Potential Analysis of Software Obfuscation to Protect Unmanned Systems against Forensic Analysis

N. Bergmann[1], E. Padilla[1], and J. Bauer[1]
[1] Fraunhofer FKIE, Cyber Analysis & Defense, Fraunhoferstr. 20, 53343 Wachtberg, Germany
Tel.: +49 228 50212-595
E-mail: {niklas.bergmann, elmar.padilla, jan.bauer}@fkie.fraunhofer.de

Summary: This paper addresses the growing need to protect unmanned systems (UxVs) from forensic analysis, notably in hostile scenarios where systems are captured and reverse-engineered by adversaries. While UxVs play critical roles in both civilian and military applications, their vulnerability to software extraction however poses significant risks, especially in the context of AI-based algorithms used for sensor data analysis. Particularly in military contexts, existing legal protections are insufficient, highlighting the need for technical defense mechanisms. We explore software obfuscation as a complementary solution to safeguard sensitive algorithms and software architecture. The paper provides an overview of current obfuscation techniques and introduces a modular and automated obfuscation framework. Through performance measurements and an initial user study, we assess its costs and effectiveness in preventing forensic analysis. Our findings demonstrate the potential of obfuscation as a robust strategy for enhancing the security of UxVs against adversarial capture.

Keywords: Unmanned Systems (UxVs), Military Drones, Cyber Security, MATE Threat, Code Obfuscation, Reverse Engineering, IT Forensic.

1. Introduction

As unmanned systems continue to develop and become more widespread in various areas of application, the importance of the security and confidentiality of the information in the systems themselves is also growing. These platforms, which are often operated remotely or semi-autonomously over large geographical distances, are not only an integral part of civilian applications, such as in logistics, infrastructure monitoring, and agriculture [1], but also play a key role in military scenarios [2,3]. Unmanned Aerial, Ground, or Underwater Vehicles (UAVs, UGVs, and UUVs) – often summarized with the abbreviation UxVs – are revolutionizing the way vast information is collected by a plurality of sensors and processed for surveillance and reconnaissance missions [2,4].

However, a central and inherent problem for UxVs is the threat scenario in which a system is captured by an adversary during a mission. In such a man-at-the-end (MATE) scenario [5], hostile actors could be able to forensically analyze the system and extract valuable information about its software architecture, software technologies, and sensitive algorithms. This poses enormous risks, as the disclosure of highly specialized software for intelligent sensor data analysis and fusion, especially in the field of artificial intelligence (AI), can lead to decisive strategic disadvantages.

To counter the risk of forensic analysis, protecting the UxV's software is of crucial importance. Legal protection mechanisms such as copyrights or non-disclosure agreements may restrict unauthorized access and use of software by third parties in the civilian sector, but do not provide direct protection against forensic analysis or reverse engineering. However, in a military context, legal protection is not sufficient to ensure protection against capture and analysis by hostile forces. Technical protection measures, on the other hand, offer direct defense mechanisms against unauthorized analyses of software, especially in MATE scenarios in which systems in the field are at risk. In practice, various approaches are used as highlighted in [6,7]. Those contain advanced approaches such as trusted native code or special hardware security modules (HSMs) to access critical data and execute algorithms exclusively in a secure area of the hardware. Server-side execution approaches that protect critical algorithms by outsourcing them to external servers, on the other hand, require a robust and secure data connection that is not available in Denied, Disrupted, Intermittent, and Limited (DDIL) military scenarios with restricted communication.

A complementary approach for the protection of UxVs' software is obfuscation [6-10]. Obfuscation hinders the analysis of binary code by deliberately modifying the structure of the code in such a way that its unaltered functionality is difficult to understand [10]. This not only protects the software from direct analysis, but also the sensitive algorithms and technologies it contains.

The contribution of this paper comprises i) a comprehensive, yet concise overview of obfuscation techniques (**Section 2**), ii) the introduction of our modular, and automated obfuscation framework that has been developed with representative techniques (**Section 3**), and iii) finally, a potential analysis of software obfuscation for the protection against forensic investigations with the help of performance measurements (**Section 4**) and with an initial user study, considering efficiency and costs (**Section 5**).

2. Background & Literature Research

At the beginning of our investigations, extensive literature research was carried out to gain an overview of existing obfuscation methods and to categorize them. In addition to surveying the different obfuscation techniques and scenarios used, particular attention was also paid to the metrics considered to quantify the quality of the respective techniques in the papers.

It resulted in an overview of 21 techniques from the categories that were introduced in [7]: data, logic, and abstraction transformations. This overview is shown in **Table 1** and formed the basis for the subsequent selection of exemplary obfuscation techniques. The names of these three categories refer to the goal of obfuscation. For example, the encryption of literals and values (#1, #8) is assigned to data transformations, while the application of encryption to parts of the functional machine code (#1, #10) falls into the category of logic transformation. The term abstraction transformation, on the other hand, covers methods for concealing logical program units and structural assignments. This includes functions that bundle software functionality into separable units and allow analysts to draw conclusions about the structure of the program.

Technically, obfuscation can be implemented on different *layers* [7]. Some techniques can only be addressed at source code level (*S*), others can also be implemented at intermediate code (compiler level *C*), and some also allow implementation at machine code (binary level *B*). The *target* of the obfuscation also varies. The intention may be either to conceal data *D* (e.g., constants D_C or variables D_V) or algorithms in the code logic C_L or abstraction C_A [6]. The modifiable *unit* ranges in a five-stage gradation according to [8] from individual instructions *1* to the entire program *5,* as highlighted in **Table 1**.

The choice of metrics for the potential analysis is inspired by a survey by Ebad et al. [9], which evaluated how frequently certain classes of metrics were used in scientific papers on obfuscation. Basically, the authors found that 54 % of the papers examined stated the obvious and mandatory *similarity*, i.e., the semantic equivalence, as a metric, while only a third of the papers considered obfuscation *costs* (31 %), e.g., the increased computational effort and program runtime, and their *potency* (30 %), i.e., the impact on the analysis. In contrast, *resilience*, i.e., the resistance of the respective obfuscation technique against recovery, and *stealth*, i.e., a low detection rate, were only considered relevant in 13 % and 12 % of the papers examined, respectively [9].

Taking into account the limited UxV resources and the specifics of MATE attacks, we decided to consider both the *potency* of the individual techniques and their *costs* as the decisive metrics, whereas *resilience* and *stealth* may not necessarily be of primary relevance for all scenarios. Based on our decades of experience in the field of malware analysis (e.g., [12,13]), an expert assessment of the respective techniques regarding the metrics is finally given in **Table 1**.

Table 1. Taxonomy of main obfuscation techniques with an expert assessment of their respective capabilities.

	#	Technique	Description	Layer	Unit	Target	Potency	Costs	Resilience	Stealth
Data	1	Coding of Constants	Use Opaque Predicates (OPs) to decode constant values at runtime.	*S*	*1*	D_C	●	○	◐	○
	2	Increased Complexity	Use of OPs to increase the complexity of the logic structures.	*S*	*3*	C_L	●	○	◐	○
	3	Anti-Disassembly	Prevention of recursive traversal disassembly by OPs in branches.	*C*	*3*	C_L	◐	○	◐	◐
	4	Splitting of Variables	Splitting variable values into several fields to hinder interpretation.	*S*	*1*	D_V	◐	●	●	●
	5	Merging of Variables	Merging variables into one field to conceal variables' relationships.	*S*	*1*	D_V	◐	●	●	◐
	6	Restructuring	Making data structures opaquer, increasing complexity of their use.	*S*	*5*	*D*	◐	○	●	●
	7	Multiple Use of Variables	Splitting fields for multiple variables to hinder flow analysis and obscure variables' semantics.	*C*	*4*	D_V	◐	○	◐	◐
	8	Coding of Literals	Decoding of values at runtime through various transformations.	*C*	*1*	*D*	●	○	◐	◐
	9	**Data Flow Flattening (DFF)**	Management of variable access through management view, which obscures relationships and identities.	***S***	*5*	D_V	●	●	◐	○
Logic	10	Coding of Functions	Encryption of the machine code of entire functions to prevent them from being analyzed.	*C*	*4*	C_L	●	○	◐	◐
	11	Packing	Decryption of original program by a decrypt. stub at start of execution.	*B*	*5*	C_L	◐	◐	○	○
	12	Virtual Architecture	Interpretation of the program on a virtual machine with a coordinated bytecode language.	*C*	*5*	C_L	●	●	●	○
	13	Self-Modifying Code	Modification of functions using edit functions to change their functionalities at runtime.	*C*	*4*	C_L	●	◐	◐	◐
	14	**Control Flow Flattening (CFF)**	Restructuring of a function into a state machine with state variables to disguise the control flow.	***S***	*4*	C_L	●	◐	◐	○
Abstraction	15	**Call Graph Flattening (CGF)**	Function calls through a central switching center to establish connections between functions.	*C*	*5*	C_L	◐	○	◐	○
	16	Function Merging	Use of OPs to decode constant values at runtime.	*S*	*4*	C_A	◐	○	◐	●
	17	Jump Insertion	Use of OPs to increase the complexity of the logic of structures.	*C*	*4*	C_L	○	○	○	○
	18	Garbage Insertion	Preventing recursive traversal disassembly by OPs in branches.	*S*	*	C_L	◐	○	◐	◐
	19	Overlapping Instructions	Splitting variable values into several fields to make it more difficult to interpret the values.	*C*	*2*	C_L	◐	○	◐	●
	20	Stack Pointer	Merging several variables into one field to conceal relationships between the variables.	*C*	*5*	C_A	●	◐	◐	○
	21	Calling Conventions	Making data structures opaquer, increasing their complexity.	C	5	C_L	●	○	○	◐

Legend: **Layer**: *S* source, *C* compiler, *B* binary; **Unit**: 1instruction, 2 basic block, 3 structure, 4 function, 5 program, * all units; **Target**: *D* data, D_C constant, D_V variable, *C* code, C_L logic, C_A abstraction; **Metrics**: ● high, ◐ medium, ○ low.

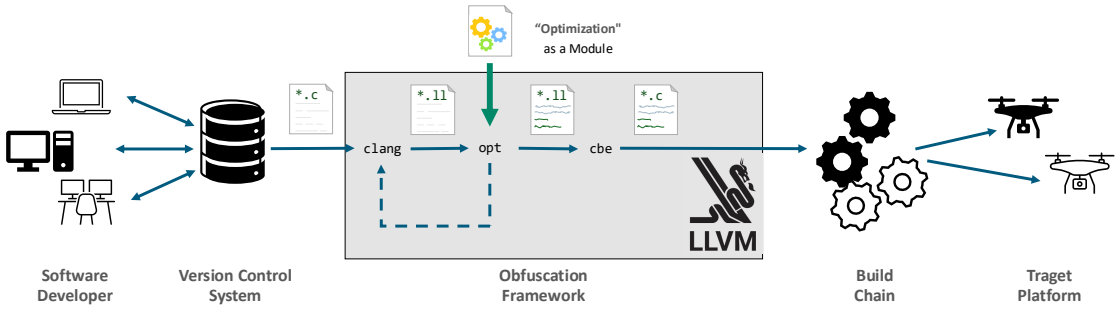

Fig. 1. Integration of the developed LLVM-based obfuscation framework into the existing software development process.

3. LLVM-based Obfuscation Framework

3.1 Conceptualization and Development

UxVs are highly complex cyber-physical systems (CPSs) and their software is typically very heterogeneous in terms of the software technologies, programming languages, and middleware, as well as the tools and compilers used for development. For this reason, a modular approach was chosen to integrate an automated obfuscation into existing build chains and, thus, to enable a language-agnostic framework that is independent of the target architecture. The basic idea is a component upstream of the actual build chain in the software development process, which takes over the obfuscation based on the source code originating from a version control system. It then passes on obfuscated source code to the compilers, as schematized in **Figure 1**. The advantage of this approach is that the obfuscation framework can be seamlessly integrated into any existing C code build chain without any changes. As a proof-of-concept, the framework was implemented based on the open-source compiler infrastructure LLVM [14], since it offers a specific C backend (CBE) [15] that could be exploited for our purpose. Our framework is designed to be very flexible, it is extensible, and it provides, among other things, a seed-based randomization of obfuscation parameters for automatic software obfuscation.

Based on the literature research presented in the previous section, three related techniques from the three categories of logic, data, and abstraction transformations, which cover different aspects relevant for software analysis, were selected for the implementation: *Control Flow Flattening* (CFF), *Data Flow Flattening* (DFF), and *Call Graph Flattening* (CGF), cf. **Table 1**. The term *flattening* hereby refers to the transformation of different graph structures to significantly increase the challenge of their analysis [9]. This is based on the insight that analysts and their tools process the information of these graphs and are optimized for certain patterns in them, which can be effectively obscured by targeted transformations.

The control flow graph is a data structure that is used by most common analysis tools. It reflects the order in which instructions are executed in a function. In this directed graph, nodes represent sets of instructions that are executed one after the other, while the edges describe the relationships between them, see **Figure 2(a)** and **(b)**. Conditional program jumps create branches in the graph and cycles represent program loops. Usually, the degree of branching in graphs of this type is low. In most cases, a function has a unique starting point, from which various paths through the functions can be followed.

The flattening of the control flow refers to a graph transformation in which the original relationships between individual nodes are obscured [10]. If it is still directly apparent which nodes (i.e., instruction sets) can be executed next when analyzing the original control flow graph, this is usually no longer the case after flattening. This is achieved by introducing a central node that acts as an intermediary between all original nodes and, consequently, becomes both the destination of all outgoing edges and the origin of all incoming edges. At runtime, calculations are performed within this block to determine which block would be executed in the original control flow. A resulting flattened graph is visualized as an example in **Figure 2(c)**. Without a precise analysis of the program code within the intermediary block, it cannot be adequately restored.

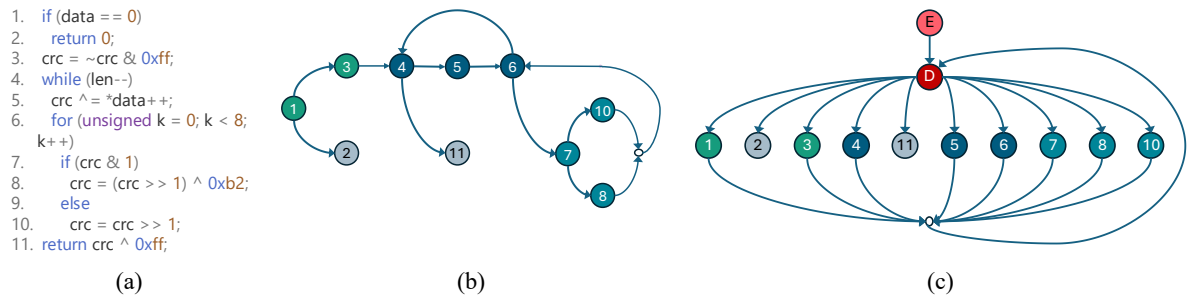

Fig. 2. Schematic representation of the control flow of a simple CRC8 implementation (a) in C code before (b) and after (c) obfuscation by Control Flow Flattening (CFF). Artificial intermediary nodes (i.e., entry E and dispatcher D) obscure the program's original systematic structure. Note that the node numbering corresponds to the line numbers from the source code.

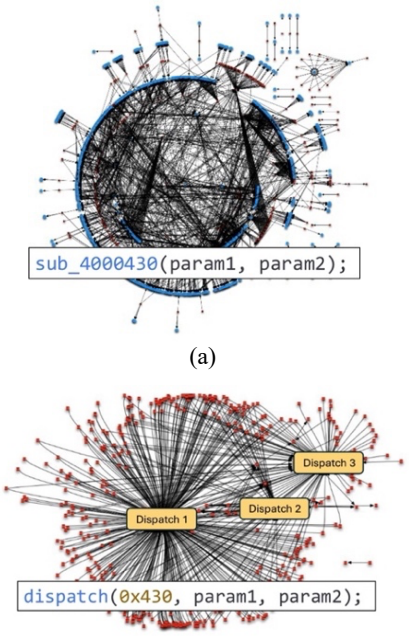

Fig. 3. Representation of the call graph of a program before (a) and after (b) Call Graph Flattening (CGF). For this purpose, three central *dispatcher* functions have been introduced, which bundle all grouped and reciprocal function calls and, thus, conceal the original graph's structure of the program's call graph (graphs adapted from [21]).

Analogous to this approach, the data flow can also be flattened. It describes the mutual dependencies of data and provides a basis for advanced analysis techniques such as symbolic execution and decompilation. When flattening this data flow graph, a targeted attempt is made to make it more difficult to understand the relationships between the data. Essentially, this is done by introducing a separate memory management. For example, reading a variable can be converted into a function call that returns different data values based on the parameters passed. If such functions are used for variable accesses, this forces an analyst to perform a complex examination of these functions.

Similar to the control flow graph, the call graph is a structure that represents the interaction of functions, cf. **Figure 3(a)**. Its analysis makes it possible to identify central functions that are either particularly important for the semantics of a program, because they call many other functions, or its core functionalities, because they are called by many functions. A flattening of this graph is achieved by inserting artificial mediator functions – like CFF – which creates a graph with few central nodes that obscures the original function interactions, cf. **Figure 3(b)**.

Even though these three selected obfuscation techniques are methodologically similar and are all based on a transformation of graphs, their technical implementations and the resulting implications for an analysis process are very different. At the same time, the selection also covers the three main obfuscation categories: CGF obfuscates the functional logic and DFF obfuscates the relationships between data, while CGF obfuscates the functional relationships and thereby addresses the software's abstraction. By varying the number of intermediaries introduced, these flattening techniques can also be scaled in their effect and are therefore particularly suitable for a user study-based potential analysis.

3.2 Discussion of Related Approaches

Pioneering work in the utilization of LLVM was already done by Junod et al. [16] a decade ago. They proposed an LLVM-based obfuscation at intermediate representation code level to protect software against tampering and reverse engineering and offer CFF as a technique, as well. However, the LLVM version they use is obviously very outdated and, according to Kang et al. [17], a core limitation is that users have to build large-scale projects.

Based on Junod et al. [16], there were various developments in the following period. Choi et al. [18], for example, use LLVM as an anti-reverse-engineering technique for Android applications, Garba et al. [19] as a deobfuscation framework to recover the control flow graph of an obfuscated binary function. On the other hand, De la Torre et al. [20] recently used LLVM to investigate a novel obfuscation approach based on genetic algorithms, whereas Kang et al. [17] focused on usability and accessibility of LLVM-based obfuscation and provided a web-based tool for this purpose. In addition to the high-level obfuscation at source code level, which our framework also uses, their tool offers a low-level functionality based on prior decompilation of a binary. Such an approach could also be implemented as an extension for our framework [17].

4. Validation of the Developed Approach

For the technical measurements of the influence of obfuscation on the program runtime, a different functional example was selected for each obfuscation technique, which prominently depicts the respective challenge of obfuscation and represents a stress test scenario. For the CFF evaluation, the CRC8 algorithm was selected, which primarily consists of a loop with several branches, cf. **Figure 2(a)**. It reacts very sensitively to performance losses because such algorithms are designed for processing large amounts of data.

The measurement campaign compares the performance of the *original* program with the obfuscated program automatically generated by the developed framework, i.e., i) with the generated obfuscated binary code (*flat*) and ii) with its *recompiled* version within the build chain modified according to **Figure 1**. Note that the runtime losses that result from the latter version compared to the direct obfuscation *flat* are the price to be paid for our modular approach.

In the measurements that were conducted on a standard desktop PC (Intel i7, 16 GB RAM), the runtimes when processing 10, 100, and 1,000 MB of randomly generated data (averaged over 20 replications) were found to increase by a factor approx. 5 with increasing data sizes, cf. **Figure 4**. It should be noted,

however, that this factor determined in this scenario represents an estimate for a maximum runtime loss. Here, the increase in the runtime is, moreover, linear with the increase in the amount of data to be processed, as can be derived from the figure.

Programs with frequent variable accesses or function calls were used as stress test scenarios for the evaluation of the techniques, DFF and CGF, respectively. In these scenarios, an increase in runtime of only 40 % to 60 % was measured and, thus, a significantly lower loss was observed. Nevertheless, it should be noted that even such losses may not be tolerable under real conditions, especially in the domain of UxVs. Consequently, obfuscation must be introduced into complex software systems with caution. A suitable compromise must be found during development and obfuscation must not be used blindly for performance-critical program functions that are used directly to process large amounts of data. Instead, obfuscation must be applied specifically to individual components of functions. Consideration may also be given to upgrading the hardware of platforms to guarantee the necessary performance of the systems even when obfuscation is used extensively.

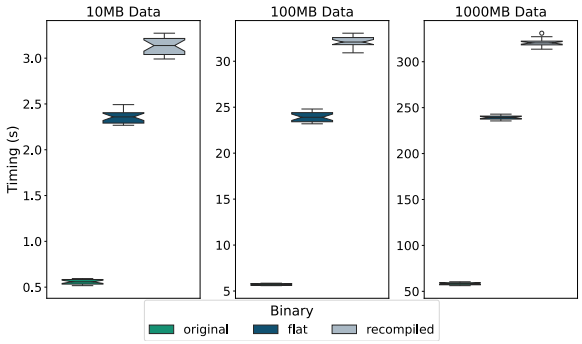

Fig. 4. CCF cost evaluation by CRC8 runtime measurements and comparison of the *original* with the obfuscated *flat* and the obfuscated *recompiled* program in a stress test.

5. User Study-based Potential Analysis

Since the obfuscation of software aims to protect against human expert analysis and reverse engineering, it is – in addition to the technical measurements – necessary to carry out empirical studies to assess the potential of obfuscation techniques, i.e., the impact on the difficulty of software analysis, individually perceived by human subjects. However, conducting such studies is time-consuming and labor-intensive and is therefore rarely practiced. In addition, it is not easy to find suitable experts in order to obtain truly meaningful results. One study worth mentioning in this context is the family of experiments by Ceccato et al. [22], which comprises five studies with obfuscated Java code, each with between 10 and 22 participants from a university background. The authors conclude that simpler techniques can often be more efficient than more complex ones. However, their experiments are limited in terms of the obfuscation techniques investigated. Meanwhile, other studies focus on individual techniques. In [23], for example, the *merging of variables* (#5 in **Table 1**) is examined in a dedicated manner.

In contrast, our user study highlights three complementary techniques that cover the main categories, cf. **Section 2**. It consisted of four blocks and begins with an introduction in which the participants are familiarized with the decompilation of a certain program function. This is followed by three blocks, randomized in order, in which this function is treated after the three obfuscation techniques have been applied. After each of these three blocks with their obfuscation examples, the study participants were asked about the perceived influence of the respective techniques on the analysts' understanding of the respective program function and also on different analysis approaches, such as *static* and *dynamic analysis*. Finally, the participants were also asked to rate the three obfuscation techniques in the form of a ranking and were given the opportunity to make comments in a free text field.

The majority of the 10 participants came from the age groups *20 to 30* and *30 to 40*, with diverse backgrounds in *software development*, *analysis/reverse engineering*, and *obfuscation*, see **Figure 5**. In the study, they spent an average of around 12 min on the individual obfuscated functions and delivered a clear result in their assessment, as shown in **Figure 6**: The impact of CFF was considered to be the most severe. Hence, eight participants ranked CFF first in the final ranking, while no one indicated that this technique had little or no influence on their understanding of the exemplar decompile. In contrast, the influence between CGF and DFF was less clearly perceived. However, both techniques were ranked second (CGF) and third (DFF) by a majority of at least five participants.

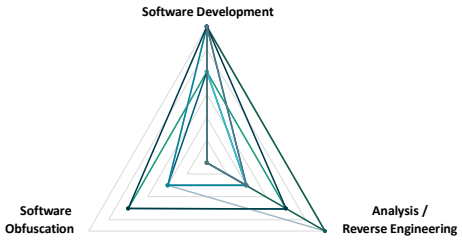

Fig. 5. The participants in the study come from different areas but have specialist knowledge in the field of software development, reverse engineering and/or obfuscation.

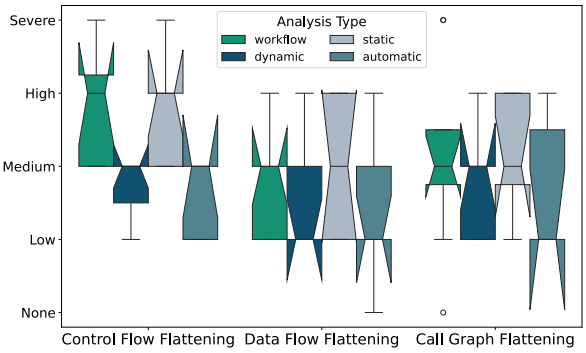

Fig. 6. Extract of the results of the user study conducted to investigate the potential of software obfuscation on the analysis process. The perceived impact of obfuscation on software analysis in the study examples is individual and varies between low and severe.

The comments received in the free text correspond to the ranking provided. The participants unanimously stated that CFF causes them the most difficulties and would mean a considerable effort in the static analysis of real software. Likewise, the fewest approaches on how to methodically deal with a function obfuscated by CFF were found here. Participants with expertise in reverse engineering stated that dynamic analysis of the function would be an option as long as an in-depth analysis of the function was not necessary. The handling of all three obfuscation techniques is a current subject of research, but the literature search conducted in this context (cf. **Section 2**) shows that more scientific publications focus on control flow obfuscation than on the other two techniques.

4. Conclusion & Outlook

As part of this project, a prototypical, modular obfuscation framework was created that can be seamlessly integrated into existing build chains of a software development process. Using three exemplary obfuscation techniques, the feasibility of automated obfuscation was demonstrated and the protection potential of obfuscating selected software examples was evaluated in an initial user study.

The determined effect of obfuscation varies according to the subjective perception of the analyst, but the forensic analysis process was significantly impeded even for small sample programs in the study – despite prior introduction of the analysts to the techniques used. Furthermore, the costs caused by obfuscation were examined in a measurement campaign regarding program performance. It was shown that these costs are by no means negligible but can be scaled by the degree of obfuscation. It was therefore concluded that it is advisable to use obfuscation only for areas that are worth protecting.

Follow-up activities in software obfuscation envisage an expansion of the portfolio of available obfuscation techniques in the developed framework. In addition, the user study is to be extended in terms of its scope and number of participants. For the partial use of obfuscation, its detection rate will also be investigated, i.e., its ability to avoid exposing protected areas of code to analysts through forensically inconspicuous modifications. Further exciting fields of research arise on the one hand from a legal consideration regarding the certification of randomized obfuscated software and, on the other hand, from AI-supported obfuscation as well as investigations of the resilience of obfuscated software through AI-based software analysis.

References

[1]. M. Mammarella, L. Comba, A. Biglia, F. Dabbene, and P.Gay. “Cooperation of Unmanned Systems for Agricultural Applications: A theoretical framework.” *Biosystems Engineering* 223 (2022).

[2]. S.G. Gupta, M. Ghonge, and P.M. Jawandhiya. “Review of Unmanned Aircraft System (UAS).” Int. *Journal of Advanced Research in Computer Engineering & Technology* (IJARCET) 2 (2013).

[3]. R.K. Barnhart, D.M. Marshall, and E. Shappee (Eds.). “Introduction to Unmanned Aircraft Systems.” *CRC Press*. (2021).

[4]. J. Li, G. Zhang, C. Jiang, and W. Zhang. “A survey of maritime unmanned search system: Theory, applications and future directions.” *Ocean Engineering* 285 (2023).

[5]. P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski, “Guest Editors' Introduction: Software Protection.” *IEEE Software* 28(2) (2011).

[6]. C. Collberg, C.D. Thomborson, and D. Low. “A Taxonomy of Obfuscatiing Transformations.” *Tech. Report #148*, University of Auckland, 1997.

[7]. S. Banescu and A. Pretschner. “A Tutorial on Software Obfuscation.” *Advances in Computers* 108 (2018).

[8]. P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. “Sok: Automated Software Diversity.” In Proc. of the *Symposium on Security and Privacy (S&P),* Berkeley, CA, USA (2014).

[9]. S.A. Ebad, A.A. Darem, and J.H. Abawajy. “Measuring Software Obfuscation Quality – A Systematic Literature Review.” *IEEE Access* 9 (2021).

[10]. S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik und E. Weippl. “Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis?” *ACM Computing Survey (CSUR)* 49.1 (2016).

[11]. N. Kuzurin, A. Shokurov, N.P. Varnovsky, and V. Zakharov. “On the Concept of Software Obfuscation in Computer Security.” In Proc. of the *Information Security Conf. (ISC)*, Valparaíso, Chile (2007).

[12]. K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, “Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study.” In Proc. of the IEEE *Symposium on Security and Privacy (SP),* San Jose, CA, USA (2016).

[13]. K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, “No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations.” In Proc. of the *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA (2015).

[14]. The LLVM Compiler Infrastructure. Website: *https://www.llvm.org* (accessed on 2025/01/09).

[15]. JuliaHubOSS. “llvm-cbe” Website: *https://github.com/JuliaHubOSS/llvm-cbe* (accessed on 2025/01/09).

[16]. P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. “Obfuscator-LLVM — Software Protection for the Masses.”, In Proc. of the Int. *Workshop on Software Protection (SPRO)*, Florence, Italy (2015).

[17]. S. Kang, S. Lee, Y. Kim, S.-K. Mok, and E.-S. Cho. “OBFUS: An Obfuscation Tool for Software Copyright and Vulnerability Protection.” In Proc. of the *Conf. on Data and Application Security and Privacy (CODASPY)*, Virtual (2021).

[18]. K. Lim, J. Jeong, S. Cho, J. Choi, M. Park, S. Han, and S. Jhang. “An Anti-Reverse Engineering Technique using Native code and Obfuscator-LLVM for Android Applications.” In Proc. of the *Conf. on Research in Adaptive and Convergent Systems (RACS)*, Krakow, Poland (2017).

[19]. P. Garba and M. Favaro. “SATURN – Software Deobfuscation Framework Based On LLVM.” In Proc. of the Int. *Workshop on Software Protection (SPRO)*, London, United Kingdom (2019).

[20]. J.C. de la Torre, J. Jareño, J.M. Aragón-Jurado, S. Varrette, B. Dorronsoro. “Source code obfuscation with genetic algorithms using LLVM code optimizations.” *Logic Journal of the IGPL*, jzae069 (2024).

[21]. J. Duart, Zynamics GmbH “Introduction to mobile reversing” In: CodeGate 2k10 (2010). Website: *https://blog.zynamics.com* (accessed on 2025/01/09).

[22]. M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. “A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques.” *Empirical Software Engineering* 19 (2014). [22].

[23]. A. Viticchié, L. Regano, M. Torchiano, C. Basile, M. Ceccato, and P. Tonella. “Assessment of Source Code Obfuscation Techniques.” In Proc. of the *Working Conf. on Source Code Analysis and Manipulation (SCAM)*, Raleigh, NC, USA (2016).