteratively filter out and simplify instructions that are observed to behave quasi-invariantly (i.e., instructions that produce the same result every time they are executed on some selected program inputs), as well as code that does not contribute to the software semantics (i.e., to the input–output relation the software displays for the selected inputs). This deobfuscation approach has been shown to succeed in undoing obfuscations ranging from opaque predicates (with corresponding conditional branches that are either always or never taken) to the use of packers (because the unpacking of program code does not depend on program inputs). Based on our experience with human attackers, this form of deobfuscation is also performed mentally by attackers that analyze code manually, i.e., when attackers derive properties from program behavior observed, e.g., with debuggers. Although such derivations are often unsound, MATE attackers only care about the result, not about soundness.

By factoring out code fragments from multiple, unrelated contexts, we aim to prevent that the fragments and the surrounding control flow behave invariantly, and hence they fall victim to the generic deobfuscation approach.

3.2 Rationale for injecting fake edges

In addition to factoring, our strategy involves the injection of fake control flow transfers into the binaries, i.e., transfers that disassemblers will consider as possibly taken during a program's execution while they will never be taken in practice.

The reason to do so is that all disassemblers we know deploy recursive-descent algorithms to group disassembled code fragments into functions. In short, whenever a direct control flow transfer (or an indirect one of which the targets can be resolved) implemented with an idiom for intraprocedural control flow is observed, the disassembler considers the source and the target of the transfer to belong to the same function.

By injecting fake edges with intraprocedural control flow idioms between code from different protection components, we want to make the disassembler group code fragments into functions incorrectly. The effect on the produced CFGs of functions will depend on the internal operation and code representation of the disassembler. We observed two major cases.

The first case consists of tools such as IDA Pro and GHIDRA that are engineered from the ground up on the assumption that each code fragment can only belong to one function. These tools hence partition the identified basic blocks into function CFGs that start at the identified function entry points. Basic blocks are iteratively assigned to functions if they are connected via intraprocedural looking CFG edges to basic blocks already assigned to a function. This iterative assignment is by default greedy: Once a block is assigned to a function, the disassemblers will never move it to another function (unless being instructed to do so by, e.g., by a attacker plug-in, as we will discuss later).

Basic blocks connected by fake edges can as a result be put into the same function incorrectly. Precisely where this will happen depends on the order in which the greedy algorithm iterates over the basic blocks. In other words, it depends on internal tool implementation details. In any case, the reconstructed functions can then mistakenly contain blocks from unrelated functions (i.e., unrelated except for the fake relation through the fake edges). As each block is only put in one function, a block being put into the wrong function implies that another reconstructed function misses that block. So in IDA Pro, GHIDRA, and alikes, the reconstructed functions

can be at the same time over-approximations and under-approximations of the original functions.

The same effect will, by the way, also be a side-effect of factoring, as the disassemblers will then put each factored block in only one function. They can then put the immediate successors of the factored blocks into that function as well, some of which will be put in there incorrectly.

The second case is Binary Ninja, which does not make any assumption about the number of functions to which a code fragment can belong. Instead, when it identifies a function entry point, it adds all basic blocks to the corresponding function for which it finds a possible (true or fake) path through intraprocedural looking control flow transfers that it can resolve. This includes all direct transfers, but also indirect transfers of which it can resolve the potential targets. In Binary Ninja, the injection of direct, fake, intraprocedural looking edges can then only lead to reconstructed functions becoming more over-approximations of the original functions. (Obviously, because of unresolved indirect transfers, the reconstructed functions can still be under-approximations at the same time, but that is orthogonal to our work.)

Also for Binary Ninja, factoring will have a similar effect on functions being over-approximated, because all successors of all factored block dispatchers of which Binary Ninja can resolve the potential targets will be put in all functions to which the factored blocks are added.

In conclusion, both classes of disassemblers can be thwarted to some extent by injecting fake edges and by code factoring. They can then produce incorrect CFGs that mix parts of the original protection components, thus hiding their boundaries from attackers, and thus making their deployment more stealthy.

3.3 Rationale for code layout randomization

As discussed above, our strategy involves confusing attackers and the tools by factoring out originally unrelated fragments and by injecting fake edges such that code fragments originally belong to different protection combines get mixed up, and such that code fragments play multiple roles to further add to the confusion.

Injecting edges by itself will not be enough, however. Manipulating CFGs by adding edges only mixes up the logical structure of the software, not the spatial structure. To avoid that attackers can undo the mix-up by relying on spatial structure, we will combine fake edges and factoring with a spatial transformation consisting of code layout randomization. So at the top level, our approach consists of three transformations, which we discuss in more detail in the next three sections.

Finally, we concede that both the attack model discussed in the previous section and our strategy to hinder attacks are fuzzy rather than well-defined. To the best of our knowledge, in the domain of practical software protection against MATE attacks, there is no alternative, however.

4 Code layout randomization

Attack heuristics include spatial proximity. Each source code file typically contains code fragments that are closely related. Software libraries to link programs against are also structured along related functionality.

Compilers and linkers typically do not mix the binary code generated for different functions in a source code file. Whole function bodies are typically placed one after another in the text sections of object files, and text sections of object files are placed one after the other in linked applications or libraries, in which they are largely grouped by the archive from which they were linked in. Unless countermeasures are taken, related code fragments are hence more likely located close to each other in binaries. Attackers hence sometimes use proximity as a guide during their hunt for code to attack. In other words, they sometimes browse the code linearly.

Taking countermeasures in a link-time rewriter like Diablo [11] is trivial, as already demonstrated in the context of software diversification [14]. Mixing unrelated code can be done at any level of granularity, because all code is represented in one big CFG [11], from which binary code in virtually any (randomized) order can be generated.

The level of granularity at which the code layout is randomized has to be considered carefully. At the coarsest level, we can simply leave function bodies intact, but randomize their order throughout a whole program or library, as previously proposed to prevent memory exploits [15]. This already breaks proximity assumptions regarding the archive and compilation unit levels. By mixing protection and application functionality, we can already improve the stealthiness of protection components. For example, identifying one function as one code guard computation then no longer automatically leads the attacker to the related functionality in related functions.

We can also randomize the order of instructions and basic blocks, and mix instructions from all function bodies. However, because of the used recursive-descent disassembler heuristics, such fine-grained code layout randomization by itself does not hamper the partitioning or grouping of code into functions by disassemblers as discussed before. Moreover, the extra branches and possibly worse instruction cache behavior following from fine-grained layout randomization can severely impact performance. When applied in isolation, fine-grained randomization below the function level is therefore costly but hardly useful.

When the randomization is combined with obfuscations that break the recursive-descent strategy of the disassembler, more fine-grained randomization can still be useful, however.

In that case, splitting up function bodies and placing the parts in a randomized order prevent the tools from deploying linear sweep strategies to make up for the then defunct recursive-descent strategy. How to do so is precisely the aim of the fake edge injection obfuscations discussed next.

5 Interprocedural opaque predicates

5.1 Disassembler function reconstruction thwarting

To thwart the strategy of partitioning or grouping of disassembled instruction sequences into functions based on direct control flow transfers, we have two options. First, we can replace direct transfers with indirect ones, such as branch functions [10], to prevent that the disassembler infers that two code fragments relate and belong in the same function. Note that this goal of thwarting the disassembler's function CFG reconstruction after bytes have already been disassembled into instructions is complementary to the original goal of branch functions, which was to thwart the disassemblers' ability to identify the locations of instruction bytes in the executables, which is known to be a difficult task [16].

Secondly, we can add "fake" direct transfers that trigger incorrect assignments of basic blocks to functions. Such transfers can be added easily by means of opaque predicates and corresponding conditional branches. If we choose the predicate of the conditional branch to opaquely evaluate to false, implying that the branch will never be taken, we can simply choose any point in the program as the target of the conditional branch, thus injecting branch-taken CFG edges between completely unrelated code fragments. If we choose the predicate to opaquely evaluate to true, we can inject fall-trough CFG edges between code fragments from completely unrelated functions. This is trivial with the already existing support for code layout randomization.

Importantly, whereas choosing the targets of the fake edges is to be done at link time when all linked-in codes are available, the actual injection of opaque predicates does not necessarily need to occur at link time. Source-level obfuscators or obfuscating compilers can be used for the latter as well. They can typically inject more complex opaque predicates, which are then integrated in the original code more stealthily as they are compiled together with the original source code as long as they can inform the link-time rewriter about the location of the opaque predicates in the code. Obfuscating compilers can do so by adding comments and mapping symbols to the generated assembly code or object code, and source-level obfuscators can do so by describing the locations of inserted opaque predicate code in terms of source line numbers. By means of debug information in the object files, a link-time rewriter can then translate the source line numbers to object code addresses, thus identifying the locations where fake edges can be redirected to unrelated fragments at link time.

Fake CFG edges confuse the disassembler tools' CFG construction algorithms because their recursive-descent strategies are implemented greedily: Starting from function entry points (identified through symbol information or pattern matching), they traverse the code and greedily assign traversed fragments to functions. During the traversal, they treat idioms for intraprocedural control flow, such as conditional branches, as precisely that: intraprocedural control flow. For unobfuscated compiled code, this works fine, because few if any source languages feature interprocedural gotos, and standard compilers do not insert interprocedural branches (with the rare exception of tail call optimization).

But without more complex data flow analysis or other mechanisms to distinguish real from fake direct edges out of conditional branches, the greedy strategies fail. Depending on whether a basic block is first reached through a fake or a true edge, it will be assigned to the correct or incorrect function body. This implies that we can try to steer the tools toward incorrect function partitioning and CFG reconstruction by inserting fake edges in a controlled manner, but it also implies that in the case of disassemblers that put a block in at most one function, like IDA Pro, the result of the partitioning will depend on the order in which basic blocks are traversed by the tools. In that regard, we observed that tools like IDA Pro tend to give fall-through paths precedence over branch-taken paths.

It is important to note that tools like IDA Pro offer different views on the CFGs to human attackers on the one hand, and to analysis tools on the other. In CFGs stored in a database in support of plug-ins and external analysis tools, IDA Pro stores all direct CFG edges it has discovered during the disassembly process. This includes all edges from direct transfers such as both paths out of conditional branches. This database hence includes the mentioned fake edges, which can be considered false positives (FPs). The IDA Pro GUI, which is typically used by humans to study code, however, does not display all such edges. Instead, it omits such edges if they are interprocedural according to IDA Pro, meaning that they connect basic blocks IDA Pro has put in different functions. So attackers manually browsing through CFGs in the tool's GUI do not get to see them. When fake edges are (accidentally) omitted that way, we can consider them as semi-true negatives (STNs). They are FPs in the database view, but true negatives (TNs) in the GUI view. When true edges are omitted as a result in the GUI, they correspond to semi-false negatives (SFNs). They are TPs in the database, but false negatives (FNs) in the GUI view.

SFN and STN CFG edges hamper manual code comprehension and code browsing activities on the GUI, as they result in code from different components, such as protections and original application code, being presented as if it

is part of the same functions, and code originating from the same functions not being displayed as such. By inserting such edges, we can contribute to a much more stealthy integration of protection components.

5.2 Resilience against counterattacks

So far, we only discussed the potency of code layout randomization and interprocedural opaque predicates to confuse attackers and tools. Another important aspect is resilience to attacks, because attackers can of course still deploy all kinds of automated attacks to make up for the deficiencies of the existing, basic CFG partitioning and grouping strategies. They include static attacks such as pattern matching [17], abstract interpretation [18], and symbolic execution [19] to detect opaque predicates, and dynamic attacks such as generic deobfuscation [20], synthetic code generation [21], and fuzzing [22]. The dynamic ones are not sound, but that typically does not hamper attackers.

A first, critical point to make is that none of the mentioned academic static techniques have been scientifically validated as successfully breaking complex forms of opaque predicates (such as the graph-based ones from Collberg et al. [23]) on software of real-world complexity. Symbolic execution, for example, was only tested on programs of at most two functions [19]. Abstract interpretation was only evaluated on opaque predicates of which the program slice (i.e., the code computing the predicate) consisted of a tiny fragment immediately preceding the conditional branch [18].

A second point is that some academic trace-semantics-based techniques such as synthetic code generation [21] aim for recovering the original semantics of short obfuscated code fragments in traces, but not for finding fake edges. Those edges correspond to the more generic concept of infeasible execution paths, which by definition do not occur in traces. Detecting the infeasibility as a form of invariant requires comparing multiple occurrences of a fragment in a trace. That is not in the scope of the existing synthetic code generation approach [21], but it is precisely what the so-called generic deobfuscation technique does [20]. We come back to the latter later in the paper.

In practice, we have observed that both pattern matching and local symbolic execution are effective attack techniques [1] that might be usable to counter our proposed transformations. In both cases, small slices of the predicates used in conditional branches are then analyzed to determine whether or not they (likely or definitely) correspond to opaque predicates. Depending on the size of the software under attack and the immediate availability of a working attack tool box, attackers perform this analysis manually or by means of tool plug-ins that automate the analysis. Less skilled attackers reuse existing plug-ins as is; expert attackers can also customize plug-ins. On small software, attackers prefer manual analysis when they assess that the cost of setting up and customizing the tools will not be worthwhile. As completely manual analysis does not scale to larger software with many predicate instances to analyze, automation is typically preferred for attacks on larger software. That automation also often requires manual effort, however, if only because the customization of plug-ins requires the attacker to first determine which forms of opaque predicates are useful to search for, i.e., which code patterns to try to support.

At first sight, the attacker's ability to perform these attacks seems not hampered by the interprocedural nature of the opaque predicates we propose to inject. After all, the interprocedural aspect only directly impacts the control flow from the conditional branch on, not the code computing the predicate leading up to the conditional branch.

However, by carefully choosing the targets of the fake edges, we can directly impact the code slices of the opaque predicates, or at least the perception thereof by the attacker. We can in fact do so trivially by interrupting a slice of one opaque predicate by means of a fake edge coming in from another one. In the best case, this results in the assembler mistakenly assigning the instructions computing the predicate to multiple functions. In that case, the GUI will not show all relevant instructions in one function CFG. This will certainly hamper all manual activities of the attacker as discussed above. But even if the whole slice is assigned to the same function and hence shown on screen with the correct control flow between the relevant instructions, the attacker will still to some extent be confused when the fake edge is drawn as well.

To overcome this confusion, how small or big it may be in practice, the attacker has to consider multiple instances of opaque predicates together. Consider the example in Fig. 1 with predicates of contrived simplicity. Fake edges are drawn dotted, but at first the attacker does not know they are fake. To learn that they are truly fake, the attacker needs to consider both fragments. In practice, we are not limited to coupling pairs of opaque predicates mutually, we can easily couple more in larger cycles. A local code comprehension task for the attacker then becomes a global one; the effort needed to undo the protection grows.

A similar reasoning holds for fully automated analyses. Had the opaque predicates not been mutually coupled in the example of Fig. 1, a *simple constant propagation*, applied locally and iteratively with unreachable code elimination, would have sufficed to detect them. In the coupled case, simple constant propagation no longer suffices. Instead, a more complex *conditional constant propagation* (CCP) [24] is now required. For the example of Fig. 1, a CCP starting only at the top blue block would never mark any of the blocks in red as reachable. Binary Ninja performs a similar conditional value set analysis (VSA) on each function to which its recursive-descent disassembler has first added all

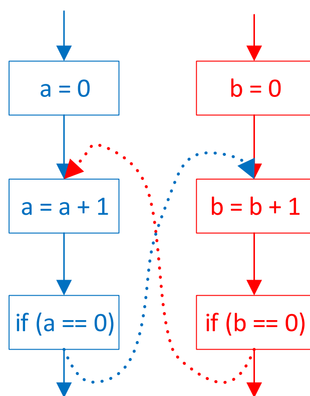


Fig. 1 Example of coupled opaque predicates (color figure online)

directly reachable blocks. In the example, if the top block is a function entry point, Binary Ninja's recursive-descent pass first adds all blocks except the top red one to the corresponding function, and then performs a conditional VSA starting at the entry block. In this simple example, the results of the VSA would indicate that the red blocks are not reachable from that function entry point. An existing Binary Ninja plug-in can then remove all red edges and blocks, and all dotted edges from the function, eventually returning a function with only the blue blocks and blue solid edges. Together with dead code elimination, this plug-in would hence be able to undo the opaque predicate insertion completely on this simple example.

In general, the (mutual) coupling of opaque predicates by letting fake edges interrupt slices implies that path-sensitive versions of analyses are needed. If those are applied locally, i.e., one slice at a time, they can suffice to identify likely opaque predicates, i.e., predicates that evaluate to constants on some execution paths. In that case, the necessary increase in complexity of the attack step is rather limited. If the attacker wants to deploy a sound(ish) analysis, however, to get a degree of certainty about the opaque predicates, the analysis has to be performed on all mutually coupled fragments together. This implies a considerable increase in complexity. In the evaluation section, we will observe and discuss how Binary Ninja's conditional VSA and other analyses fail to scale to realistically sized programs protected with coupled opaque predicates.

In summary, we can conclude that the resilience of code layout randomization and interprocedural opaque predicates, i.e., the effort needed to minimize their potency, with respect to attacks of which we know they are used in practice, is improved by coupling them in the proposed way.

Admittedly, this security analysis is fuzzy rather than well-defined. We consider a formal analysis out of reach at this point in time, not only for the protections against MATE attacks presented in this paper, but for most if not all MATE protections. In Sect. 7, we will perform a quantitative evaluation of a prototype implementation in which we mimick some real-life attacks.

Finally, we acknowledge that because it only injects invariant behavior into the software, the proposed protection via mutually coupled opaque predicates and code layout randomization does not protect in any way against dynamic attacks such as the tracing-based generic deobfuscation. While we deem this acceptable, as other protections can be used to shield of dynamic attacks, such as anti-debugging, anti-emulation, and anti-taint protections, we will still build on the protections presented so far in the next section to also make some dynamic attacks less effective.

6 Code factoring

To prevent that some of the stronger attacks can reconstruct the CFGs of a program's functions completely by identifying fake edges that can never be executed, we need to insert control flow transfers with more than one true outgoing edge. In line with what we discussed in Sect. 2, those true outgoing edges should look like intraprocedural edges. In other words, intraprocedural control flow transfer idioms should be used in general. In order to thwart the partitioning or grouping of code into functions, however, the edges should be interprocedural, connecting code from different functions.

We can meet these requirements by deploying control flow flattening [25] and branch functions [10] across multiple functions. Both control flow obfuscations can be implemented with many forms of intraprocedural looking control flow transfers such as conditional branches, switch tables, and computed jumps. Some simple examples are depicted in Figs. 2 and 3. However, in that case the transfers can still be observed to be semantically irrelevant: in a program trace, their executions will never depend on actual input values, only on constants such as those assigned to `next` in Fig. 2 and `param` in Fig. 3. Furthermore, apart from steering control to the appropriate continuation points depending on how they are reached, the injected code fragments then do not contribute to the output of the program. For both reasons, these fragments will get de-obfuscated by the approach of Yadegari et al. [20].

To counter this, we propose to combine the mentioned obfuscations with code factoring, as illustrated in Fig. 4. Blocks B and E are identical in the original code. If both of them can actually be executed in the original program, both edges coming out of the factored block BE will be executable in the transformed program. So the transfer at the end of block BE will show variable behavior. Moreover, the code in BE will be executed on data from two different contexts, and hence also display variable behavior. Moreover, as the original fragments B and E mattered for the original program, we can assume the factored block BE to be semantically rele-

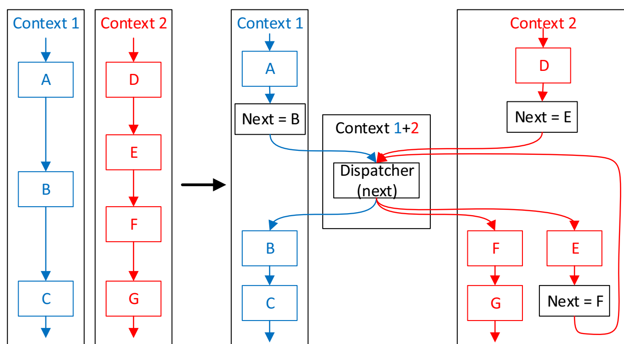


Fig. 2 Control flow flattening

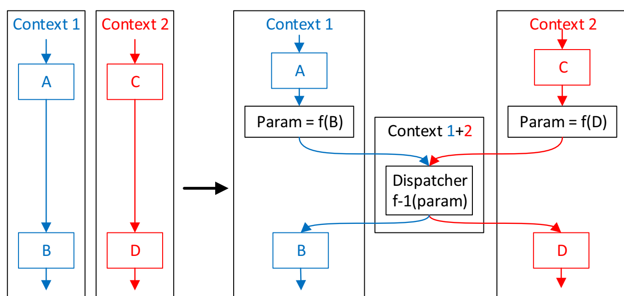


Fig. 3 Branch function

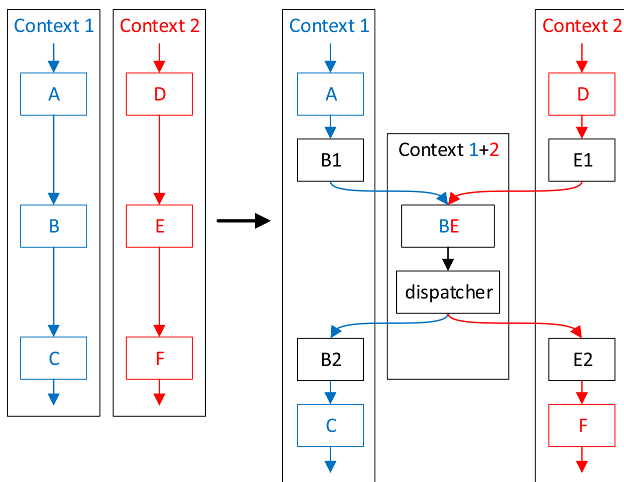


Fig. 4 Code factoring for obfuscation

vant in the transformed program. The generic deobfuscation approach of Yadegari et al. will therefore fail.

Code factoring is not new. Several forms have been proposed in the past to compact programs [12]. Our deployment of factoring serves the purpose of obfuscation, however, so it differs in two significant ways from previous deployments. First, we do not factor code into new functions that get called and end with return instructions. Instead, we use idioms of intraprocedural control flow, such as conditional branches, switch tables, and computed jumps. Secondly, we do not strive for more compact code. This implies that we can transform non-identical code fragments to make them identical, even when that involves prepending or appending extra instructions to the original fragments that move values between registers.

With these different requirements, we developed a significantly different code factoring technique. The most relevant aspects are a fast preliminary identification of potential fragments to be factored, the identification of actual factoring candidates, the order in which those are selected for transformation, the preparation of the selected ones, and the actual factoring transformations themselves.

6.1 Potential factoring candidates

To factor code, identical code fragments need to be identified or created. Existing factoring techniques [11,26,27] pre-partition code fragments using fingerprints. The fingerprinting functions are simple and strike a balance between recall and precision. They are defined such that code fragments that are "similar enough" to be likely candidates for factoring are mapped onto the same fingerprint. Much more complex and time-consuming precise checks of factoring pre-conditions, which also analyze the fragments' surroundings, are only performed on sets of fragments within the same partition, i.e., with the same fingerprint.

In existing code factoring techniques focussing on compaction, "similar enough" is defined as "nearly identical," i.e., having identical instruction schedules, and (almost) identical register allocations. The underlying assumption is that less similar fragments might well be factorable, but likely glue code will have to be injected around them before they can be factored, which will likely undo the compaction gains. Furthermore, to further limit the search space by focusing on worthwhile cases, existing techniques typically consider fragments consisting of one or more basic blocks, such as whole single basic blocks, single-entry CFG subgraphs of multiple blocks, and whole functions/methods [12,28–31]. An underlying assumption is that it is much less likely to find nearly identical, worthwhile fragments inside single basic blocks if the containing blocks are not nearly identical as a whole.

For our obfuscation purpose, the size of the glue code is only a secondary concern. We hence have to strike a balance differently. We opted to do so by not factoring fragments consisting of one or more whole basic blocks. Instead, we focus on slices (as defined by Horwitz [32]) and instruction sequences that are limited to, i.e., originate from within, single basic blocks. We only consider slices and sequences that exclude control flow transfer instructions.

The *slices* we consider as candidates for factoring are directed acyclic graphs (DAGs) with a single sink node. The DAGs' nodes are instructions, and their edges are data dependencies. Instructions can define multiple slices, ranging from

the single-instruction slice consisting of only the instruction itself, to the largest possible incoming data-dependency DAG within the instruction's basic block. Besides in the slices they define themselves, instructions can also show up in the slices defined on instructions further down in their basic blocks. The *sequences* we consider are sequences of instructions in the order in which they occur in the basic blocks. All subsequences of the instruction sequence constituting the block are considered. In the remainder of this paper, we use the term *fragments* to denote both slice and sequences. They are treated mostly identically in our factoring approach.

The only point where their treatment differs is in the computation of fingerprints. For sequences, we iterate over the instructions in their order in the original program. For slices, we iterate over the instructions in a canonical order that abstracts from the precise order in which the instructions occur in the program. This canonicalization is useful because nodes in a DAG are only partially ordered, and compilers generate different instruction orders for the same DAGs depending on the other instructions mixed in between them.

The fingerprints consist of the concatenation of at most four instructions' opcode (e.g., ADD, MOV,...), their operand types (e.g., two registers, one register and an immediate,...) and some of their flags (e.g., pre- or post-indexed). We found that including only four instructions in the fingerprint strikes a good balance between precision, recall, and memory consumption.

It can also be useful to consider the hotness of code fragments, i.e., their contribution to the total execution time of a program as determined with profiling. Excluding the hottest fragments helps to reduce the performance overhead.

6.2 Actual factoring candidates

Being nearly identical does not suffice for actual factoring. For sets of nearly identical fragments, we also need (i) to extract the fragments from their basic blocks; (ii) to make fragments truly identical by reallocating registers and by replacing non-identical immediate operands by constants stored in registers; (iii) to add a dispatcher to "return" from the factored fragment and to feed that dispatcher with the necessary inputs at each "call site." The latter two result in increased register pressure. Our binary rewriter does not convert the higher-level executable code to a higher-level IR. Hence, we need to transform the code and handle the register pressure locally. Concretely, this means we have to inject glue code in the form of register transfer instructions such as move, copy, swap, and spills to memory around the fragments. Foremost, we need to check whether we can actually perform the required rewriting within the capabilities (available transformations and analysis precision) of the link-time rewriter. As different dispatchers come with slightly different requirements, we also need to check which dispatchers can be used for which sets.

Figures 5, 6 and 7 illustrate the required transformations with 32-b ARMv8 code. The selected slices are marked in bold in Fig. 5. They have been rescheduled into separate blocks in Fig. 6. To enable the factoring already applied in Fig. 7, the differences in immediate operands and register allocations have been overcome by inserting a number of move-and-swap operations in blocks 1b, 2a, and 2b. The dispatcher in block 3b is a simple conditional branch. In the first instruction of block 2a, the controlling register r9 is set to zero, to control and enable the execution path 2a–3a–3b–2b. For controlling and enabling the path 1a–1b–3a–3b–1c, register r9 does not need to be set to a specific value. Instead, the fact that r9 is used as a base address in the store preceding slice 1 is relied upon: As user applications have no data mapped onto the lowest page in virtual memory, we can assume that r9 will be nonzero in the code following the store. This assumption is optional and can easily omitted in scenarios where it would not hold, such as kernel code.

To test whether sufficient glue code can be generated to make a fragment set actually factorable, we use a bi-directional, context-sensitive interprocedural liveness analysis [33]. To identify already available constants as input to dispatchers, we perform a flow-sensitive, context-sensitive (k-depth with $k = 1$) constant propagation analysis [34]. On top, we developed a simple flow-sensitive, context-sensitive (k-depth with $k = 1$), bi-directional, interprocedural nonzero analysis that tracks which registers hold values that are definitely nonzero. As these data flow analyses operate at the level of executable code, where useful alias information is sparse [35], they only analyze data in registers.

The constant analysis and the nonzero analysis allow us to reuse values that already have semantic relevance in the original program to control the dispatcher. If, for some factored fragment, this is the case for more than one of the contexts from which the factored fragment was extracted, the dispatcher is then controlled by semantically relevant data originating from more than one execution context. The invariants that held in those original contexts in isolation likely do not hold in the merged context after factoring. We conjecture that this makes code comprehension harder. It also ensures that deobfuscation techniques based on (quasi-)invariants will not work on the factored code.

In the example, slice 2's registers were renamed to those of slice 1. In many cases, candidate sets consist of more than 2 slices. Trying out all possible register renamings to select the best one would increase the code analysis time significantly, so instead we use a simple heuristic to select one of the slices as reference slice to which the others are renamed. This simple heuristic in practice also favors more likely successful renamings over less likely successful ones. In slice 1 of the example, the value loaded into r5 by the second load is live-

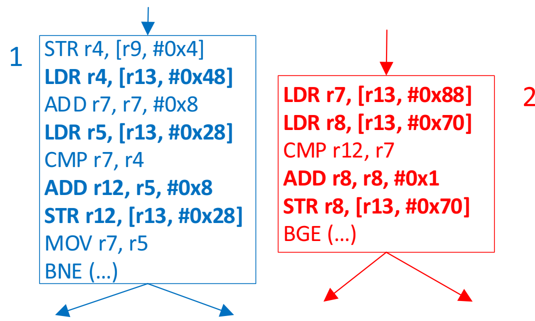


Fig. 5 Factoring candidate slices in bold in their respective basic blocks

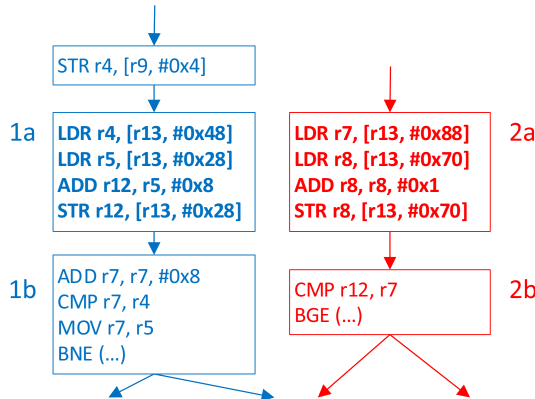


Fig. 6 Split factoring candidate slices

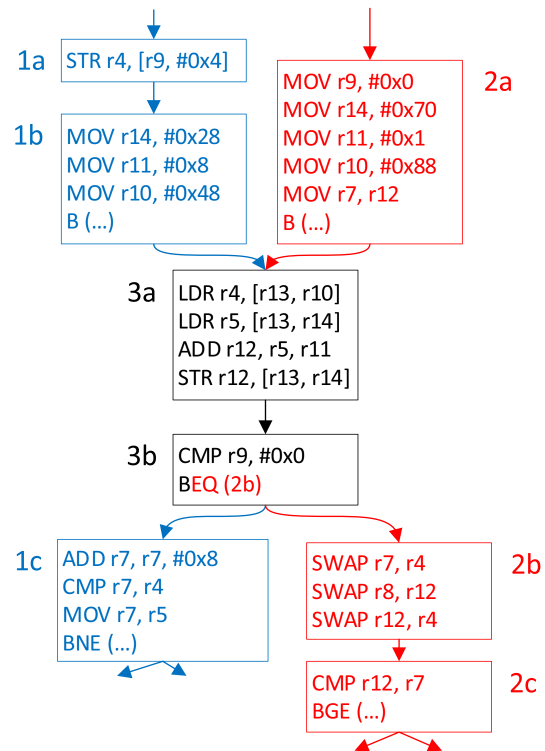


Fig. 7 Factored slices (color figure online)

out. In slice 2, the value loaded into r8 by the corresponding load is overwritten by the add. So an allocation like that of slice 2 cannot replace that of slice 1. In our simple heuristic, we count the number of different registers occurring in the original fragments, and we pick the one with the highest number as reference fragment. In case the heuristic does not favor one fragment over the others, and when (optional) profiling information is available, we pick the fragment with the highest execution count as reference fragment. While these simple heuristics are clearly not optimal, they provide a good balance between analysis time, performance and size overhead, and success ratio of the transformations.

6.3 Selection order

Instructions can be present in multiple factoring candidate sets, but each instruction can only be factored once. Furthermore, factoring a set of fragments changes the data flow properties in the surrounding code, e.g., by making previously dead registers containing nonzero or constant data live, so one factoring can impact the potential of another candidate one. The order in which we select and apply actual factorings is therefore important.

The selection order also needs to strike a balance between the level of protection and obfuscation speed. The former requires a global optimization and decision process that considers all potential candidate sets. However, that would require too much computation time. The potential candidate sets can be very large, up to hundreds of fragments, especially for small fragments of one or two instructions. The larger subsets thereof are typically not actual factoring candidates because our local register renaming technique is not powerful enough to overcome the differences in data flow properties of all the fragments surroundings. For smaller candidate subsets, the renaming is much more likely to succeed. Our approach hence starts from small candidate sets, that we expand as much as possible, i.e., as long as the estimated protection value increases.

6.3.1 Priority function

To order and compare candidate sets in terms of protection value, we need to consider measurable features (i.e., metrics) that contribute to the potency, resilience, and stealth of factoring them. We propose the following ones:

1. the fragment size as their number of instructions;
2. the numbers of archives, object files, and functions from which the fragments come;
3. the numbers of archives, object files, and functions in which fragments were observed to be executed for at

least one input, as determined by (optionally) profiling or fuzzing;
4. the possible dispatchers, and, if applicable, the already available constants or nonzero values.

The first metric prioritizes larger code fragments over smaller ones. We conjecture this is useful because factoring larger fragments results in more semantics being merged from different contexts, thus increasing the potency of a factoring transformation. It can also be useful for stealth, as it allows for better mixing of the injected dispatcher code with the factored code. Finally, it can contribute to the resilience against certain attacks. For example, undoing a factoring transformation by statically rewriting the code is more difficult when more instructions need to be re-inserted in the contexts from which they were factored.

The second metric, which actually consists of three metrics, contributes to potency. Assigning higher value to factorings of unrelated fragments originating from multiple object archives, object files, or functions, allows us to prioritize candidate sets that break proximity-based attack heuristics and that obfuscate component boundaries.

The third metric, again a set of three metrics, relates to resilience against dynamic attacks that build on observations of executions of the software under attack. These metrics allow us to prioritize candidate sets of which the effect of factoring them on the reconstructed CFGs cannot be undone by omitting edges and nodes that the attacker cannot trigger during dynamic attacks and by then simplifying the remaining code, as is done in the generic deobfuscation attack by Yadegari et al. [20].

The fourth metric allows to consider the potency, resilience, and stealth of the different types of dispatchers: Some are harder to analyze but not very stealthy (e.g., dynamic switch dispatchers); others are stealthy in the sense that they resemble already occurring fragments in the original programs (e.g., conditional jumps). Some are more resilient to automatic deobfuscation, others are less so. The different dispatchers are discussed in Sect. 6.4.

The metrics can be combined in a priority function in various ways: in weighted sums, in decision trees, etc. They can also be combined with profile information to give lower priority to fragments on frequently executed code paths to minimize the performance impact of the factorings. The definition of the best priority function is out of the scope of this paper. Importantly, a user of our protection tool chain can customize it depending on his use case at hand, taking into account the security requirements of the software assets at hand (confidentiality, integrity,. . .), a risk assessment of different attack scenarios, and the performance budget.

6.3.2 Selection and actual factoring

Our factoring algorithm consists of two phases.

At the start of the *selection phase*, we perform the already mentioned data flow analyses. Then a list of *initial factoring candidates* is assembled, ordered by their protection value. This list includes sets of fragments that are actual factoring candidates in the untransformed program. In other words, the data flow properties of the original program meet the necessary pre-conditions to apply the factoring transformations. No factorings are applied yet, however.

To decide on the initial candidate sets to add to the list in the selection phase, we implemented an iterative algorithm that is applied to each of the potential candidate sets. For each such set, the algorithm starts by marking *pairs* of fragments that can be factored, i.e., pairs for which register renaming can be performed and at least one dispatcher can be generated. Using the priority function to sort all possible pairs in terms of protection value, we select the best starting pair as the seed set. Next, we iteratively try to expand the seed set. In each iteration, we add the one fragment from the potential candidate set that results in the biggest increase in protection value. This continues as long as the protection value increases. The final expanded set is then added to the list of actual factoring candidates, in which we also keep track of the possible dispatchers, available constants or nonzero values, and other useful information to steer the dispatcher. The fragments in the expanded set are removed from the potential candidate set, and the whole process is repeated with other seeds until no sufficiently valuable seed sets can be found anymore.

In the *factoring phase*, we iterate over the ordered list of actual factoring candidate sets in decreasing priority. We factor each set if the necessary pre-conditions have not been invalidated by a previously applied factoring. Our prototype implementation can be configured on how to choose specific dispatchers from the available ones for each factoring, such as randomly or giving priority to specific forms. After each factoring, we update data flow information by means of incremental versions of the mentioned analyses to propagate the impact of the performed factoring on available registers, constants, and nonzero values to the necessary program locations.

6.4 Dispatchers

Many different dispatchers can be designed. We developed support for four types.

6.4.1 Conditional jump dispatcher

For sets of two fragments, a simple conditional branch can serve as dispatcher, as in Fig. 7. A branch condition like

equal-to-zero can be steered with a zero, and an unknown nonzero value that already has a semantic role in the original program. If no constants or nonzero values are available at the program locations of the original fragments, glue code is injected to produce them, possibly in an obfuscated manner and hoisted in the code such that a local static analysis does not suffice to detect it. We will come back to this in Sect. 6.5. Moreover, there is no need to keep it in a register, it can also be stored in memory. All kinds of schemes can be imagined that opaquely produce or load specific constant values or other values, always negative or always positive values, etc.

These dispatchers offer the major advantage that disassemblers like IDA Pro and Binary Ninja will recognize them as intraprocedural control flow, and thus we can rely on them to steer the disassemblers toward incorrect partitioning and grouping of code into functions.

In terms of pre-conditions, it is important to note that this type of dispatcher sets the processor's status flags. If those were live-out in the original fragments, it means the status bits have to be saved somehow, either in registers or by spilling them to memory. Saving and spilling status flags is rarely done in compiler-generated code, however, so when it occurs, it makes the code immediately suspicious in the eyes of attackers. For that reason, we opted not to use this type of dispatcher when the status flags are live-out in any of the involved fragments. Whether or not this is the best choice under all circumstances admittedly is open for debate.

For sets of more fragments, trees of multiple conditional branches can be used, but our prototype implementation is currently limited to single branches that are fed data (zeroes and nonzero values) directly through registers.

6.4.2 Indirect branch dispatcher

For larger fragment sets, we can use branch-to-register dispatchers, similar to the branch functions of Linn et al. [10]. In the simplest implementation, the exact addresses of the destination blocks are produced in the glue code preceding the extracted fragments, but less manifest schemes can easily be constructed.

Very simple schemes in which addresses are produced directly and locally, i.e., in glue code immediately preceding the transfer to the factored fragment, are not resilient to even relatively simple static analysis. For example, IDA Pro out-of-the-box identifies directly produced addresses during its recursive disassembly process and continues disassembling at those addresses. If the bytes at those addresses correspond to valid instruction encodings, IDA Pro adds the code at those addresses to CFGs, albeit in separate functions to which it does not create edges from the dispatcher. Complex schemes in which addresses are computed right before the branch-to-register instruction can be made completely resilient against static analysis and even the generic deobfuscation of Debray et al., but they come with the disadvantage that they are not at all stealthy. For example, it happens pretty rarely that values are XOR-ed before serving as a branch target, so schemes based on XOR-ing can be targeted with pattern matchers.

Unlike conditional jump dispatchers, IDA Pro does not add outgoing edges to this type of dispatcher. So while it can be used to prevent the tool from constructing complete function CFGs out of the box, it cannot, by itself, steer IDA Pro toward incorrect CFGs that incorporate basic blocks from multiple, unrelated functions. As we will discuss in Sect. 6.5, we can combine this type of dispatcher with other obfuscation constructs to reach exactly that.

In our prototype obfuscator, we only implemented support for schemes with direct address production in a dead register in the glue code preceding the factored fragments.

6.4.3 Static switch table dispatcher

Whereas computed jumps occur rarely in compiled C and C++ code, indirect jumps via table look-ups occur regularly, because switch statements are typically compiled into such look-ups. Two variations exist: address tables and branch tables. In the former, the address of the case to be executed is loaded from a table and jumped to, in the latter a computed jump is performed into a table of branches, which then forwards control to the case to be executed. Before the look-up, a bounds check is often performed. If it fails, control is transferred to the default case.

Table-based dispatchers mimicking switch dispatchers are therefore more stealthy than branch-function-like dispatchers. With this type of dispatcher, the glue code before factored fragments passes indexes to the dispatcher. These can again be produced directly or in some obfuscated way, and either locally or hoisted. Indexes can also be derived from known constants already in registers in the original code upon entry to the factored fragment.

The tables can be inflated with fake target addresses or jumps to fake targets. Tools like IDA Pro and Binary Ninja handle many patterns of switch table implementations and implicitly assume that the dispatchers implement intraprocedural transfers, so by implementing this dispatcher in a suitable pattern, they can be steered toward creating many fake edges that result in incorrect CFG partitioning and grouping. Disassemblers will typically also use the bounds check to determine the size of the table, so by inserting a fake bounds check, they can be fooled also in that regard.

In our prototype tool, we implemented support for both forms of tables. The tool inserts (fake) bounds checks if the condition registers are available. If not, there simply is no bounds check inserted. In that case, tools like IDA Pro typically do not analyze the switch statement and the table, and simply do not add outgoing edges at all.

Finally, we need to note that whereas look-up-based indirect control flow transfers are more stealthy than computation-based indirect transfers, their use for factoring can still lack stealthiness, in particular for large fragment sets. This is of course the case because in non-obfuscated and hence well-structured code, switch statements typically have a low fan-in. Our factored fragments, however, have a fan-in equal to their (true) fan-out. High fan-ins are suspicious in the eyes of attackers.

The strength of this form of factoring therefore has to come from its improved potency and resilience. The potency can be improved by combining this factoring with other obfuscations, as we will discuss in Sect. 6.5.

6.4.4 Dynamic switch table dispatcher

To improve both the potency of look-up-based dispatchers and their resilience against static analyses, we propose to make the look-up tables dynamic rather than static.

In compiled code, there is a static one-to-one mapping of dispatchers to tables. We are not bound by this restriction, however, and can let dispatchers dynamically switch between multiple tables. To that extent, we designed and implemented what we call *dynamic switch tables*. Given a set of global data tables, one such dispatcher may address any of these tables during the execution of the program. The key idea is to separate data table selection from its usage, both spatially and temporally. We do this by introducing the so-called *table selection points* in the CFG: locations where we insert a small instruction sequence to select one of the global data tables. We store the base address of the selected data table in a global variable used by the dispatcher. By separating the selection and use of the tables, a single dynamic switch table dispatcher may address different global data tables at different times during a single run.

Figure 8 shows the example factoring of two fragments B and E. The end result is shown on the right: three table selection points, the factored block BE, a dynamic switch table dispatcher, its global variable (`x`), and data tables `T1`, `T2` and `T3`. The glue code with the transfers to the factored block only contains instructions to produce the switch indices for each control flow path (`m` for fragment A and `n` for fragment B). The location where `x` gets assigned a new value does not really matter; the distance between the dispatcher and the table selection points can be arbitrarily large. Using a reachability analysis, the obfuscator determines which table selections reach which assignments of switch indices. In the example, selections of `T1` and `T2` reach the point where the index is set to m. This leads to the constraint that `T1[m]=T2[m]=C`. The tables need to be filled in respecting all such constraints. Similar to static switch tables, we can also add false entries (e.g., at index `m` in table `T2`) to confuse the attacker and his tools.

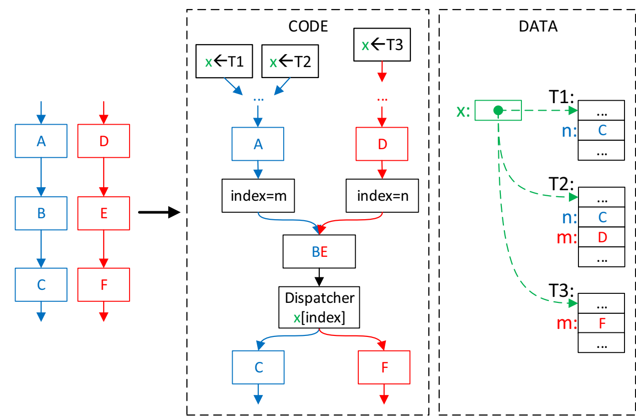


Fig. 8 Transformation with dynamic switch tables

Compared to static switch table dispatchers, dynamic table dispatchers increase the complexity by introducing an extra layer of indirection, which known static analysis cannot resolve, in particular when multiple obfuscations get combined, as will be discussed in Sect. 6.5. We also observed that these dispatchers mislead IDA Pro into constructing incomplete CFGs, because it is incapable of analyzing them properly. Consequently, the recursive-descent disassembler does not always disassemble all the instructions in the binary and associations between (sometimes large) portions of code are lost. The potency and resilience of this dispatcher are thus high. By contrast, this dispatcher is not stealthy: An attacker may find it strange that a dispatcher exists with no detected outgoing control flow. Given the high potency and resilience, we believe this lack of stealthiness does not completely void its usefulness.

The pre-conditions for this dispatcher are identical to the ones for traditional switch-based dispatchers, with the additional requirement that one extra register needs to be available to store a temporary value in.

6.5 Integration with other protections

A potential weak point of the factoring is that the computation of the values controlling the dispatchers (such as the index into a table, or a zero constant) is done in a linear control flow path leading up to the transfers to the factored code. We can fall back to all kinds of existing obfuscations to obfuscate this calculation, but the level of obfuscation is limited by the performance budget.

Complementary, e.g., to light-weight obfuscation, we can increase the potency and resilience of the proposed techniques by coupling the factorings. We can couple them with each other as well as with the opaque predicates. In Sect. 5.2, we already discussed how multiple opaque predicates can be coupled by directing fake edge to points in the middle of (other) opaque predicate computations. Likewise, we can

also redirect fake opaque predicate edges to the middle of instruction sequences that compute dispatcher control values. And we can choose the targets of fake entries in the tables of switch-based dispatchers in exactly the same way to obfuscate opaque predicate computations as well as dispatcher controller computations. That way, we turn the static analysis and deobfuscation of opaque predicates and factoring into one global hurdle for attackers.

7 Experimental evaluation

With our experimental evaluation, we aim at providing (partial) answers to the following research questions.

1. Are the proposed transformations easy to apply? In other words, are there enough relevant fragments to be found in real programs to which we can apply the transformations.
2. To what extent do the proposed transformations hamper an attacker that wants to reverse engineer protected programs? In other words, what is the potency of the proposed transformations?
3. How easy is it undo or circumvent the protection achieved by the transformations? In other words, what is their resilience against a number of feasible counterattacks.
4. What is the cost of applying the transformations in terms of overhead?
5. To what extent are the results dependent on the precise configuration of our tools.

In the following sections, we try to answer these questions to some extent by reporting on experiments we conducted. We do so by analyzing the effect that a prototype implementation has on a number of popular tools in attacker tool boxes when that prototype is deployed on benchmarks that are representative enough of real-world programs. At the end, we also draw some lessons from our experimentation.

7.1 Prototype implementation

We implemented the proposed techniques in the ASPIRE Compiler Tool Chain (ACTC) [36], which can compose multiple protections through source-to-source and binary code rewriting. All proposed techniques are implemented in Diablo [11], the ACTC's link-time binary code rewriter. The code is available as open source at https://github.com/csl-ugent/diablo/tree/oisp.

Our prototype has limitations. The binary rewriter does not support trees of conditional branch dispatchers, and lacks global register allocation and the option to spill and free status registers. Furthermore, the currently supported opaque predicates are limited to algebraic ones. More complex ones can be supported by combining the ACTC's source-to-source rewriting to inject complex predicates (e.g., graph-based ones [7] or predicates resilient to symbolic execution [37]) with binary rewriting to let fake edges cross component boundaries. Finally, the rewriter lacks support for C++ exception handling.

7.2 Benchmarks

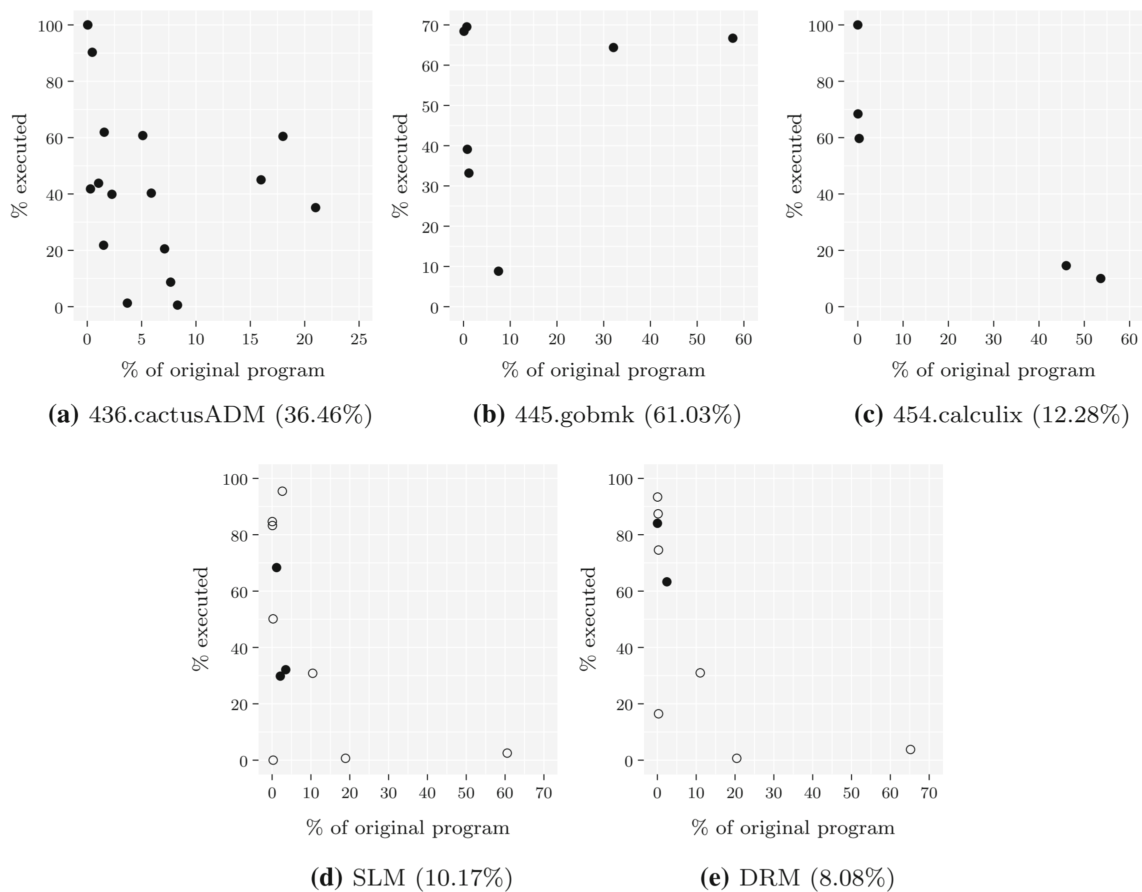
We have validated correctness on all C and C++ programs from the SPEC CPU2006 benchmark suite [38] (excluding `453.povray` and `471.omnetpp` that depend on exception handling) and on two industrial use cases from the ASPIRE research project [39]. Whereas the SPEC programs are stand-alone Linux binaries, the industrial use cases are dynamically linked Android libraries that are loaded into third-party applications. Nagravision contributed the first use case, a Digital Rights Management (DRM) plug-in that is loaded into the Android DRM and mediaserver daemon processes. SafeNet contributed the second use case, a software license manager (SLM) that is loaded into the Android Dalvik engine. Those daemons and engines are complex third-party multi-threaded processes that load and unload the libraries frequently. They hence stress-test our prototype.

The ASPIRE project deployed and validated the many ACTC-supported protections on those two use cases to mitigate attacks on the assets embedded in them, in line with the assets' security requirements as formulated by the security experts of the companies that contributed them [40]. As part of these protections, numerous archives are linked into the libraries. The protected use cases thus form perfect candidates to evaluate the proposed methods for stealthy, obfuscated integration of components proposed in this paper. Table 2 lists the deployed protections, and the number of components linked into the libraries thereto. In addition, we consider the SLM use case to consist of three components itself (the manager and linked-in open-source crypto and math libraries) and the DRM case of two components (the manager and some linked-in libgcc.a functionality). From the overall instruction count numbers in Table 2, it is clear that our use cases are not microbenchmarks, but applications and libraries of real-world complexity.

By contrast, the ACTC does not deploy additional protection on the SPEC benchmarks, as those embed no security-sensitive assets. Still, three of those benchmarks have their source code split over multiple directories: `436.cactusADM`, `445.gobmk`, and `454.calculix`. By treating each directory as a separate archive, we can still evaluate our techniques on them. Figure 9 plots the relative sizes of the benchmarks' components on the x-axis; the y-axis is the code coverage in the different components obtained when we profiled the benchmarks on our training inputs. These data enable the interpretation of measurement results below.

Table 2 Number of components in the benchmarks

		SLM	DRM	436	445	454
Number of archives constituting benchmark (● in Fig. 9)		3	2	17	7	5
ACTC protection archives linked into benchmark (○ in Fig. 9)						
Call stack checks	No support components linked-in	0	0	N/A		
Code mobility	Libwebsockets, libcurl, libssl, libcrypto, implementation	5	5			
Anti-debugging	Minidebugger	1	1			
Code guards	Implementation and guards	1	1			
Custom bytecode interpreter	Application-specific VM implementation	1	N/A			
Overall component (=archive) count		11	9	17	7	5
Overall instruction count without our obfuscations		276k	255k	99k	152k	366k



(a) 436.cactusADM (36.46%) **(b)** 445.gobmk (61.03%) **(c)** 454.calculix (12.28%)

(d) SLM (10.17%) **(e)** DRM (8.08%)

Fig. 9 Relative archive sizes and their individual coverage in the benchmarks, plus overall coverage per benchmark

7.3 Applicability

First, we analyze the applicability of the different transformations. Code layout randomization is applicable everywhere trivially. Opaque predicates and related conditional branches can also be inserted almost everywhere easily. In our prototype obfuscator, the user can specify the probability with which an opaque predicate is injected into each basic block. A pseudo-random process then chooses blocks and opaque predicate constructs accordingly.

By contrast, the proposed factoring techniques are not applicable trivially: factorable fragments need to be available, preferably over component boundaries. So first, we measured the applicability of factoring. Figure 10 shows the fraction of the original instructions that get factored in five cases: when all four types of dispatchers (indirect branches, switches, switches with dynamic tables, and condi-

tional jumps) are mixed with some randomization, and when each of those four is deployed in isolation. In each bar, the colored segments in the stack mark the number of different *archives* from which the slices/sequences factored together originate. The lowest segment corresponds to instructions that are factored from within only one archive. The second to instructions that are factored from within two archives, etc. It is clear that a considerable fraction of all instructions gets factored. It is also clear that the amounts of instructions factored from within multiple archives clearly correlate with the number of available archives and with the uniformity with which the application is partitioned into archives. Figure 11 similarly shows that many instructions are factored from within multiple object files, at least for dispatchers that support slice sets with more than two slices. Also at that level of granularity, and hence also at the still lower levels of individual functions and code contexts, the factoring approach is hence capable of obfuscating component boundaries.

In the context of dynamic attacks such as generic deobfuscation that focus on covered instructions with quasi-invariant behavior, it is also useful to know how many of the factored instructions were originally covered (i.e., executed) in one or more contexts. To that extent, Fig. 12 shows the distributions of the factored instructions in terms of the number of the covered slices/sequences from which they were factored. Each segment marks the fraction of all instructions that got factored in a set of slices/sequences, where the number of slices/sequences covered in the original program is indicated by the color of the segment. This means that the lowest segment corresponds to the instructions that got factored in a set of which no slice/sequence is covered in the original program. The next segment to instructions that got factored in a set in which one slice/sequence is covered, etc. The observed distributions are in line with the data in Fig. 9: When few instructions are covered in the first place, even fewer get factored from within one or more covered contexts.

Figure 13 shows similar data, but rather than considering all instructions, it only considers the covered instructions in the protected program, i.e., the instructions targeted by dynamic attacks. From the overall height of the bars, it is obvious that significant parts of the covered instructions are factored. Moreover, the vast majority of the factored covered instructions are factored from multiple covered contexts. This implies that in the protected program, most of the factored fragments are executed on data from two contexts. This implies that the injected dispatchers for the vast majority of the covered and factored slices do not display quasi-invariant behavior.

Figure 14 presents a further dissection of the factoring applicability, for the SLM benchmark. The heatmap displays the relations between the number of factored fragments in the protected program (color), the sizes of the factored fragments (first *x*-axis), the number of archives from which they are factored (second *x*-axis), and the number of archives in which the factored fragments were covered (*y*-axis).

As to be expected, the number of factored shorter fragments is significantly larger than that for longer ones. Second, the longer factored fragments all come from within a single archive. From further examination, we actually observed that the exceptionally long fragments originate from loop-unrolled code.

It is also clear that the most interesting factorizations, i.e., those from multiple contexts executed in multiple archives, are relatively rare, and involve only rather short sequences. This clearly indicates that there are practical limitations to the level of protection that our techniques can provide. Still, the colored cells in the upper right corner show that even if attackers completely neglect all uncovered control flow edges and code fragments, some dispatchers that are executed in more than one direction will keep hampering their reconstruction of the original program. In future work, we will investigate techniques to generate more and larger factorizable fragments by transforming code fragments rather than simply selecting existing ones like we do now.

7.4 Potency

To estimate the potency of the presented obfuscations, i.e., the extent to which they confuse human attackers, we performed three measurements on binaries protected with our Diablo-based tool. For these experiments, we configured the tool as follows. For factoring, we enable all dispatchers (with switch tables filled with 30% fake entries) and only factor fragments of at least 2 instructions but with no other restrictions, e.g., regarding hotness. We insert opaque predicates and corresponding conditional branches into 20% of randomly selected basic blocks, making the fall-through edge the fake edge whenever possible. After code layout randomization, we redirect fake edges throughout the binary to create cycles of four coupled obfuscations as discussed in Sect. 5.2.

7.4.1 Theoretical interconnectedness

First, we measure the extent to which the code of different components has become interconnected by intraprocedural-looking edges. For each instruction, we count from how many function entry points those instructions are reachable through intraprocedural control flow idioms only (i.e., through direct branches, fall throughs, switches, and from call sites to their corresponding return addresses). We then count from how many archives, objects, and functions those entry points originate. This metric thus measures the number of different components to which an attacker or his tools can potentially assign each instruction, and from which he has to make a choice to reconstruct the CFGs correctly. For the SLM

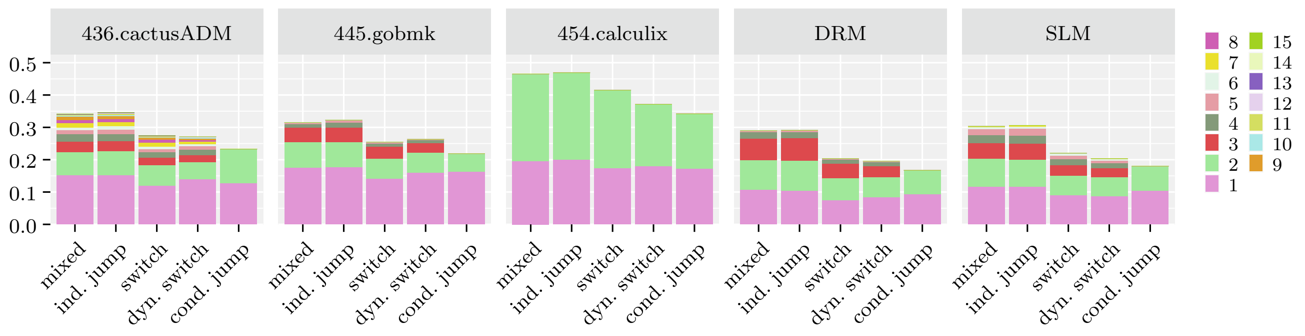


Fig. 10 Fraction of all instructions that get factored from within the indicated number of archives

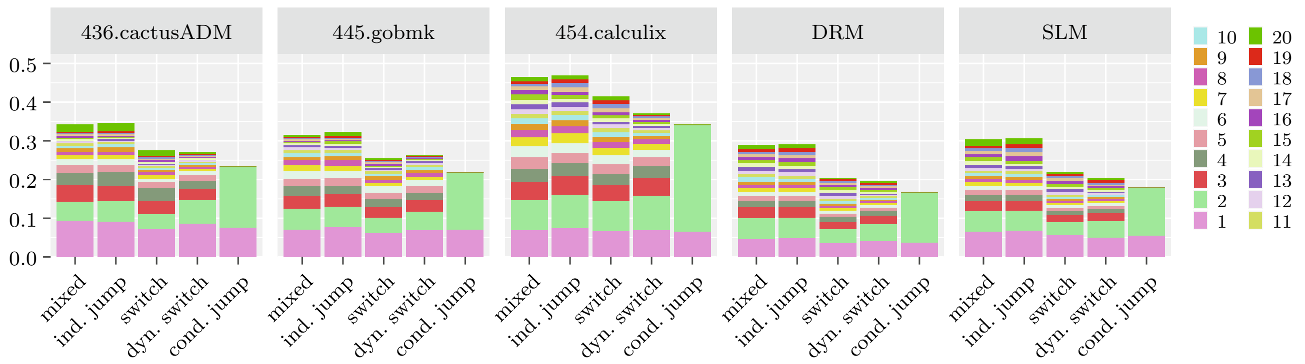


Fig. 11 Fraction of all instructions that get factored from within the indicated number of object files

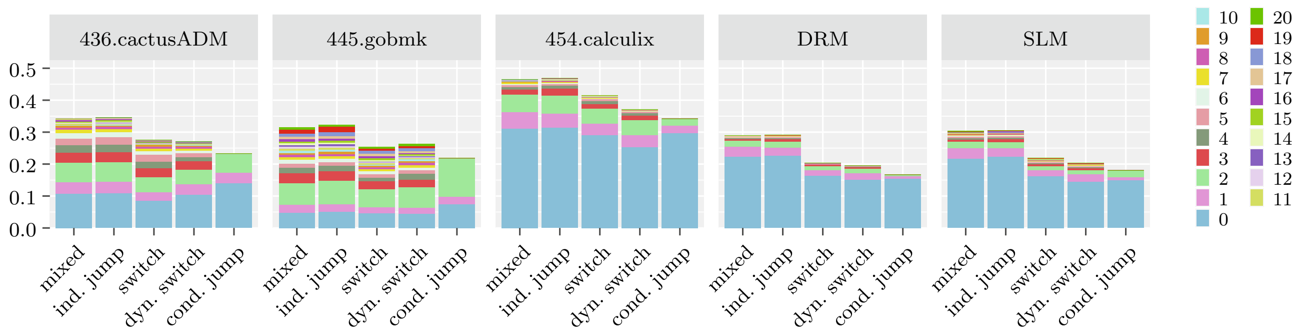


Fig. 12 Fraction of all instructions that get factored from within the indicated number of covered slices/sequences (irrespective of additional uncovered slices/sequences)

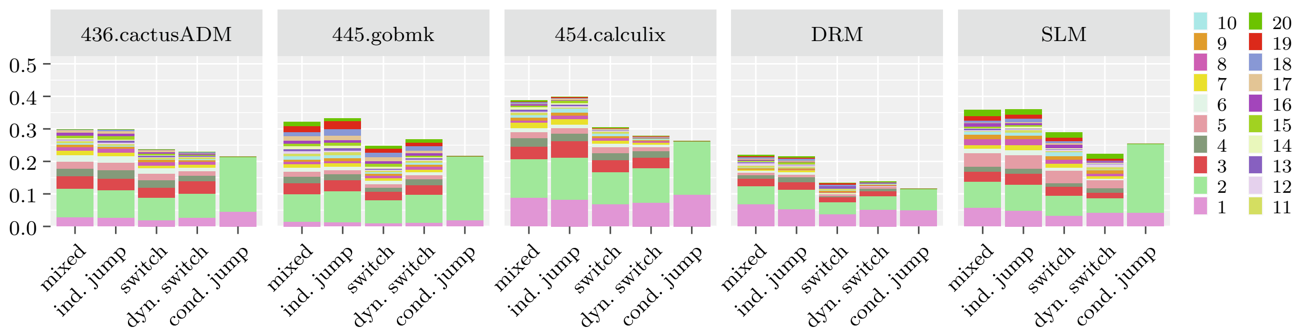


Fig. 13 Fraction of covered instructions that get factored from within the indicated number of covered slices/sequences (irrespective of additional uncovered slices/sequences)

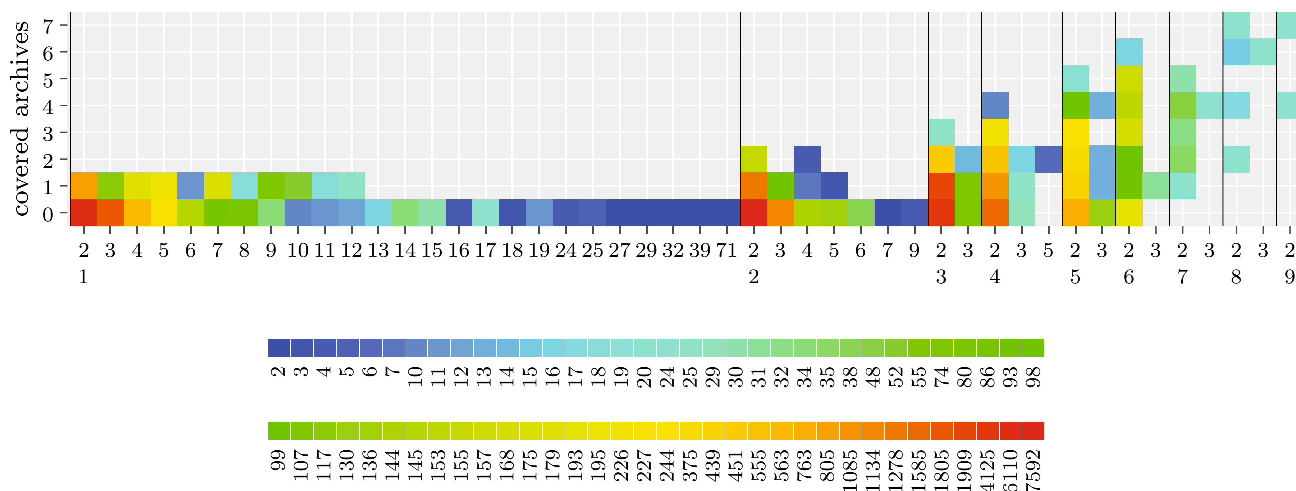


Fig. 14 Heatmap dissecting the applicability of factoring on the SLM benchmark, showing the number of fragments (color) of size (minor, top *X*-axis) in sets covering (major, bottom *X*-axis) archives versus the number of covered archives (*Y*-axis) in that set (color figure online)

benchmark, Fig. 15 shows the results. For other benchmarks, the results are similar. Before factoring, most code is reachable from a single function entry point, as one expects for code written in C. The few exceptions mainly originate from manually written and optimized assembly functions in the linked-in crypto library. After factoring, the vast majority of the code is reachable from within a vast number of function entry points that originate from a large number of different object files, and from all archives. The reason is that a large part of the code ends up in one big intraprocedurally strongly connected component in the combined CFGs of the program. So at least in theory, our transformations succeed in obfuscating the boundaries between components at the three levels of granularity.

7.4.2 IDA Pro

Secondly, we measure a practically oriented metric in the form of the amount of incorrect information that the popular reverse engineering tool IDA Pro (version 6.8.150428, 32-b) presents to the user due to the obfuscations. Concretely, we measure the fraction of fake CFG edges that IDA Pro stores in its database and/or shows in its GUI, as well as the fraction of true CFG edges that IDA Pro does not store and/or show. The former are FP rates, and the latter are FN rates.

It should be noted that IDA Pro is not designed for reverse engineering obfuscated binaries. In particular, it is not designed to handle basic blocks that are reachable via intraprocedural control flow idioms from multiple function entry points. It simply assigns basic blocks to functions based on the order in which the recursive-descent assembler visits them, not based on heuristics that take into account the effects of our transformations. IDA Pro can easily be augmented by an attacker, however, as it exports the constructed CFGs in a database that attacker scripts can manipulate. In other words, a skilled attacker can easily override and extend the disassembler and function reconstruction heuristics of IDA Pro.

To mimic skilled attackers, we experimented with various algorithms to maximize the amount of code in a binary that IDA Pro actually disassembles, as well as with various heuristics that repartition the disassembled code fragments (i.e., basic blocks) into functions such that the reconstructed functions better resemble the actual functions. We observed that many similar algorithms yielded very similar results, so the exact implementation details do not matter, as long as they incorporate three main ideas. First, one should try to put all identified code in functions, even if that code was not identified as being reachable by the original IDA Pro. This is the case, e.g., for code fragments that are only reachable through switch tables that IDA Pro cannot analyze precisely. Secondly, for such code fragments as well as for code fragments that the original IDA Pro already did put into functions, one should determine the function to which the fragment is most connected through incoming and outgoing intraprocedurally looking CFG edges in the IDA Pro database, and then put the fragment in that function. Finally, for determining the function to which a fragment is most connected and in which it hence belongs, one should assign different weights to different types of edges. Most importantly, edges originating from indirect control flow transfers such as those used to implement switches should have lower weights that other direct control flow edges. In addition, if the attacker knows somehow that the fake edges in opaque predicates are mostly fall-through edges or mostly taken edges, he can assign different weights to those types of conditional branch edges as well. Our code implementing these heuristics is available online at https://github.com/csl-ugent/oisp.

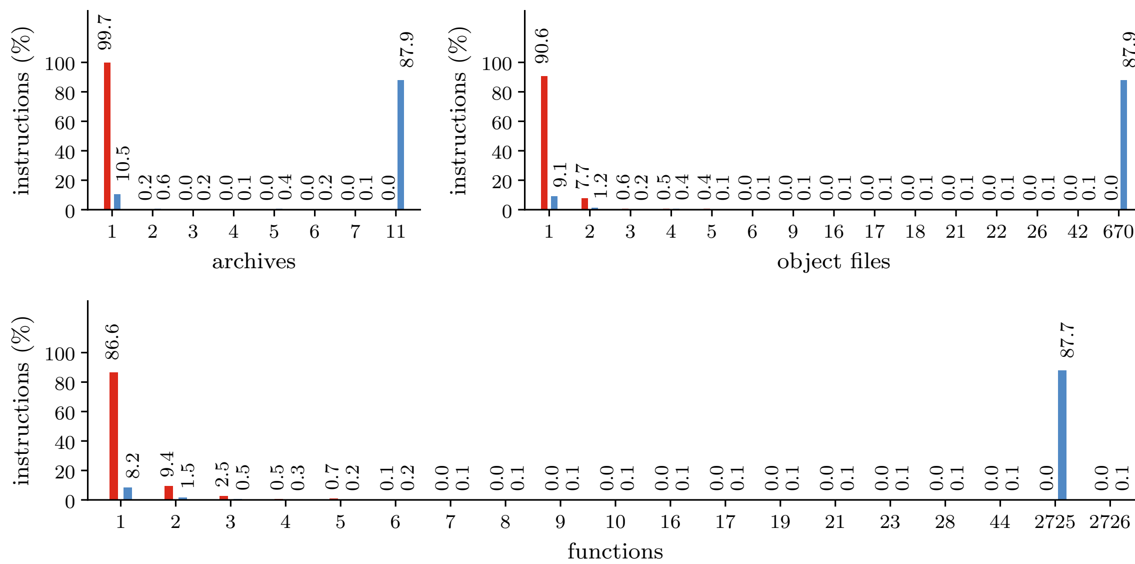


Fig. 15 Instructions reachable from function entry points in different numbers of archives/object files/functions, before (left bars) and after (right bars) factoring (SLM)

Table 3 Potency metrics for SLM without the protections proposed in this paper

	FP/FN CFG edges drawn in GUI							FP/FN CFG edges stored in database						
	Total	IA	IO	IF	iA	iO	iF	Total	IA	IO	IF	iA	iO	iF
# FP	0	0	0	0	0	0	0	0	0	0	0	0	0	0
FPR	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
# FN	1.8k	24	618	753	1.8k	1.2k	1.1k	916	0	45	48	916	871	868
FNR	3%	0%	1%	1%	3%	2%	2%	1%	0%	0%	0%	1%	1%	1%

Pairs of fragments split by factorization			CFG edges				Instructions	
Total	Wrong	Correct	Total	True	Fake	Drawn in GUI	Total	Functionless
0	0 (0%)	0 (0%)	67.2k	67.2k (100%)	0 (0%)	65.3k (97%)	281.8k	4.0k (1%)

Tables 3, 4 and 5 present the results for the SLM benchmark. Similar results for the other benchmarks are available in a technical report [41]. The top part of each table shows the aforementioned FP and FN rates of correctly or incorrectly handled CFG edges. The bottom parts additionally present the total amounts of edges and instructions in the binaries to ease the interpretation of the false rates, where we also mention how many edges are drawn in the GUI. The overall counts and corresponding false rates are further refined into 6 partially overlapping categories xy, with x being either I (Inter) or i (intra), and y being A (archive), O (object file), or F (function). The category IA, for example, is that of edges from a block originating from one archive to a fragment originating from another archive, i.e., interarchive, while category iO is that of edges between two blocks originating from the same object file. Furthermore, we present separate numbers for the edges that IDA Pro stores in its database because it has detected them in the code, and the ones it shows in the GUI because it considers them to be intraprocedural edges, meaning that it has correctly or incorrectly put the source and sink nodes of the edges in the same functions.

Table 3 shows that for the unprotected program, IDA Pro does a pretty good job in detecting the true edges. There are no fake edges of course, and most code is put into functions. Exceptions are rare, and mostly related to manually written and optimized assembly functions in the linked-in crypto library that feature interprocedural jumps.

Table 4 shows that IDA Pro out-of-the-box performs poorly on a protected program. In the GUI, it draws about 74% of the fake edges (75–76% for other benchmarks), of which more than half connect blocks from different archives. Furthermore, the GUI does not draw 56% of the true edges (53–56% for other benchmarks). As a result of the obfuscation, IDA Pro also gave up on about 28% of the identified instructions (23–31% for the other benchmarks), and simply did not put that code in any function. Obviously, this also contributes to the FN rates.

Table 4 Potency metrics for a fully protected SLM with IDA Pro out-of-the-box

	FP/FN CFG edges drawn in GUI							FP/FN CFG edges stored in database						
	Total	IA	IO	IF	iA	iO	iF	Total	IA	IO	IF	iA	iO	iF
# FP	16.5k	9.6k	12.5k	12.6k	6.9k	4.0k	3.9k	20.0k	11.8k	16.0k	16.0k	8.2k	4.0k	3.9k
FPR	74%	43%	57%	57%	31%	18%	18%	90%	53%	72%	72%	37%	18%	18%
# FN	101.4k	24	622	760	101.3k	100.7k	100.6k	63.2k	9	131	165	63.2k	63.1k	63.0k
FNR	56%	0%	0%	0%	56%	55%	55%	35%	0%	0%	0%	35%	35%	35%

Pairs of fragments split by factorization			CFG edges				Instructions	
Total	Wrong	Correct	Total	True	Fake	Drawn in GUI	Total	Functionless
28.4k	26.6k (94%)	1.7k (6%)	204.4k	182.2k (89%)	22.2k (11%)	97.4k (48%)	772.3k	213.6k (28%)

Table 5 Potency metrics for a fully protected SLM with attacker-improved IDA Pro

	FP/FN CFG edges drawn in GUI							FP/FN CFG edges stored in database						
	Total	IA	IO	IF	iA	iO	iF	Total	IA	IO	IF	iA	iO	iF
# FP	17.1k	10.0k	13.0k	13.0k	7.1k	4.1k	4.0k	21.2k	12.6k	17.0k	17.1k	8.6k	4.2k	4.1k
FPR	77%	45%	59%	59%	32%	18%	18%	96%	57%	77%	77%	39%	19%	18%
# FN	74.5k	16	492	588	74.5k	74.0k	73.9k	27.5k	0	17	20	27.5k	27.5k	27.5k
FNR	41%	0%	0%	0%	41%	41%	41%	15%	0%	0%	0%	15%	15%	15%

Pairs of fragments split by factorization			CFG edges				Instructions	
Total	Wrong	Correct	Total	True	Fake	Drawn in GUI	Total	Functionless
28.4k	24.1k (85%)	4.3k (15%)	204.5k	182.3k (89%)	22.2k (11%)	124.9k (61%)	772.4k	122 (0%)

Notice how these total numbers are comparable for different benchmarks, despite their different constitution. This is of course due to the fact that the totals do not depend on the number of archives or object files making up the programs. For the intra- and interarchive FPs, the rates vary more from one benchmark to another, but they are still comparable. For example, the GUI IA FPR with IDA Pro out-of-the-box ranges from 39 to 55%. All numbers are available in the technical report [41]. This relatively small variation implies that the obtained potency ports rather well from one benchmark to another, which is of course beneficial for users of tools that implement the obfuscations, as it will limit the need to retune the tool configuration for each benchmark.

At first sight, it might seem strange that there are also intrafunction GUI FPs, since we never purposely inject fake intrafunction edges. Those FPs are a side-effect, however, as they correspond to the never executed fall-through paths of injected switch dispatchers, which are intrafunction in our prototype.

Table 5 shows that an attacker-improved IDA Pro puts almost all code into functions. The FP rates go up as a result, and the FN rates drop significantly. Different versions of the repartitioning algorithm never got significantly better results than the ones reported here. Without more advanced data flow analysis or other attacks to identify fake edges, those edges simply confused the disassembler's code partitioning strategies. The proposed protections thus display a significant amount of practically relevant potency.

The above results and in particular the FNs are to some extent inherent to IDA Pro, which can put each basic block in only one function. For the example of Fig. 7, at least one of the incoming edges of block 3a and one of the outgoing edges of block 3b inherently become FNs. So additionally, we measure how many (source, sink) pairs of code fragments that were split apart by factorization (e.g., pairs (1a,1b) and (2a,2b) in Fig. 6) are correctly put in the same function by IDA Pro. The results are presented in the bottom left parts of the tables. Most importantly, the results in Table 5 indicate that even with the repartitioning heuristics, the vast majority (85%, 85–88% for the other benchmarks) of related block pairs are not put in the same function. There are two reasons. First, when the factoring is applied as frequently as we applied it, many non-factored fragments end up in between two factored fragments, and thus are no longer connected directly to any non-factored fragment. Secondly, even if we drop the frequency of factoring to a low number (such as 1% of all factorizable cases), the number only drops to about 82%. It remains that high because of the negative impact of the opaque predicate insertion on IDA Pro's performance. When no opaque predicates are inserted at all, and very little

Table 6 Binary Ninja results for three benchmarks. For each benchmark, the static number of instructions in the original benchmark and the number in the protected one are presented next to the benchmark name

	Original binary, default config				Protected binary, default config				Protected, large config	
	CF	B	BA	FU	CF	B	BA	FU	CF	B
bzip2 (10,671 ins original, 28,448 ins obfuscated)										
Analysis time (s)	0.03	0.03	0.30	1.06	10	380	1511	1497	9.86	379.89
Memory consumption (B)	85M	86M	129M	204M	135M	1.5G	11G	11G	135M	1.5G
% of instructions in CFGs	51%	0%	100%	102%	686%	667%	3902%	3964%	686%	667%
% of individual instructions identified	32%	0%	100%	100%	36%	34%	70%	70%	36%	34%
436.cactusADM (98,549 ins original, 313,375 ins obfuscated)										
Analysis time (s)	0.05	0.09	4.17	20.11	224	220	Failed	Failed	4802	Failed
Memory consumption (B)	87M	97M	277M	737M	2.5G	2.5G			2.5G	
% of instructions in CFGs	3%	3%	118%	109%	3172%	3171%			3171%	
% of individual instructions identified	2%	2%	100%	100 %	25%	25%			25%	
SLM (282,411 ins original, 905,217 ins obfuscated)										
Analysis time (s)	0.33	0.66	10.56	65.43	1758	1954	Failed	Failed	26,999	Failed
Memory consumption (B)	109M	150M	450M	1G	13G	13G			14G	
% of instructions in CFGs	6%	5%	103%	100%	6541%	6488%			6540%	
% of individual instructions identified	6%	5%	100%	100 %	21%	21%			21%	

factoring is performed, the number still does not drop below 49% (51–59% for the other benchmarks). The reason is that at about half of the points where factorization can be applied, the points before and after the factorized fragments are only connected via one direct control flow path, which then gets interrupted because of the factoring. More detailed results are available in our technical report [41].

We can conclude that unless IDA Pro gets the capability of putting blocks in more than one function, which by the design of its APIs seems like a rather fundamental and hence hard to change underlying principle of its implementation, the proposed factoring obfuscation has a strong potency.

7.4.3 Binary Ninja

Finally, we measured how well Binary Ninja (version 1.2.1954-dev, build ID af67f758) performs on our obfuscated binaries. This experiment is particularly interesting because Binary Ninja differs from IDA Pro precisely in the above aspect of putting blocks in multiple functions. More precisely, it adds all identified code fragments that are reachable from an identified function entry point through direct control flow transfers to the corresponding function's CFG. The theoretical results discussed in Sect. 7.4.1 hint that this will result in an explosion of the CFGs, and that is indeed what we observe in practice: In obfuscated binaries, most basic blocks that Binary Ninja identifies are part of the single strongly connected component that makes up the vast majority of the code and that is reachable from almost all function entry points. So Binary Ninja puts duplicates of those basic blocks in all the corresponding function CFGs. Those functions hence become too big for the more advanced data flow analyses in Binary Ninja. At the same time, a large fraction of the code is not identified as actual code, because it is only reachable through switch-based dispatchers of which Binary Ninja can resolve few if any targets. How many targets (and hence code that becomes reachable through those targets) it can detect depends on the type of dispatcher and the complexity of the analyses that are enabled and are able to execute in Binary Ninja, i.e., that scale up to the size of the exploded functions. As its most complex analyses only succeed on really small obfuscated benchmarks, we added the tiny benchmark program of bzip2 to our benchmark set, simply to confirm that Binary Ninja can handle at least such small benchmarks.

We experimented with the four available global analysis modes (CF: control flow, B: basic, BA: basic analysis, FU: full analysis, in order of increasing complexity), and initially used the default configuration parameter for analyzable function size, which prevents that badly scaling analyses are not run on overly large functions to avoid out-of-memory crashes and other issues. We also experiment with higher parameter values, in particular values large enough to cover the large function CFGs resulting from including all blocks in the strongly connected component created by our obfuscations. Table 6 presents the most interesting results.

On bzip2, all Binary Ninja analyses can run to completion with the default parameter. Even though the obfuscated version is "only" about three times as big as the original program, Binary Ninja requires three orders of magnitude more time for analyzing the obfuscated version, and the more complex analyses require 2 orders of magnitude more

memory. This is a first clear indication that our obfuscations stress Binary Ninja's scalability. The third line with results for bzip2 (and for the other benchmarks) shows how many instructions Binary Ninja included in all of its function CFGs combined (i.e., including duplicates when instructions are added to more than one function), relative to the number of instructions that really belong in the CFGs according to the ground truth. It can be seen that Binary Ninja's representation of the functions in the program is indeed blown up heavily, by a factor of 7 when only simple analyses and disassembler heuristic are used, and by a factor 39 when the more complex ones are used. This blow-up occurs despite the fact that Binary Ninja identified only about 35% and 70%, respectively, of the static instructions in the binary as code (as the result of unresolved dispatchers). We conclude that in its simplest modes, Binary Ninja is only able to identify 2/3 of the code as such, and already inflates its CFGs of the code fragments by an order of magnitude. With its more complex modes, it can identify an additional 1/3 of the code as such, but results in another order of magnitude more CFG inflation. So whatever combination of analyses is enabled in Binary Ninja, a reverse engineer is significantly hampered by our obfuscation. Quantifying this effect by counting false rates as we did for IDA Pro is meaningless here, as each true and fake edge can (and typically) now occur multiple times in Binary Ninja's internal database representation, as apparent from the already presented CFG inflation results.

On 436.cactusADM, our smallest true benchmark, all but the simplest Binary Ninja analyses fail to scale to the inflated function CFGs. When the more complex analyses are run, the tool crashes as it goes out of memory (on a machine with 64 GB of RAM) or as it tries to write out the huge database representing its IR of the program. The simpler modes fail to produce useful results as well, as they only identify about one-fourth of the code as such, but already inflate the CFGs with close to a factor 32. Deploying the simpler modes with a larger analyzable function configuration did not help, it only results in longer running times. Even if we only tried to run the more complex analyses on single inflated functions, the memory usage slowly increases while the analysis outruns our 24-h time limit. Similar results are observed for the larger benchmark SLM, only with worse results and much longer running times.

As for the simpler analyses CF and B modes in Binary Ninja, the fractions of the code identified as code in the three original binaries actually already indicate that those modes are next to useless because an attacker cannot rely on them to identify the relevant code.

In summary, none of the analysis modes in Binary Ninja succeeds in producing truly useful results for a reverse engineer of our protected binaries. In particular for non-trivial benchmarks, Binary Ninja seems to be completely defeated by our protections.

7.4.4 GHIDRA

While we lack the time and resources to conduct as extensive experiments with the recently released GHIDRA (version 9.1, build DEV 2019-Dec-02) reverse engineering tool suite as we did with IDA Pro and Binary Ninja, we did perform some preliminary experiments with it. GHIDRA behaves to a large degree similar to IDA Pro, in the sense that it puts each instruction in at most one function. Like in IDA Pro, this leads to false-positive edges and false-negative edges in the GUI showing the functions. Studying some microbenchmarks, we observed that GHIDRA's ability to resolve dispatcher targets differs somewhat, and also that is less aggressive in combining connected basic blocks to functions: Compared to IDA Pro, GHIDRA puts many more basic blocks from the protected binaries in separate functions.

GHIDRA's analyses are terribly slow. On non-trivial benchmarks, including bzip2 and the real benchmarks used in this program, we observed that the default analyses consume too much time to be practically useful. Even on the protected bzip2, the default analyses do not yield results for the first 24 h. Without those analyses, GHIDRA only produces a sequential listing of disassembled instructions similar to what can be obtained with the basic GNU binutils tool objdump.

Our overall first impression is therefore that attackers using GHIDRA are therefore at least as hampered by our proposed protections as attackers using IDA Pro. In future work, we plan to analyze which particular analyses fail to scale, and why.

7.5 Resilience

To evaluate the resilience of the presented obfuscation, we analyze to what extent some attack techniques observed in empirical research [1] and described in the literature [20] can bypass or undo the protections. Obviously, we cannot claim that the protections provide complete protection against attackers with unlimited resources and time. But we can demonstrate that at least some common attack strategies do not overcome the protection trivially.

7.5.1 Pattern matching attack in IDA Pro

First, we consider an attacker that can resolve opaque predicate computations when he observes their complete pattern in the code, either because he is good at recognizing them manually, or because he has a pattern matcher. We consider the attacker strong enough to identify opaque predicate computations even if they are mixed with other instructions, including (direct) control flow transfers. He is hence knowledgeable, but he is also prudent: If he only observes part of an opaque predicate computation or observes that only

Table 7 Metrics for a fully protected SLM, after detection and removal of observable opaque predicates (soundish DB attack)

	FP/FN CFG edges drawn in GUI							FP/FN CFG edges stored in database						
	Total	IA	IO	IF	iA	iO	iF	Total	IA	IO	IF	iA	iO	iF
# FP	16.4k	9.5k	12.3k	12.3k	6.9k	4.1k	4.0k	21.1k	12.6k	17.0k	17.0k	8.5k	4.2k	4.1k
FPR	74%	43%	55%	56%	31%	18%	18%	95%	57%	77%	77%	39%	19%	18%
# FN	73.4k	13	459	544	73.4k	73.0k	72.9k	27.5k	0	17	20	27.5k	27.5k	27.5k
FNR	40%	0%	0%	0%	40%	40%	40%	15%	0%	0%	0%	15%	15%	15%

Pairs of fragments split by factorization			CFG edges				Instructions		Opaque predicates	
Total	Wrong	Correct	Total	True	Fake	Drawn in GUI	Total	Functionless	Total	Resolved
28.4k	24.0k (85%)	4.3k (15%)	204.5k	182.3k (89%)	22.2k (11%)	125.2k (61%)	772.4k	122 (0%)	13.3k	29 (0%)

Table 8 Metrics for a fully protected SLM, after detection and removal of observable opaque predicates (unsound GUI attack)

	FP/FN CFG edges drawn in GUI							FP/FN CFG edges stored in database						
	Total	IA	IO	IF	iA	iO	iF	Total	IA	IO	IF	iA	iO	iF
# FP	14.9k	8.3k	10.9k	11.0k	6.6k	4.0k	4.0k	18.2k	10.3k	14.1k	14.2k	7.9k	4.1k	4.0k
FPR	67%	38%	49%	49%	30%	18%	18%	82%	46%	64%	64%	36%	18%	18%
# FN	73.0k	13	448	526	73.0k	72.6k	72.5k	27.5k	0	17	20	27.5k	27.5k	27.5k
FNR	40%	0%	0%	0%	40%	40%	40%	15%	0%	0%	0%	15%	15%	15%

Pairs of fragments split by factorization			CFG edges				Instructions		Opaque predicates	
Total	Wrong	Correct	Total	True	Fake	Drawn in GUI	Total	Functionless	Total	Resolved
28.4k	24.0k (85%)	4.4k (15%)	204.5k	182.3k (89%)	22.2k (11%)	124.2k (61%)	772.4k	122 (0%)	13.3k	3.0k (22%)

part of the computation is guaranteed to be executed leading up to the conditional branch, he does not guess that it will be an opaque predicate with a certain outcome. In the empirical experiments reported by Ceccato et al. [1], attackers described how they manually eliminated the identified fake edges and how they could implement simple pattern matchers to automate that attack task.

To assess how far such an attacker might get in the worst case, we implemented a script that iteratively removes all fake edges of opaque predicates that such an attacker can resolve. The script does not need to detect the patterns of the opaque predicate computations itself, instead it gets the necessary information from the ground-truth logs produced by our obfuscator.

We developed two versions of the script. A first one mimicks an automated attack that considers the information in IDA Pro's database. So it observes all edges and all code identified by IDA Pro. We refer to this attack as the "soundish" attack, because it considers all available code and control flow. As IDA Pro might have missed some code and edges, it is not completely sound, but it is the closest to sound an automated tool based on IDA Pro disassembler results can get.

The second version of the script mimicks a manual, human attack that considers only the information displayed in the IDA Pro GUI. This attack is on the one hand weaker because it does not resolve opaque predicates of which IDA Pro put parts of the computations in two or more different functions, as those parts are then not shown to the attacker together. On the other hand, this attack is stronger in cases in which IDA Pro has put all the predicate computations in the same function, but in which it does not draw a fake edge that arrives into the middle of the computations, i.e., in which such a fake edge is a GUI TN. So this attacker will miss some opportunities, but he will also remove fake edges because other (fake) edges remain invisible to him. We refer to this attack as the "unsound" attack, because the attacker chooses to neglect information readily available in the IDA Pro database that an attacker trying to be sound would not have neglected. As each deleted fake edge can result in opportunities to improve the partitioning of the code into functions, the scripts also execute the repartitioning algorithm discussed in the previous paragraph to potentially improve IDA Pro's performance after every deletion of a fake edge.

The results for the soundish attack are shown in Table 7; those for the unsound attack are shown in Table 8. To indicate to which extent the modeled attacker was able to resolve the opaque predicates, we report the number of inserted and resolved opaque predicates in the bottom right parts of the tables.

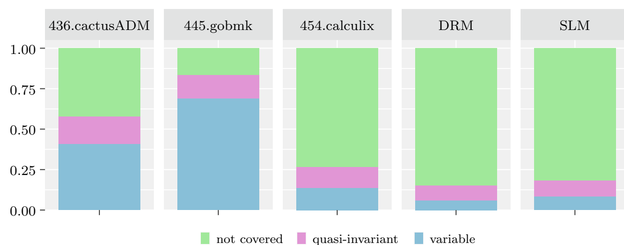


Fig. 16 Variability of dispatcher execution paths

With the soundish attack, almost no (0–1% for the other benchmarks) opaque predicates can be resolved. This demonstrates the effectiveness of the strategy to couple opaque predicates.

With the unsound attack, about 22% (20–22% for the other benchmarks) of the opaque predicates can be resolved. In this scenario, a relatively large drop of about 10% (9–11% for the other benchmarks) for the number of drawn fake edges in the GUI is observed. Still, about 67% of the fake edges remain. This is due to the coupling of opaque predicates in cycles, as discussed in Sect. 5.2 and because of the addition of fake entries in the switch tables of the dispatchers. Here, too, the number of true edges that do not get drawn remains high. While the attack has therefore weakened the confusion created by our obfuscations in the eyes of the attacker, he has not been able to remove it completely.

7.5.2 Generic deobfuscation

Regarding the resilience against the automated, generic deobfuscation technique of Yadegari et al. [20], we already noted in Sect. 7.3 that the majority of covered dispatchers does not display quasi-invariant behavior. Figure 16 shows the fractions of the dispatchers that are not covered (i.e., not executed for our training inputs), feature quasi-invariant behavior (i.e., "return" to only one "return site"), and show variable behavior (i.e., "return" to multiple "return sites"). Note the correlation with the overall coverage numbers in Fig. 9. Obviously, if only a small percentage of the code is covered, and factoring is done on both covered and uncovered slices, only a small percentage of the dispatchers will be covered, let alone display variable behavior. Of those covered, between 39% (DRM) and 83% (445.gobmk) have variable behavior, and will hence not be simplified by the quasi-invariance based generic deobfuscation.

7.5.3 Binary Ninja's conditional value set analysis

Finally, we studied the theoretical capabilities of Binary Ninja's conditional VSA as already introduced in Sect. 5.2 to reduce the size of the huge function CFGs it constructs. As discussed in Sect. 7.4.3, the current implementation of this analysis does not scale to realistically sized protected binaries, so we instead studied this capability on microbenchmarks such that we can assess the potential of the analysis in case it would be reimplemented to improve its scalability over its current quadratic complexity. This is the complexity of almost all dataflow analyses currently implemented in Binary Ninja, as their developers told us at the time of this writing.

When we ran Binary Ninja on microbenchmarks, we initially discovered that its VSA is pretty powerful. In fact, the analysis could resolve many of the opaque predicates that our initial prototype inserted, thus omitting many fake edges. It was also able to resolve many factored code fragments, in the sense that when a copy of the factored fragment is inserted into a function's CFG, the VSA would correctly detect which targets of the dispatcher belong in that function and which do not. For the opaque predicates, the reasons were their simple nature, as the ones implemented in our prototype initially included only simple algebraic predicates (such as $x^2 - x \bmod 2 = 0$) that were computed entirely in processor registers, starting from live register values from the original program. If the VSA was able to determine that those input registers (accidentally) held values from a limited set, it was also able to determine that opaque predicate could only evaluate to one value. Our obfuscator relied on much less advanced data flow analyses than Binary Ninja's VSA to pick the input registers for the inserted opaque predicate operations, so that accidental scenario occurred relatively frequently. Similarly, within each function the VSA was often able to determine the precise set of controller values that were being fed to the dispatcher of a factored block. This was again the result of our implementation being overly simplistic, as illustrated in the sample in Fig. 7. In the context of the red function fragment, the VSA identifies that controller register r9 can only hold the value 0, while in many contexts similar to that of the blue function fragment, the VSA was able to determine that r9 could only hold a limited set of pointer values, and is hence always nonzero.

In short, our initial prototype implementation was somewhat vulnerable to Binary Ninja, in particular when deployed on really small programs where the lack of scalability of Binary Ninja's analyses is not a problem for an attacker. This would also imply that our protections would be vulnerable if they are only deployed on a small part of a program, e.g., in case only a small part contains sensitive assets to be protected.

Fortunately, our initial vulnerability was easy enough to fix. In general, it suffices to make the inserted computations more complex than what the analyses can handle. As data flow analyses are always limited in precision to remain useable (i.e., have acceptable running times), injecting enough complexity in the code to thwart the analyses is always possible in theory. Of course, in practice the amount of required

complexity can come with a significant price in terms of additional overhead that needs to be injected, but with Binary Ninja, that was not necessary. From the Binary Ninja authors, we learned that its data flow analyses do not propagate any information through writable global data memory. So by simply adding a minimal number of memory indirections to the opaque predicate computations and to the code that sets controller values of dispatchers, we were able to completely mitigate the VSA of Binary Ninja. For example, to mitigate the analysis of the example in Fig. 7, it sufficed to replace the first move in the red glue code by a load operation that loaded the value 0 into r9 from an array stored in the mutable statically allocated data section. In the blue fragment, it sufficed to insert the same load of that value 0 from the same array, and to insert an addition that adds that 0 to the existing nonzero value in r9. As the VSA does not know that a zero is loaded in both cases, its analyses completely fail. Now while this may look like an overly simplistic remediation from our side, the general principle is that any scalable data flow analysis will have weaknesses, and that it suffices to exploit those in the remediation.

In the end, our simple fix sufficed to make the more complex analyses in Binary Ninja fail completely, both in terms of making it not produce good results, and in terms of requiring all to long running times on all, but the tiniest programs.

7.6 Overhead

Obfuscating transformations always come with performance and code size overhead. The performance penalty can be limited by using profile information to stay clear from the hottest code. As we only proposed a new way to redirect fake edges of opaque predicates, rather than introduce new ones which require new code sequences to be injected, we do not evaluate the performance penalty of opaque predicate insertion. Instead, we focus on the proposed factoring technique, which can involve the insertion of considerable glue code, and which is hence expected to have a major impact on performance and code size. Those impacts are summarized in Fig. 17. Solid lines represent run time overhead, dashed lines code size overhead. More detailed results and descriptions of the experiments are available in a technical report [41]. The measured run times are averages of 5 runs. For the SPEC benchmarks, we used slightly altered reference inputs to reduce run times on the (relatively slow) developer boards; for the SLM benchmark, we used a custom input; for the DRM benchmark, we have no run time measurement as this is an interactive application. Each pair of dashed/solid lines on the chart corresponds to one benchmark. The different points denote different amounts of factoring, guided by profile information. To collect profile information, (standard) training inputs were used that in each case differ from the measurement inputs. The measured versions range from

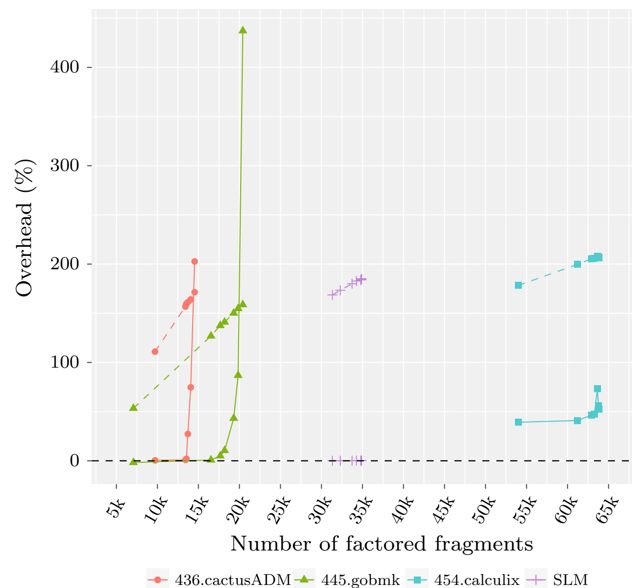


Fig. 17 Overhead versus factored code fragments

no covered code being factored (lower left points) to all code being factored (upper right points). In between, gradually more, hotter code (i.e., more frequently executed code) gets factored. It is clear that the overheads can become very large if the transformation is deployed blindly, but also that the overheads, in particular the performance overhead can be easily reduced by excluding the hottest fragments from the factorization. To what extent a certain reduction limits the practical effectiveness of the protection of course depends on the software at hand. In any case, excluding all covered code cannot result in factored code dispatchers with variable behavior. So clearly one should be willing to accept some performance overhead. We do not consider this a big problem: All MATE protections inherently come with some overhead. Note that for the code size, the smallest overheads are still rather large because we only excluded the executed code. If program size is more important than performance, a better strategy would be to exclude non-executed fragments. Then much smaller size overheads can still be obtained.

7.7 Sensitivity analysis

The opaque predicate insertion and factoring can be configured in many ways: the mixture of fake fall-through and fake branch-taken edges, amounts of fake edges in switch tables, use of different dispatchers, frequency of deployment, execution frequency threshold, priority function, cycle size of coupled protections and dispatchers, etc. A quantitative sensitivity analysis can be found in our technical report [41]. Some major qualitative results are that:

- the false rates rise with more fake fall-through edges;
- the FN rates increase with increasing cycle size until cycles of size 4. After that, the false-negative rates stabilize;
- the FP rates decrease with increasing cycle size.

7.8 Lessons learned

Throughout our experiments, we learned quite some useful lessons. The first is that it is really hard to come up with meaningful metrics to approximate the impact of protections on an attacker. Moreover, all metrics that we could come up with to reflect the impact of the protections depend heavily on the considered attacker tools and on how those tools model and handle the code of a program. Such metrics therefore have an ad-hoc nature that makes it difficult to compare the strengths of the protection against different tools, or to come up with a unified methodology to evaluate the defensive and offensive strengths of protections and attacker tools, respectively. Finally, the proprietary nature of the disassemblers with which we performed the most extensive experiments, and which to the best of our knowledge are the most popular in practice, makes it hard to understand why certain implementations of protections work better than others, or to predict the outcomes of potential changes to those protections. That proprietary, closed nature also makes it hard to evaluate custom attacks on newly proposed protection schemes, like ours, because it limits the ways in which existing analysis and heuristics (that have been tuned for unprotected binaries) can be tweaked to perform better on the obfuscated versions.

8 Related work

8.1 Code factoring

Existing work on code factoring focused mainly on compaction, i.e., the removal of duplicate code to make binaries smaller. Production tool chains already include optimization passes to factor identical procedures: Microsoft's Visual C++ compiler [28], GNU GCC [42,43], Gold [44] and LLVM [45]. In academic research, Debray et al. [12], De Sutter et al. [29] and Von Koch et al. [30] have developed code factoring techniques to factor almost identical code on the basic block level (the former two) and the procedural level (the latter two). Computation time is reduced by defining a fingerprint for each basic block and/or procedure, and small differences between procedures are compensated for by parameterizing the factored code. Debray et al. and De Sutter et al. mitigated differences between basic blocks by using an ad-hoc register renaming algorithm and by canonicalising the instruction schedule. This was not an issue for Von Koch et al. because LLVM IR was used. Recently, Rocha et al. [31] used a DNA sequence alignment algorithm from bioinformatics to identify factoring candidates and to compensate for differences between them. Similar to the work by Von Koch et al., they only support factoring on the procedural level but, as they implemented their technique on LLVM IR, they are not bound by register allocation schemes.

Inspired by the existing implementations, we factor (sub)blocks for obfuscation rather than compaction. Thus, we can give up on code size overhead to factor more code. Our technique is orthogonal and complementary to whole function merging, which by definition does not obfuscate function boundaries. Importantly, we rely only on intraprocedural control flow idioms rather than calls and returns.

8.2 Obfuscations

Many obfuscation transformations exist, each with its own strengths and weaknesses, as surveyed by Schrittwieser et al. [46]. Collberg et al. [47] categorized obfuscation techniques into layout (e.g., code layout randomization), control flow and data transformations. One example of control flow obfuscations is opaque predicates. These can range from simple [48] to complex [49]. While easy to implement, the simple ones are not resilient against modern attacks such as symbolic or concolic execution [19]. Recently, some alternatives were proposed to counter these advanced attacks. The range dividers of Banescu et al. [37] introduce additional feasible code paths, exploding the analysis complexity. The bi-opaque predicates of Xu et al. [50] exploit the NP-hard problem of resolving symbolic memory. Zobernig et al. researched a technique to make opaque predicates indistinguishable from the program's predicates by hashing the calculation of each (opaque) predicate [51,52]. Attackers need to invert the hash function to prove the opaqueness of a predicate, a process that is known to be impossible but for brute-forcing. Our use of opaque predicates in this paper is orthogonal to mentioned work, as our work focuses on choosing the target of the fake edge, which needs to be done whatever the kind of computation is used to implement the opaque predicate that steers the conditional branch.

Another example of control flow obfuscations is branch functions [10], which replace direct with indirect branches to thwart code identification heuristics. Control flow flattening [25] is another obfuscation, which replaces direct with dispatcher-based control flow. Our factoring dispatchers resemble these obfuscations, but focus on thwarting function repartitioning heuristics, as we aim for attackers to identify fake intercomponent control flow paths to confuse them even more and to hide the boundaries of components, rather than to obfuscate the components' internals themselves. Asghar et al. propose another way to obfuscate the control flow of a program by removing conditional branches [53]. Contrary to

other techniques, they avoid the insertion of additional calculations but instead build on the increased complexity of the linearized calculations.

Obfuscations can be inserted at source level [54], by compilers [55] or by binary code rewriters [11]. As we want to hide the boundaries of linked-in components, we obviously opted for a post-link-time rewriter.

9 Conclusions and future work

We presented a novel technique to apply code factoring across component boundaries with intraprocedural control flow idioms. We combined our technique with existing opaque predicates with which we also inject fake direct control flow across component boundaries, and with fine-grained code layout randomization. In our extensive evaluation with IDA Pro, a commonly used, state-of-the-art reverse engineering tool, we demonstrated that our techniques thwart IDA Pro's disassembler and CFG reconstruction heuristics and that a program protected with our technique is more resilient to some known attacks. For GHIDRA, another reverse engineering tool, preliminary experiments yielded similar results. For a third disassembler, Binary Ninja, which is built on very different principles, another extensive evaluation demonstrated that the tool becomes mostly useless on our obfuscated programs. We can conclude that our technique increases the potency and resilience of protected applications against modern reverse engineering attacks.

In future research, approaches to generate more similar, and thus factorable code fragments can be investigated, rather that only identifying existing ones. Another research path can be the use of machine learning techniques to steer the insertion of fake control flow, so that attack tools are more purposively thwarted rather than stochastically. Furthermore, more experimentation with the open-source tool GHIDRA can be useful to assess the potential of custom attacks and disassembler heuristics specifically tuned to the features of our protected programs.

Funding This research was funded by the Agency for Innovation by Science and Technology in Flanders (IWT) (Grant Number 141758). Part of this research was conducted in the EU FP7 project ASPIRE, which has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under Grant Agreement Number 609734. Part of the research was also funded by the Cybersecurity Initiative Flanders from the Flemish Government. Part of this research was also funded by the Fund for Scientific Research - Flanders (FWO) as part of project grant 3G0E2318.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Ceccato, M., Tonella, P., Basile, C., Falcarin, P., Torchiano, M., Coppens, B., De Sutter, B.: Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. Empir. Softw. Eng. **24**(1), 240–286 (2019)
2. Cabutto, A., Falcarin, P., Abrath, B., Coppens, B., De Sutter, B.: Software protection with code mobility. In: Proceedings of the 2nd ACM Workshop on Moving Target Defense, pp. 95–103 (2015)
3. Ceccato, M., Dalla Preda, M., Nagra, J., Collberg, C., Tonella, P.: Barrier slicing for remote software trusting. In: 7th IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 27–36 (2007)
4. Viticchié, A., Basile, C., Avancini, A., Ceccato, M., Abrath, B., Coppens, B.: Reactive attestation: Automatic detection and reaction to software tampering attacks. In: Proceedings of the 2016 ACM Workshop on Software PROtection, pp. 73–84 (2016)
5. Abrath, B., Coppens, B., Volckaert, S., Wijnant, J., De Sutter, B.: Tightly-coupled self-debugging software protection. In: Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, p. 7 (2016)
6. Ghosh, S., Hiser, J.D., Davidson, J.W.: A secure and robust approach to software tamper resistance. In: Proceedings of the International Workshop on Information Hiding, pp. 33–47 (2010)
7. Nagra, J., Collberg, C.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Pearson Education, London (2009)
8. Wang, Y.: Cognitive complexity of software and its measurement. In: 2006 5th IEEE International Conference on Cognitive Informatics, vol. 1, pp. 226–235 (2006). https://doi.org/10.1109/COGINF.2006.365701
9. Woodward, M.R., Hennell, M.A., Hedley, D.: A measure of control flow complexity in program text. IEEE Trans. Softw. Eng. **5**(1), 45–50 (1979)
10. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, pp. 290–299 (2003)
11. Van Put, L., Chanet, D., De Bus, B., De Sutter, B., De Bosschere, K.: Diablo: a reliable, retargetable and extensible link-time rewriting framework. In: Proceedings of the 5th IEEE International Symposium on Signal Processing and Information Technology, 2005, pp. 7–12 (2005)
12. Debray, S.K., Evans, W., Muth, R., De Sutter, B.: Compiler techniques for code compaction. ACM Trans. Program. Lang. Syst. (TOPLAS) **22**(2), 378–415 (2000)
13. Muchnick, S., et al.: Advanced Compiler Design Implementation. Morgan Kaufmann, Burlington (1997)
14. Coppens, B., De Sutter, B., Maebe, J.: Feedback-driven binary code diversification. ACM Trans. Arch. Code Optim. (TACO) **9**(4), 24 (2013)
15. Kil, C., Jun, J., Bookholt, C., Xu, J., Ning, P.: Address space layout permutation (ASLP): towards fine-grained randomization of commodity software. In: Proceedings of 22nd Annual Computer Security Applications Conference, pp. 339–348 (2006)
16. Meng, X., Miller, B.P.: Binary code is not easy. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 24–35 (2016)
17. Ngo, M.N., Tan, H.B.K.: Detecting large number of infeasible paths through recognizing their patterns. In: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and

the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 215–224 (2007)
18. Dalla Preda, M., Madou, M., De Bosschere, K., Giacobazzi, R.: Opaque predicates detection by abstract interpretation. In: International Conference on Algebraic Methodology and Software Technology, pp. 81–95 (2006)
19. Yadegari, B., Debray, S.: Symbolic execution of obfuscated code. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 732–744 (2015)
20. Yadegari, B., Johannesmeyer, B., Whitely, B., Debray, S.: A generic approach to automatic deobfuscation of executable code. In: IEEE Symposium on Security and Privacy, pp. 674–691 (2015)
21. Blazytko, T., Contag, M., Aschermann, C., Holz, T.: Syntia: Synthesizing the semantics of obfuscated code. In: Proceedings of the 26th USENIX Conference on Security Symposium, pp. 643–659 (2017)
22. Madou, M.: Application security through program bfuscation. Phd thesis, Ghent University (2007)
23. Collberg, C.S., Thomborson, C.D., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: POPL (1998)
24. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. (TOPLAS) **13**(2), 181–210 (1991)
25. Wang, C., Hill, J., Knight, J., Davidson, J.: Software tamper resistance: obstructing static analysis of programs. Technical Report, Technical Report CS-2000-12, University of Virginia (2000)
26. Debray, S., Evans, W., Muth, R.: Compiler techniques for code compression. In: Workshop on Compiler Support for System Software, pp. 117–123 (1999)
27. De Sutter, B., De Bus, B., De Bosschere, K.: Sifting out the mud: low level C++ code reuse. ACM SIGPLAN Not. **37**, 275–291 (2002)
28. /OPT (Optimizations)—Microsoft Docs (2018). https://docs.microsoft.com/en-us/cpp/build/reference/opt-optimizations?view=vs-2019. Accessed 17 Apr 2019
29. De Sutter, B., De Bus, B., De Bosschere, K.: Sifting out the mud: low level C++ code reuse. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), vol. 37, pp. 275–291 (2002)
30. Edler von Koch, T.J., Franke, B., Bhandarkar, P., Dasgupta, A.: Exploiting function similarity for code size reduction. ACM SIGPLAN Not. **49**(5), 85–94 (2014)
31. Rocha, R.C., Petoumenos, P., Wang, Z., Cole, M., Leather, H.: Function merging by sequence alignment. In: Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization, pp. 149–163 (2019)
32. Tip, F.: A survey of program slicing techniques. J. Program. Lang. **3**(3), 121–189 (1995)
33. De Sutter, B., De Bus, B., De Bosschere, K.: Bidirectional liveness analysis, or how less than half of the alpha's registers are used. J. Syst. Arch. **52**(10), 535–548 (2006)
34. Debray, S.K., Evans, W., Muth, R., De Sutter, B.: Compiler techniques for code compaction. ACM Trans. Program. Lang. Syst. **22**(2), 378–415 (2000)
35. Debray, S., Muth, R., Weippert, M.: Alias analysis of executable code. In: Proceedings of ACM POPL, pp. 12–24 (1998)
36. Basile, C.: D5.11 ASPIRE framework report. Techreport, POLITO (2016). https://aspire-fp7.eu/sites/default/files/D5.11-ASPIRE-Framework-Report.pdf. Accessed 17 Sept 2018
37. Banescu, S., Collberg, C., Ganesh, V., Newsham, Z., Pretschner, A.: Code obfuscation against symbolic execution attacks. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 189–200 (2016)
38. Standard Performance Evaluation Corporation: SPEC CPU 2006 (2018). https://www.spec.org/cpu2006/
39. Home—Aspire-FP7 (2018). https://aspire-fp7.eu/
40. De Sutter, B.: D1.06 ASPIRE validation. Techreport, Ghent University (2016). https://aspire-fp7.eu/sites/default/files/D1.06-ASPIRE-Validation-v1.01.pdf. Accessed 6 May 2019
41. Van den Broeck, J., Coppens, B., De Sutter, B.: Extended report on the obfuscated integration of software protections (2019). arXiv:1907.01445
42. Liška, M.: Optimizing large applications (2014). arXiv preprint arXiv:1403.6997
43. mliska: [PATCH 3/5] IPA ICF pass (2014). https://gcc.gnu.org/ml/gcc-patches/2014-06/msg01246.html. Accessed 17 Apr 2019
44. Tallam, S., Coutant, C., Taylor, I.L., Li, X.D., Demetriou, C.: Safe ICF: pointer safe and unwinding aware identical code folding in gold. In: GCC Developers Summit (2010)
45. Ueyama, R.: Elf: implement ICF (2016). https://reviews.llvm.org/rL261912. Accessed 17 Apr 2019
46. Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E.: Protecting software through obfuscation: can it keep pace with progress in code analysis? ACM Comput. Surv. (CSUR) **49**(1), 4 (2016)
47. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report. Department of Computer Science, The University of Auckland, New Zealand (1997)
48. Myles, G., Collberg, C.: Software watermarking via opaque predicates: implementation, analysis, and attacks. Electron. Commer. Res. **6**(2), 155–171 (2006)
49. Majumdar, A., Thomborson, C.: Manufacturing opaque predicates in distributed systems for code obfuscation. In: Proceedings of the 29th Australasian Computer Science Conference, vol. 48, pp. 187–196 (2006)
50. Xu, H., Zhou, Y., Kang, Y., Tu, F., Lyu, M.: Manufacturing resilient bi-opaque predicates against symbolic execution. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 666–677 (2018). https://doi.org/10.1109/DSN.2018.00073
51. Zobernig, L., Galbraith, S.D., Russello, G.: Indistinguishable predicates: a new tool for obfuscation. IACR Cryptol. ePrint Arch. **2017**, 787 (2017)
52. Zobernig, L., Galbraith, S.D., Russello, G.: When are opaque predicates useful? In: 2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), pp. 168–175. IEEE (2019)
53. Asghar, M.R., Galbraith, S.D., Russello, G.: Obfuscation through simplicity (2016). https://www.math.auckland.ac.nz/~sgal018/simplicity.pdf. Accessed 24 June 2019
54. Collberg, C., Martin, S., Myers, J., Zimmerman, B.: The tigress diversifying c virtualizer (2015). http://tigress.cs.arizona.edu/. Accessed 17 Apr 2019
55. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM—software protection for the masses. In: Wyseur, B. (ed.) Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015, pp. 3–9. IEEE (2015). https://doi.org/10.1109/SPRO.2015.10

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.