Diversification and Obfuscation Techniques for Software Security: a Systematic Literature Review

Citation

Year
2018

Version
Publisher’s PDF (version of record)

Link to publication
TUTCRIS Portal (http://www.tut.fi/tutcris)

Published in
Information and Software Technology

DOI
10.1016/j.infsof.2018.07.007

Copyright
This work is licensed under a Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/

Take down policy
If you believe that this document breaches copyright, please contact cris.tau@tuni.fi, and we will remove access to the work immediately and investigate your claim.
Diversification and obfuscation techniques for software security: A systematic literature review

Shohreh Hosseinzadeh*,a, Sampsa Rautia, Samuel Laurèna, Jari-Matti Mäkeläa, Johannes Holvitiea, Sami Hyrynsalmib, Ville Leppänen

a Department of Future Technologies, University of Turku, Vesilinnantie 5, Turku 20500, Finland
b Laboratory of Pervasive Computing, Tampere University of Technology, Pohjoisrantat 11 A, Pori 28100, Finland

ARTICLE INFO

Keywords:
- Diversification
- Obfuscation
- Software security
- Systematic literature review

ABSTRACT

Context: Diversification and obfuscation are promising techniques for securing software and protecting computers from harmful malware. The goal of these techniques is not removing the security holes, but making it difficult for the attacker to exploit security vulnerabilities and perform successful attacks.

Objective: There is an increasing body of research on the use of diversification and obfuscation techniques for improving software security; however, the overall view is scattered and the terminology is unstructured. Therefore, a coherent review gives a clear statement of state-of-the-art, normalizes the ongoing discussion and provides baselines for future research.

Method: In this paper, systematic literature review is used as the method of the study to select the studies that discuss diversification/obfuscation techniques for improving software security. We present the process of data collection, analysis of data, and report the results.

Results: As the result of the systematic search, we collected 357 articles relevant to the topic of our interest, published between the years 1993 and 2017. We studied the collected articles, analyzed the extracted data from them, presented classification of the data, and enlightened the research gaps.

Conclusion: The two techniques have been extensively used for various security purposes and impeding various types of security attacks. There exist many different techniques to obfuscate/diversify programs, each of which targets different parts of the programs and is applied at different phases of software development lifecycle. Moreover, we pinpoint the research gaps in this field, for instance that there are still various execution environments that could benefit from these two techniques, including cloud computing, Internet of Things (IoT), and trusted computing. We also present some potential ideas on applying the techniques on the discussed environments.

1. Introduction

In most organizations, information is a key asset that comes in the form of, for example, financial information, client data, and product design data. Intentional or accidental leakage of any of this information exposes both the business and the customers. Therefore, it is highly significant for any business to have security strategies for protecting the information and services and ensuring the confidentiality, integrity, and availability of the information.

Computer security assures that the system functions under the expected circumstances, and prevents undesired behavior. Many security breaches begin with identifying and exploiting the vulnerabilities in the system. Vulnerabilities are the defects that occur in the process of design and implementation of the software. Defects in design are known as flaws, and the defects in implementation are known as bugs. To ensure the security of software, we need to prevent or mitigate the risk of software vulnerabilities. In other words, we should either eliminate these bugs and flaws, or make it harder to exploit them.

In this paper, we focus on making exploitation of vulnerabilities harder, and reducing the possible damage of the attack. To this end, we center our research around two software security techniques, diversification and obfuscation.

Code obfuscation is the process of scrambling the code and making it unintelligible (but still functional), in order to make reverse...
engineering more difficult [1]. The transformed code is functionally and semantically equivalent to the original code, but is more complicated and harder to comprehend [33]. With the help of code obfuscation, even if adversaries get access to source code, analysis of the code and finding the vulnerabilities will no longer be a simple task. This requires more time and energy and makes the reverse engineering of the code harder and more costly. Obfuscation does not guarantee that the program is not tampered/reverse engineered, but adds an additional level of defence by increasing the effort and cost for an attacker to learn the underlying functionality of the protected program. Various obfuscation techniques exist that obfuscate different parts of the code at different phases of software development process. For instance, using opaque predicates [75] is a common way of obfuscating the control flow of a program, at source code [109] or binary code level [247], at implementation [109] or compile-time [17].

Software diversification refers to changing the internal interfaces and structure of the software to generate unique diversified versions of it. The users receive unique instances of the software that all function the same, although differently diversified. In other words, diversification breaks the "monoculturalism" and introduces "multiculturalism" in the software deployment process.

Malware (malicious software) is any software that intends to run its code on user’s computer to disrupt the computer’s operation or manipulate the system towards the attacker’s desire [2]. To do this, it needs knowledge on how to interact with environment and access the resources. Software diversification alters the internal interfaces of the software and makes it challenging for malware to gain this knowledge. Thus, malware becomes incompatible with the environment and eventually becomes unable to take effective actions to harm the system. It should however be noted that, in order to maintain the access of legitimate applications to resources, we need to propagate the changes to trusted applications, i.e., they will be diversified as well to be compatible with inner layers.

Diversification does not attempt to eliminate the vulnerabilities of a software, but tries to avoid or at least make it too time-consuming for malware to exploit them and perform a successful attack. In a worst-case scenario, even if the malware succeeds in running its malicious code and attack a computer, this attack can only work on that particular computer. The designed attack model does not work on other computers, since their software are diversified differently with different diversification secrets. To take a large number of computers under control, different attack models should be designed specifically for each software instance, which makes it an expensive and arduous task for the attacker. On that account, diversification can be considered as an outstanding approach for securing largely-distributed systems, and mitigating the risk of massive-scale attacks.

It is worthwhile mentioning that the terms obfuscation and diversification, sometimes, have been used interchangeably in the literature. In this paper, we make a clear distinction between these two concepts.

1.1. Method of the study

The method of study we chose in this research is Systematic Literature Review (SLR). A SLR is a means of research that identifies, evaluates and interprets all high quality studies related to a particular research question, or an area of interest [3]. This method of study was originally used in medical sciences [4], but later gained interest in other fields as well. A systematic review can improve a traditional review [4], in a way that the set of studies is not restricted to better-known and frequently-cited publications, and not biased towards the research area/interest of the researcher, as all studies in the field are captured. A systematic review, by classifying and mapping the scattered research studies, identifies research gaps and produce baselines for future research.

We conducted a SLR on studies that deal with the two techniques, obfuscation and diversification, with the aim of securing the code and software. There have been previously some other reviews [5,6,248]. However, they (a) cover a more limited number of studies (14, 69, and 10 papers respectively), (b) consider these two mechanisms from other perspectives than security, (c) focus on one of these two mechanisms, or (d) discuss only one particular technique.

The surveys studying the obfuscation related studies include a review on control-flow obfuscation techniques [6], and a review on code obfuscation approaches [5]. These research works cover less than 15 studies and are published, respectively, in 2005 and 2006, which implies that the studies published after that are missing. Larsen et al. [248] authored a survey that reviews the state-of-the-art in automated software diversity with the aim of security and privacy. Another recent literature review on software diversification, surveyed by Baudry et al. [284], investigates diversification from five various perspectives aimed at different goals, including fault tolerance, security, testing, and reusability.

The main factors that differentiate our survey from the existing ones, are: (1) the systematic process for collecting the data, (2) a thorough list of covered studies on both obfuscation and diversification, (3) the focused scope of the study (security), and (4) classification and analysis of the collected studies.

1.2. Structure of the study

The remainder of this paper is structured as follows: Section 2 discusses the aim of our study, and specifies the research questions we have formulated and addressed in this research. Section 3 reports the process of search and selection of the relevant studies, and also the data extraction from these papers. Section 4 presents the results of the data collection and analysis of the results. In Section 5, we present the discussion. Limitations of the study, concluding remarks, and the future work come in Section 6.

2. Aims and research questions

We undertook a SLR of the papers reporting the use of obfuscation and diversification techniques in software security domain. Before starting the search, we determined the research questions, and formed the search strings. Our SLR addresses the following research questions:

- **RQ1:** What is the aim of obfuscation/diversification being used?
- **RQ2:** What is the status of this field of study? (E.g., outputs per annum, types of studies reported, collaboration of academia and industry)
- **RQ3** In what environments the techniques are used/studied in order to boost the security (i.e., the programming language and execution environment the techniques are used for).
- **RQ4:** What mechanisms have been proposed/studied? (i.e., the obfuscation/diversification method used, (b) target of transformation, (c) level and stage, (d) cost and effectiveness of the approach.

3. Search and selection process

In order to carry out the research review systematically, we need to follow a protocol that defines the search strings and strategy, inclusion and exclusion criteria, and methods to extract data and synthesis the results. In this regard, we based our SLR on the research protocol suggested by Kitchenham et al. [7], and conducted our SLR in seven different phases. These phases are as follow: search and selection process (Phase I), inclusion and exclusion (Phase II to IV), snowballing (Phase V), data extraction (Phase VI), data analysis (VII). Fig. 1 illustrates the different phases in this process. The numbers on the arrows indicate the number of search results and included papers after each phase. In what follows, the details of the protocol developed for our SLR are presented.
3.1. Search

3.1.1. Initial search

Before starting the search process, we conducted an initial search to assure that there are sufficient numbers of articles available in the target field to study. In this stage, we found 48 articles discussing the improvement of software security using diversification/obfuscation techniques, which confirmed that this could be an appropriate topic to conduct a SLR on.

3.1.2. Manual search

For the manual search, Phase I, we selected a set of proper search strings, with which we assumed we would find the majority of the related articles. We also selected six of the largest digital databases, including IEEEXplore Digital Library, ACM Digital Library, Wiley online library, ScienceDirect, dblp, and SpringerLink. We limited our search to titles, abstracts and keywords of the articles to avoid false positive results of the full-text search. In some cases, search query was adapted according to requirements of the search engine. The following search command was used to retrieve studies from the databases:

```
(software OR code OR program) AND (diversification OR obfuscation OR obfuscate OR obfuscator)
```

We undertook the manual search separately in the databases and combined the results in a large spreadsheet. After removing the duplicates, 6040 articles proceeded to Phase II for inclusion and exclusion (Section 3.2).

3.1.3. Automatic (citation-based) search

To complete the manual search, we performed an automatic search (backward snowballing), in Phase V. Backward snowballing is done by analyzing the reference lists of selected papers to find any missing related paper [7]. Therein, 268 papers were collected, for which we repeated the inclusion/exclusion process (Phase II to IV).

3.2. Selection of the studies

After collecting the papers in Phase I, we should include relevant and drop irrelevant papers. For that, we defined some inclusion/exclusion criteria, based on which we make decision (in Phase II-IV) whether to include/drop a paper. The followings are the inclusion criteria in our study:

- papers that are written in the English language;
- peer-reviewed papers (however, we did not exclude technical reports and books, since there exists some widely cited high quality technical reports in this domain, e.g., [13]);
- papers in the context of software production/development;
- papers related to software security;
- papers related to obfuscation/diversification; and
- obfuscation/diversification in the paper is used/discussed with the aim of improving/enhancing the security in software/code/program.

Considering that obfuscation and diversification techniques have been used in different domains for various purposes, we decided to narrow down our results. To this end, we focused our search on studies that are using obfuscation/diversification with the aim of software security and leave out the papers that were falling in our exclusion criteria:

- studying the possibility/impossibility of obfuscation;
- studying the use of obfuscation/diversification by malware, to hide their malicious code from scanners and malware analyzers;
- studying the techniques at a level other than software (e.g., hardware/network);
- proposing an approach that needs hardware assistance;
- studying obfuscation/diversification from cryptographic point of view;
- using the approaches to protect software watermark, birthmark and intellectual property rights; and
- unavailable studies, that we were not able to access in anyway.

Considering the defined criteria, we followed this process to select the relevant studies:

1. In Phase II, we screened the papers based on their titles. Each paper title was checked by four authors to determine whether it is relevant to our study or not, according to the defined inclusion/exclusion criteria.
2. In Phase III, two of the authors screened the papers based on their abstracts, and included the papers that were compatible with the inclusion criteria and dropped the papers that were not.
3. In Phase IV, the same process was repeated as Phase III, but based on the full text of the papers this time. There were several cases in which the full texts were not available in online databases. We tried...
to contact the author(s) or find the text from other sources. If we were not successful finding the text in any way, we dropped the paper.

3.3. Data extraction

Each of the 357 selected papers was read through by two reviewers. The first reviewer extracted the data from the papers using a data extraction form, and the second reviewer checked the correctness of the extracted data. In case of any disagreement, the paper was discussed in a meeting with other authors, till reaching an agreement.

We divided the papers into two main categories, Constructive and Empirical, and defined different sets of questions to extract data from them. The papers that propose a new (implementable) obfuscation/diversification method, or apply/implement a technique fall into the category of constructive papers. The papers that evaluate/assess/experiment/discuss/review some (existing) obfuscation/diversification techniques fall into the category of empirical papers. There also exist papers that could be considered as both constructive and empirical. This class includes the papers that carry out an empirical study and at the same time conduct a constructive work.

For the category of constructive papers we extracted the following data, and presented the classification of the captured data in Section 4.1:

- **Aim:** For what purpose is obfuscation/diversification used and what types of software security problems is solved (e.g., what type of attack is mitigated)?
- **Level:** At what level is obfuscation/diversification applied (e.g., source code, binary level)?
- **Stage:** At what stage of software production is obfuscation/diversification applied (e.g., compile-time, run-time)?
- **Target:** What is the subject of obfuscation/diversification transformation (e.g., control flow)?
- **Mechanism:** What type of obfuscation/diversification method is used/proposed?
- **Language:** What language is the paper targeting?
- **Execution environment:** What environment is the obfuscation/diversification technique introduced?
- **Resiliency:** How has the resiliency of the proposed approach been tested, and what results have been achieved?
- **Overhead:** What kind of overhead does the proposed obfuscation/diversification technique introduce?

For the category of empirical papers, we extracted the following data, and presented the classification of the captured data in Section 4.2:

- **Relevance:** How is the paper related to obfuscation/diversification?
- **Outcome:** What are the outcomes/findings/results of the study?

4. Results

As mentioned before, based on the method of the study used, we divided the selected studies into three main categories of (a) constructive, (b) empirical, and (c) constructive and empirical. Fig. 2 shows the distribution and the number of papers in each category. As is seen, the highest interest has been on constructive methods and obfuscation studies.

4.1. Constructive studies

By analyzing the data we captured from the data extraction phase, we answered the research questions defined in the beginning of our study.

4.1.1. RQ1: Status of the field of study

After the search and selection step, we extracted data from the 357 included studies. The studies come in six different types, including conference paper, journal article, workshop papers, book section, technical report, and doctoral theses. Also, there were 2 studies in other formats that did not fit into these categories. Table 1 shows different types of studies and the number of studies found in each type. The numbers indicate that the majority of the studies were published in conferences.

We analyzed the author affiliations for the included papers to associate the papers to their originating organizations and countries.

Fig. 3 captures the ten most associated countries for the considered set of studies. United States has by far the largest (c. 39, 5%) share, followed by China (c. 10,1%). However, as a continent, UK and Europe lead the statistics (c. 40,1%), with research divided mainly among Germany, Belgium, and Italy. The list also includes Japan and India – Asia as a whole contributed to one third (c. 32,2%) of the papers in the study. The research is relatively concentrated to a selected number of regions as the five and ten most affiliated countries count for circa 60,8% and 80,1% of all the affiliations.

Fig. 4 captures the ten most associated organizations for the considered set of studies. From this, we note that Microsoft Corporation (inclusive of Microsoft Research) is the only non-academic organization to be prolific in this area. Further, the ten most prolific organizations correspond to almost a third (c. 29, 1%) of the total affiliations for these studies. This is a notable portion from the affiliations, and arguably, indicates that majority of the research is concentrated to a rather small set of organizations. In Belgium, Finland, and New Zealand, the majority of research can be traced to a single organization.

It was of our interest to know the annual growth and decrease rates of the publications in this field of study. This can indicate the changes in interests and the significance of the field of study. An upward trend can be a sign of increasing interest to the field; while, a downward trend could state that the field is reaching a dead end. Fig. 5 illustrates the distribution of the selected studies in the SLR, between the years of 1993 to 2017. There is a relative fluctuation in the whole period, with an overall upward trend in the number of published studies, except for the slight decline in 2017. This implies that while the field has been fairly unpopular research subject, it has recently drawn fair attention among researchers. Between obfuscation and diversification, the former has almost always been a more popular technique – significantly so between 2000–2010, while diversification has gained in popularity since then.

We also examined the articles’ publication forum types as a function of their publication years and the distribution is captured in Fig. 6. We note that through the queried year span, the dominant publication forum type is conference. However, the type selection gets more varied as we approach the present day, and as a publication forum, the journal type is almost on par with the conference in the year 2014. The observed increase in variety could be taken as evidence for the domain getting more mature: existence of more established research in the domain shows as increase in the number of journal articles and book chapters while the discovery of new sub-domains shows as an increasing number of workshop publications.

Fig. 7 displays, for the considered set of studies, the associated organizations’ sector as a function of the publication year. Observations made here relate closely to the ones made for Fig. 4: while some publications are affiliated solely to industrial organizations (c. 2, 6% publications in the year 2015 and c. 5, 6% in total for the considered time-span) or to both industrial and academic bodies (c. 13, 2% in the year 2015 and c. 12, 6% in total), majority of the considered studies are made in an academic vacuum. While the distribution is understandable for theoretical research, it raises concerns regarding the applicability and correspondence of the research in this domain.

4.1.2. RQ2: Aim

In the reviewed literature, we identified a set of aims for which obfuscation and diversification were used for securing code and software and defeating known attacks, and hopefully unknown future attacks [238]. In Table 2, we summarize the generic aims that the related
studies were following.

In the process of reviewing the selected studies, we identified four broad categories that could encompass most of the presented literature. We acknowledge that these categories are not completely orthogonal, that is, there is some overlap between the different categories and a single piece of research could reasonably be classified as belonging to multiple categories. Still, being aware of the common aims or use cases associated with obfuscation and diversification research can be a valuable resource. With this classification, we try to answer the question what real-world problems are being solved by the use of diversification and obfuscation methods.

a) Making reverse engineering of the program logic more difficult: The most commonly stated aim of this research area was simply to make malicious reverse engineering of programs harder [113,165,171,277], i.e., making the act of debugging and disassembling of the software more complex to get the original source code [71,91,123,198,247]. By reducing the readability and understandability [47,110] of the software through these techniques, it becomes more resistant to unauthorized modification, i.e., becomes more tamper-proof [25]. Making understanding programs harder might be a desirable aim in order to protect proprietary algorithms or other intellectual property. Assembly code obfuscation [211], increasing complexity of dynamic analysis [240], preventing control-flow analysis [75], and introducing parallelism in order to obfuscate control-flows [239] are examples of research aiming to make programs harder to understand. Furthermore, obfuscation is an effective approach to counter both static [60,226,268] and dynamic analysis [122,126,240].

b) Prevent widespread vulnerabilities: Obfuscation and diversification techniques were also employed for their potential security benefits in preventing widespread vulnerabilities [81,262,268]. Exploits often depend on minute details about program internals. Introducing diversity into deployed applications can make it more challenging to construct exploits that reliably work against multiple targets. Diversification works by introducing variability in the software. Increased diversity makes the number of assumptions an adversary can make about the system smaller. Aside from diversification, obfuscation can also serve as a method of making software more secure. By making it more challenging for an attacker to understand the piece of software, obfuscation helps to increase the costs associated with exploit development. Examples of research specifically targeting security include randomization measures to defeat Return-Oriented Programming (ROP) attacks [216], randomized instruction set emulation [66], metamorphic code generation [230], and diversifying system call interface to defeat code injection attacks [159,233,282].

c) Preventing unauthorized modification of software: Research on tamper-resistance tries to find ways for making it more challenging for an adversary to produce derived version of programs [26,107,127].

---

Table 1
Types of studies.

<table>
<thead>
<tr>
<th>Type</th>
<th>Diversification</th>
<th>Obfuscation</th>
<th>Both</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conference paper</td>
<td>68</td>
<td>134</td>
<td>7</td>
<td>209</td>
</tr>
<tr>
<td>Journal article</td>
<td>29</td>
<td>51</td>
<td>2</td>
<td>82</td>
</tr>
<tr>
<td>Workshop paper</td>
<td>12</td>
<td>17</td>
<td>1</td>
<td>30</td>
</tr>
<tr>
<td>Book section</td>
<td>10</td>
<td>8</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>Technical report</td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>Doctoral Thesis</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Other</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>122</td>
<td>223</td>
<td>12</td>
<td>357</td>
</tr>
</tbody>
</table>

---

Fig. 2. Distribution of the studies.

Fig. 3. Prolific countries: ten most associated countries in the considered studies (total number of country level affiliations $N=420$).
This might be desirable in order to preserve the intended operation of a program in an uncontrolled environment. For example, applications employing some form of digital rights management or computer games trying to prevent players from cheating might employ such techniques in order to make it harder to circumvent the protection mechanisms [30,259]. Techniques aiming for tamper-resistance often utilize methods for making understanding the program more difficult but they can also include methods for verifying program authenticity. Tamper-resistance was explicitly mentioned as one of the aims in the context of obfuscating Java bytecode [30], run-time randomization in order to slow down the adversary’s locate-alter-test cycle [103] and obfuscation of sequential program control-flow [24]. Control flow obfuscation conceals the real control flows of the program and generates a fake control flow [145,175]. This makes it difficult for an analyzer to comprehend the logic of the program [245], also prevents spying and manipulating the control flow [75].

d) Hiding data: Aside from making programs more complex to analyze, obfuscation was also utilized for hiding static non-executable data within programs [99,231,281]. Hiding cryptographic keys and protecting intellectual property are few examples of scenarios were such measures are considered. Such techniques have been used to hide static integers [138,191] and obfuscate arrays by splitting them [97].

The results signify that the two techniques are used to mitigate the risk of a wide range of attacks, and in best case scenario hamper them. Table 3 presents the top attacks that were impeded with the help of obfuscation and diversification, such as code injection attacks [55,105,108,197], ROP attacks [195,215,260,263], buffer over-flow attacks [35,57,268], and Just-in-Time (JIT) spraying attacks [186,208,263]. From Table 3 we can deduce that not all the studies (209 papers) were explicitly discussing particular attacks that they aim to impede.

4.1.3. RQ3: Environment

For classifying the environments, two subcategories were chosen: a) language of the program being obfuscated/diversified and b) execution environment.

a) Language: The reviewed literature used a diverse set of over 20 different programming languages. Circa 36.8% of the languages were the topic of only one research and two thirds (63.1%) were mentioned at most thrice. Most research discussed one (c. 63.4%) or two (c. 10.6%) specific languages, with two systems programming (C/C++) or high level languages (Java & JavaScript) representing the vast majority of such pairs. A quarter (25.0%) of the research did not specify a single language or generalized the presented work for a class of languages. Only a minority of research [135,158,163,167,191,232,340] mentioned multiple languages or language classes.

A more descriptive view of the kinds of the languages was achieved by further classifying the research into four language categories representing hardware oriented, high level, scripting, and domain specific languages. The distribution of languages into these languages is as follows:

- Systems programming (N=158): C (52), Assembly (29), C++ (21), Cobol (1)
- Managed (N=81): Java (54), C# (3), Haskell (2), J# (1), Lisp (1), OCaml (1), VB (1)
- Scripting (N=19): JavaScript (11), Python (3), Perl (2), PHP (1)
- Domain specific, DSL (N=7): SQL (5), HTML (1)

The systems programming languages are compiled to native hardware without a run-time virtual machine and provide direct access to memory. Due to this low level direct hardware access, these languages benefit from obfuscation and diversification to protect this access. Some

![Fig. 4. Prolific organizations: ten most associated organizations for the considered set of studies (total number of organization level affiliations N = 544).](image)

![Fig. 5. Number of papers published yearly on the topic of security and privacy through obfuscation/diversification.](image)
examples of the applications of these languages in the research include operating systems and drivers, low-level libraries, server software, high performance computing, and embedded software. The managed languages typically require a virtual machine to provide a safer programming model for application programming. The most common problem for these languages is that the code is relatively easy to reverse engineer. The Java virtual machine is the most common platform in the selected research studies, but others, such as the Microsoft’s Common Language Runtime (CLR), were also covered. A typical application of this class is mobile code, that is, code expected to run in an unknown environment. Finally, the scripting languages introduce new levels of insecurity since manipulating their code is even simpler. The DSLs have other issues, for example injection attacks or the need to protect intellectual property.

The following three figures present the language trends in the reviewed papers. First, Fig. 8 shows the popularity of various language categories based on our classification. The majority of the research has focused on systems programming, managed languages come as the second most popular category. Script languages are a bit more researched than DSLs. Fig. 9 shows the overall distribution of language popularity in selected studies. A raw binary code (of native or virtual machine bytecode instructions) is the most popular "language" in this field of research. This is natural as most software is compiled to binary form for distribution. It represent the lowest level language and often requires disassembly to reconstruct the program structure for analysis. We distinguish assembly language as a separate form with its structured form intact for further analysis. Assembly is commonly used when obfuscation/diversification is used as a language agnostic compiler pass. The C and Java languages are other popular choices, followed by C++ and JavaScript.

Fig. 10 shows the trend over time for the five most used languages. The other languages are presented as the sixth group, as a reference. Like in the other figures, the research seems to be a bit more active in the 2000s and even more active in 2010s. Each of the top five languages appears to be almost equally represented each year.

b) Execution Environment: The environments in the reviewed literature can be classified in various ways as there are many interesting areas of focus. We have focused on two approaches in our review. First, the target environment of deployment (Table 4) plays a significant role when analyzing the applicability of a security mechanism. The majority of reviewed approaches are general enough to work in a multitude of environments. The most significant group of special environments were distributed and agent based systems with mobile code. As the code executes in a possibly remote, uncontrolled system, the need for protection is obvious - especially since the mobile agents often rely on bytecode that is relatively easy to reverse engineer. Virtualization and cloud computing can introduce similar kinds of problems if the host is owned by a third party, but virtualization is also used as a protection mechanism. Web services and servers offer an attack surface via the service layers, and mobile and desktop users are threatened by unreliable software. We distinguish between generic servers and cloud by denoting XaaS platforms for hosting third party services as the cloud. Embedded environment might use obfuscation or diversification for example to avoid the computational cost of encryption. Furthermore, most mobile devices are embedded platforms, but not all embedded platforms are mobile.

The second way to classify the reviewed literature is by the run-time environment (Table 4). This classification focuses on the abstraction level on the deployed software stack, with native code on the bottom and the virtual machine managed code on top, if both run-times are being used. Over a half of the research targets a native code environment. The more specific mechanisms are further discussed in the level
(Section 4.1.4b) and stage (Section 4.1.4c) sections. Around fifth of the research focuses on managed environments such as Java virtual machines. Few papers target both environments, e.g., in the case of JIT compilers. Almost a third of the research claims to operate in all kinds of environments as a general purpose security mechanism.

4.1.4. RQ4: Mechanism

a) Method: In order to make diversified program instances, various transformation mechanisms are proposed in the literature. Each of these mechanisms are applied at different stages and levels of software development life-cycle (discussed in Section 4.1.4.b and Section 4.1.4.c). In this section, we classify the transformation techniques, based on the target of transformation. In other words, “what” is transformed and “how” the transformation is applied. Fig. 11 illustrates these techniques as a tree. On first level of the tree come the targets of transformations and on the lower levels the transformation techniques to obfuscate these targets. We base our classification on the taxonomy presented by Collberg et al. [13], which introduces control obfuscation, data obfuscation, layout obfuscation, and preventive obfuscation as different transformation targets. In the following we discuss each category.

- **Control flow obfuscation** aims at altering/obscuring the flow of a program to make it difficult for an attacker to successfully analyze and understand the code [1]. There exists a large body of research on control flow obfuscation techniques [259,261,266,334]. The most common technique to disturb the control flow is bogus insertion [11,12,22,31,63,73,104,268,362]. This technique works as inserting gray/dead/dummy code [351] that is never executed, fakes the control transfer [100], and/or introduces confusion for the analyzing tools [16,24,45,51,98,109,129,145,179,187] to attain the actual flow. Adding dummy blocks [122,160,169], dead statements [170], redundant operands [113], dummy instructions to camouflage the original instructions [38,242], new segments [247], dummy classes [84], dummy sequence using dead registers [47], and junk byte insertion to instruction stream [34,169], all fall into this class of transformation. Inserting additional NOP instructions [215,226,283] is another type of bogus insertion. NOPs are instructions that perform no operations but make it harder to predict where the pieces of code are placed in memory.

- **Code injection attacks** [16,24,45,51,98,109,129,145,179,187] to attain the actual flow. Adding dummy blocks [122,160,169], dead statements [170], redundant operands [113], dummy instructions to camouflage the original instructions [38,242], new segments [247], dummy classes [84], dummy sequence using dead registers [47], and junk byte insertion to instruction stream [34,169], all fall into this class of transformation. Inserting additional NOP instructions [215,226,283] is another type of bogus insertion. NOPs are instructions that perform no operations but make it harder to predict where the pieces of code are placed in memory.

Another widely used technique for disturbing the program’s control flow is using opaque predicates [16,24,45,51,98,109,129,145,179,187] to attain the actual flow. Adding dummy blocks [122,160,169], dead statements [170], redundant operands [113], dummy instructions to camouflage the original instructions [38,242], new segments [247], dummy classes [84], dummy sequence using dead registers [47], and junk byte insertion to instruction stream [34,169], all fall into this class of transformation. Inserting additional NOP instructions [215,226,283] is another type of bogus insertion. NOPs are instructions that perform no operations but make it harder to predict where the pieces of code are placed in memory.
always evaluated as “true” or as “false”, yet needs to be evaluated at execution time. This hardens the task of analyzing the control flow and enhances the cost of comprehending the program [16,75]. Transformation can be applied to loops [268] by loop unrolling [166,201,272], loop intersection [73,182], extending [16,104,113] and eliminating [109], and changing the loop conditions [330]. Transformation can also be applied at instruction level to camouflage the original flow of the application [271] through instruction reordering [103,114,166,245,268], instruction hiding [226], and instruction replacement with dummy/fake instructions [38,76,175,291,346], or instructions that raise a trap [47,103,186,247]. Self modification mechanisms [38,62,165,182,202] alter/replace instructions at run-time which could be used to introduce an additional layer of complexity while obfuscating the code [161].

Modifying the control of a program not only makes it difficult to analyze the actual program’s flow, but also results in diverse binaries/executeable. This can be achieved through reordering the instructions [103,114,166,245,268] and blocks [27,103,135,202,239,346], while the semantics and dependency relations are preserved. Code transformation [52,63,162,202,211,214,230] is another way of producing dissimilar binaries. As an example, by randomizing the software in a sensor network, the nodes receive diversified versions of the software [149].

Other forms of control flow obfuscation are polymorphism [44,84], branching functions [34,47,123,157,179,209,240], and transforming/faking/spooﬁng jump tables [34,242]. Inlining method [41,103] replaces the function call with the function body, so the function is eliminated and the primary structures are not disclosed. Cloning method [229,231] creates different versions of the function and tries to conceal the information about the function calls.

- **Data obfuscation** aims at obscuring data and concealing data structure of a program [207]. In the surveyed studies, various approaches have been used to this aim [259,279,288]. First is array obfuscation [29] that targets the structure (and the nature) of an array, trying to make it confusing to the reader. This can be done through splitting an array into smaller sub-arrays [97,112,130,171], or merging multiple arrays and making one larger array [130,171]. Other ways of array obfuscation are array folding [85,112,130,171], that increases the dimensions of an array, and conversely, array ﬂattening [48,85,112,130,171], that decreases the dimensions of an array. Second is variable transformation to obscure/obfuscates variables [29,41,67,110,116,238,256]. Variables can be encoded [104], substituted with a piece of code [11], split into multiple variables [94,104,113], and vice versa, multiple variables can be merged together. Third is a more complex obfuscation technique, class transformation, which confuses the reader to comprehend the structure of a class [72]. This transformation includes class splitting into smaller sub-classes [36,41,128,177], merging/coaloesing multiple
classes together [36,41,148,177,223], class hierarchy flattening [84,128,223] which removes type hierarchy from programs, and type hiding [36,72,177]. There exist other classes of techniques to obfuscate the data structure of the program, such as code substitution [145], and encryption [53,67,86,110,147,213,235].

- **Layout obfuscation** is a class of obfuscation techniques that targets the program’s layout structure [13,336] through renaming the identifiers [45,51,98,101,110,117,125,163,187,212,213,233,320] and removing the comments, information about debugging, and source code formatting [113,170,201,223]. By reducing the amount of information for the human reader, the reverse engineering becomes harder. Layout transformations are considered as one-way approaches, as when the information is gone there is no way to recover the original formatting. Instruction Set Randomization (ISR) [55,66,105,140,154,158,167,186,192], Address Randomization [35,39,46,57,105,106,192,193,215,283], and Layout Randomization [41,52,88,113,146,149,160,178,193], Address Space Layout Randomization (ASLR) [263,265,308,337] can also be seen as identifier renaming techniques.

b) **Level**: We identified several phases in the software development, deployment, and execution as levels of obfuscation. In the reviewed research (Table 5), most techniques apply to development time \( (n = 282) \), runtime \( (n = 95) \), or both \( (n = 58) \). The development time techniques mostly apply to human readable source code (high level language & assembly), but obfuscation and diversification tools manipulating the generated binary formats (bytecode, native code, intermediate representation) are equally common. The application program itself provides the main platform for applying various mechanisms. At runtime, the techniques either target the source code (scripting languages), intermediate formats (e.g. JIT compilation), or the execution environments. Modified runtime systems are process level techniques for both managed (e.g. CLR & Java virtual machine) and native code,

**Table 5**

<table>
<thead>
<tr>
<th>Level</th>
<th>Development</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application design</td>
<td>11</td>
<td>–</td>
</tr>
<tr>
<td>Assembly source code</td>
<td>12</td>
<td>–</td>
</tr>
<tr>
<td>Bytecode</td>
<td>40</td>
<td>–</td>
</tr>
<tr>
<td>Executable</td>
<td>76</td>
<td>–</td>
</tr>
<tr>
<td>High level language source code</td>
<td>104</td>
<td>7</td>
</tr>
<tr>
<td>Intermediate representation format</td>
<td>39</td>
<td>5</td>
</tr>
<tr>
<td>Managed code</td>
<td>–</td>
<td>3</td>
</tr>
<tr>
<td>Native code</td>
<td>–</td>
<td>43</td>
</tr>
<tr>
<td>Hardware</td>
<td>–</td>
<td>3</td>
</tr>
<tr>
<td>Operating system</td>
<td>–</td>
<td>18</td>
</tr>
<tr>
<td>Virtualization</td>
<td>–</td>
<td>16</td>
</tr>
<tr>
<td><strong>Total no. of papers (impl &amp; runtime effects)</strong></td>
<td><strong>58</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Total no. of papers</strong></td>
<td><strong>282</strong></td>
<td><strong>95</strong></td>
</tr>
</tbody>
</table>
but operating systems, operating system / machine virtualization, and hardware level modifications are also presented.

In terms of obfuscation and diversification techniques, operating with source code that is not yet compiled is relatively effortless. Many of the reviewed techniques work purely on the lexical and syntactic levels and the parsing technology is mature, ranging from simple pre-processors to frameworks with compiler-like abilities. It is also possible to manipulate many high-level structures (classes, data structures) that are not available in machine code form [36]. In interpreted languages (e.g. JavaScript), the source code obfuscation is the only option [110], which also explains why some of the source code obfuscations are deferred to run-time. Collberg et al. have extensively described techniques available for source code obfuscation in [13,16]. Some of the mechanisms extend the range of obfuscations to semantically richer forms, the intermediate formats available during the compilation. Abstract syntax trees [133,207] are used by syntax oriented techniques while semantically richer intermediate formats provide access to e.g. control flow analysis. These mechanisms are provided for instance as compiler plugins.

The motive to obfuscate the source code is usually preventing the adversary from easily understanding and altering the code even if he or she has managed to reverse engineer it. Source code obfuscation might not ultimately prevent a dedicated attacker from understanding software, but it will significantly raise the bar of complexity and decrease the probability of a successful attack [49]. Source code obfuscation is often used for intellectual property protection [104,113]. Worth noting is that source code obfuscation is usually also reflected to the bytecode or binary code after compilation.

In managed environments, bytecode techniques have received lots of attention. For example, in Java, it is not that hard to reverse the compiled bytecode back to source code. This reverse-engineering can be performed via automatic tools [50,51]. Naturally, this poses problems for the confidentiality of source code and has elicited lots of research on bytecode obfuscation. Several approaches such as [30,45,177,182,223] have been proposed to prevent adversaries from understanding, reverse-engineering or cracking the bytecode. One major advantage of bytecode obfuscation (along with other binary code obfuscation techniques discussed next) is that source code is not needed in the process. This is quite often the case with closed source, third party software.

Reverse engineering and the manipulation of security measures are also issues with native code executables, but the native instruction sets are inherently harder to analyze due to more complex instruction sets and the lower level of abstraction. A large set of obfuscation and diversification techniques are applied to symbolic assembly code (with relocation information etc. intact) or disassembled final binaries [259,276]. This is often done in order to make reverse engineering considerably harder [171,247] or to prevent disassembling the program from binaries [175,240]. Low level obfuscation usually involves using control flow obfuscation transformations changing the sequence of instructions [123,245]. In general, increasing the entropy of the low level code also makes it harder for a piece of malware to modify the code or inject its own malicious payload [11,233]. One technique related with low level obfuscation is ISR [42,66]. An execution environment unique to the running process is created so that the attacker does not know the "language" used and therefore, cannot "talk" to the machine. A new instruction set is created for each process executing within a system.

c) Stage: Although modern software development is iterative, we observe the software life cycle as a linear sequence of stages: (a) development, (b) distribution and deployment, and (c) execution. This model captures the fact that each stage is characterized by a different set of obfuscation and diversification techniques and tools. The development stage is further split into design, implementation, compilation and linking phases. When analyzing the types of tools used to manipulate the application’s code, the compilation can be further refined into pre- (e.g. source to source transformations and code generators) and post-compilation (e.g. link-time code transformation) phases. The software deployment includes installation and updates [248]. Application loading occurs in conjunction with execution and thus is included in this stage. The surveyed studies discussed and applied obfuscation and diversification techniques during all these stages.

The Venn diagram in Fig. 12a illustrates all the observed stages and their overlap in five main groups, from design to application execution. These groups reflect the different stakeholders and roles in the software’s life cycle. We identified 16 different types of use of stages, with 201, 60, and 9 studies operating on one to three stages, respectively. None of the studies suggested taking part in four or more groups of stages. A majority of research involves compilation and linking. Execution time techniques form another large group. A small number of research is associated with either of these approaches and some other stage (n = 29) or is applied outside these stages (n = 22).

In Fig. 12a, the first group contains design and implementation phases. Mechanisms applied at this stage are involved in software development effort. Data obfuscation [118,137,162], control flow obfuscation [109,169] and, in general, source code obfuscation [63,99,118,188,225] are the most common approaches that target the code at implementation level.

The mechanisms in the next group, compilation and linking, can be applied to the deliverables of iterations for in-house software or to pre-made software, available either as source code, in intermediate forms, or as executable binaries that can be analyzed or reverse-engineered. This group is further dissected in Fig. 12b as the majority of reviewed
literature forms a cluster in this stage. The third stage, installation and update, includes the task of local deployment of software and updates. The next stage, loading, covers the process of loading the executable to memory (e.g. from a network stream or disk) and dynamic linking. Finally, execution stage includes all sorts of mechanisms that activate during application’s execution. Code obfuscation and software diversification can also be applied at execution time. Dynamic software mutation [79] is a repeated transformation of the program during its execution. It makes a region of memory occupied by various code sequences during execution. Identifier renaming [163], ASLR [57,265], camouflaging the instructions by overwriting them with fake instructions [76], and randomizing the location of critical data elements in memory [140] are other examples of execution-time diversification.

Fig. 12b focuses on the mechanisms applied on various stages of the compilation (pre-, post-, and during compilation). In the figure, all the remaining stages after the compilation techniques have been combined as a single post-distribution stage. The reviewed research is distributed quite evenly between the different compilation stages. Second large class of mechanisms is to use compilation or post-compilation in conjunction with the execution time techniques.

Diversification at the compile-time makes the process fairly automatic by eliminating the need to change the program’s source code [248]. M. Franz [150] has proposed a practical approach for generating diverse software at compiler-level. This approach is based on an app-store that contains a multi-compiler, which works as a diversifying engine that generates unique binaries with identical functionality. In [207] they have developed a compiler plugin to generate diverse operating system kernels, through memory layout randomization. As we mentioned before, in the literature, there are several works that study the pre-compile time and post-compile diversification. Control flow transformation in the source code [43] is an example for the former, and class transformation in Java bytecode [177] an example for the latter.

d) Cost: Despite of the security obfuscation and diversification bring, they introduce cost and overhead to the system, like any other security measure. In fact, the higher level of obfuscation/diversification, the more penalty is forced to the system. Therefore, based on the need of the system, it is decided how much the program needs to be obfuscated/diversified. In the studied works overhead mainly was reported as a) increase in the program size [50,84] (e.g., number of instructions [34], memory size, code size [240,256,264], binary patch size [140], byte code size), b) increase in program performance [261,263,266,285,290] (e.g., compile time [175], process time, execution time [260,268], CPU overhead [119], higher memory usage [119,273], and c) latency and throughput [210] (in load time or run time). It is worth mentioning that among the diversification mechanisms, some introduce more cost and some less. For instance, changing variable names, function names, and system call numbers often introduces no additional costs.

e) Effectiveness: In the studied works, the effectiveness of the proposed approaches were mainly measured through the following metrics:

- **Potency** determines to what degree a human reader is confused, as a result of the applied security measure [13]. Measuring the potency can be done by comparing the obfuscated/diversified version of the software with the original version and presenting the similarity rate [257,290]. In [131] clone detection is used to analyze the similarity of the obfuscated code with the original one, and the code dissimilarity is the metric for representing the potency of the approach. Another way to measure the potency of an obfuscation mechanism is to evaluate how much harder it has become for a human reader to comprehend the obfuscated code, comparing to the original code. For instance, the obfuscation mechanism in [27] has been tested empirically with a group of students, programmers and crackers and illustrated that only a few crackers were able to deobfuscate the obfuscated code. In [227] the effectiveness of the proposed approach is measured by static and dynamic analysis of the obfuscated code.

- **Resiliency** determines how well the obfuscated/diversified program resists automatic decomplers/disassemblers/deobfuscators [13]. Analyzing the reverse engineering effort demonstrates how the proposed technique is effective against disassembly tools (e.g., through presenting confusion factor, and disassembly errors). For instance, in [211,240] the strength of the obfuscation mechanism has been evaluated against IDA PRO automated deobfuscators [8], and demonstrate that obfuscated code increases the effort for an attacker, by making it harder to reconstruct the original code. Similarly, Linn et al. [34] have used three state-of-the-art disassembly tools, and demonstrated the effectiveness of their approach through confusion factors, disassembly errors, and incorrectly disassembled code, that they gained by disassembling the obfuscated code.

- **Attack Resistance** determines how much harder it has become to break the obfuscated code. It can be done by running the obfuscated/diversified software against different attacks and analyzing the outcome [260,263,268,285]. As an example, the obfuscated kernel in [207] is tested against four kernel rootkits, and it is shown that all they were disabled. RandSys prototype [105] implemented for Linux and Windows has been tested against two zero-day exploits (code-spraying attacks), and 60 existing code injection attacks. It was shown that the approach is successful in thwarting them. In [268] they run the program against various types of attacks (e.g., code injection, memory corruption, code reuse, tampering and reverse engineering attacks) and measure the resistance.

### 4.2. Empirical studies

As mentioned before, in the set of studies collected, 68 of them were studying the obfuscation/diversification techniques empirically. These empirical studies come in the form of discussion, experiment, evaluation, comparison, optimization, survey, and presenting a classification. The following categories illustrate how these studies were related to obfuscation and diversification:

- survey of related works on obfuscation and diversification as software protection techniques [5,6,37,64,180,248,284]; Baudry and Monperrus [284] survey the related works on design and data diversity which consider fault tolerance and cybersecurity. They also study randomization at various system levels.
- overview/classification of existing obfuscation/diversification techniques [59,78,132,324];
- studying the obfuscating transformations that are (more) resilient to slicing attacks [92,96,329];
- comparing different obfuscation mechanisms [87,95,190];
- discussion on a particular obfuscation mechanism [19,78,132]; In [132], obfuscation is being discussed as a way to make understanding the software more difficult. In [19], identifier renaming is discussed as an obfuscation mechanism to protect Java applications. By making the classes harder to decode, the act of unauthorized decompilation becomes difficult. In [78], the authors overview the existing obfuscators and obfuscation mechanisms, and also illustrate the possibility of achieving binary code obfuscation through source code transformation.
- studying/evaluating the effectiveness of an obfuscation/diversification approach (e.g., identifier renaming and opaque predicates) against human attackers [54,64,74,93,176,183,246,266,269,278,289,295]; In [68] the authors qualitatively measure the capabilities
and performance of two commercial obfuscators for three different sorting algorithms. In [93], the effectiveness of decompilers and obfuscators are quantified through a set of metrics. It is done by comparing the original Java source code with the decompiled and obfuscated code respectively. The metrics will then measure whether the decompiler produces valid source code, and whether the obfuscator produces garbled code. [176] measures the effects of different obfuscation techniques on Java code in terms of complexity. In [278] several different metrics are suggested for measuring the incomprehensibility of the obfuscated code.

- optimizing and reducing the overhead of software diversity [83,202,237,255];
- experimenting and evaluating the potency of an obfuscation technique; The strength and incomprehensibility of the obfuscated programs can be evaluated by measuring the performance of human analyzers in analyzing the obfuscated code (to what degree a human reader is confused) [111,121,145,228,234,354].
- studying the effectiveness of software diversity [32,65,136,237,274,287,292,360]; For instance, to evaluate the effect of diversity, several different computer attacks are tested against the diversified programs [32]. In [274], automatic software diversification is discussed as a means for securing the software. The authors investigate the types of expotation it can mitigate, the different levels of software life-cycle the diversification can be applied at, and the possible targets of diversification.

5. Discussion

The idea of protecting software through generated diversity and obfuscated code originated in early 1990's and gained more attention in the past decade. The rationale behind these techniques is to increase the cost and effort for a successful attack. This study, by surveying the literature about the use of these two techniques for securing software, elucidates several points.

First, these methods have been used in various ways with different aims, such as protecting software from malicious reverse engineering and tampering, hiding some data and protecting watermark information, preventing the wide spread of vulnerabilities and infections, mitigating the risk of massive-scale attacks, and impeding targeted attacks. In a previous study, we have studied the aims and environments that these two techniques have been applied to [324].

Second, the field has grown in many directions, and new areas have emerged. Moving Target Defence (MTD) [172] is an example of newly born defence mechanisms. MTD randomizes the system components, and presents a continuously changing attack surface, which shortens the time frame available for attacker.

Third, studying all the related works sheds light on the research gaps, and the potential research directions. We discuss these research gaps in Section 5.2.

Fourth, There are also some challenges associated with practical diversification and obfuscation that require further study. In Section 5.1 we discuss some of these challenges.

5.1. Challenges

One of the challenge of applying diversification is the fact that it has to be propagated to every parts of the system that makes use of the diversified interface. For example, diversified system call numbers in the operating system kernel have to be propagated to libraries that employ system calls. It is not straightforward to automatically find all the dependencies.

Software updates are also a challenge for diversified systems, because each update has to be individually diversified to be compatible with the uniquely diversified interfaces it uses. Then again, each patch is also an opportunity to re-diversify some parts of the system.

Using diversification and obfuscation is always a race against time in the sense that the adversary will ultimately be able to guess the correct diversification key or deobfuscate the code. Dynamically changing obfuscation/diversification is a good defence, but raises complexity and performance costs of the approach.

5.2. Research gaps

As mentioned before, the main goal of conducting a systematic literature review is to collect and analyze the studies related to a field of research, in order to pinpoint the research gaps in that field.

One of the gaps we have found in this field of study is the lack of a standard metric in this field for reporting the overhead and also the effectiveness of the proposed approaches. The terms, potency and resilience, introduced by Collberg et al. [16] discuss how much more complex the code has become in the presence of the obfuscation method; however, we believe that a standard metric to present a numeric degree is missing. Moreover, as also discussed in [276], the majority of the existing research are constructive papers and there is need for more empirical research, such as measuring the efficiency of diversification and presenting results on performance and space requirements of the obfuscation techniques. However, these concerns have not fully addressed by the research community.

Another research gap that we noted was that there are still many environments obfuscation and diversification have not been applied to yet, even though this would potentially be beneficial. In this section, we discuss these environments and present some ideas on how to utilize the two techniques as a complementary measure to the security measures these environments already have.

One of these execution environments is cloud computing and virtual environments, in general. Lately, virtual technologies have become very dominant as many enterprises and service providers are shifting towards the cloud and to deliver their services through it. Thus, we believe that due to the significance of these environments, there are needs for proactive approaches for securing their software. Obfuscation and diversification could become helpful in this manner. Our recent survey on the use of these techniques for securing cloud computing environment [293] clearly shows that there have been very limited number of studies on this domain and there is still room for more studies in this area. Also, the related works do not address the propagation issues, especially regarding propagation to higher level interfaces/APIs. One possible solution that we propose here is to diversify the internal interfaces of cloud and virtualization systems, for instance, diversifying the machine language of the virtual machines.

Therewith, container-based virtualization is drawing more attention recently, because of the advantages that it has compared to traditional hypervisor-based virtualization (such as higher efficiency in CPU, memory, and storage). We believe that this environment can also benefit from diversification techniques. However, most notably there is hardly any research related to diversification of hypervisors or containers. One potential idea is to diversify interfaces and APIs of containers, so each container would have a diverse execution environment.

IoT network is another type of environment that is becoming prevalent more and more these days. Protecting these networks is also crucial not only at network level but also at (device) application level. In the reviewed literature, obfuscation and diversification have been used very little as a security measure for this purpose. Therefore, this is another research direction to consider. We have proposed applying these two techniques on the operating systems of the IoT devices and also on the communication protocols used among them [275,322]. Then, in another study we applied diversification on Thingsee and
Raspbian operating systems [305] (the limitations of this study is discussed in Section 5.3).

These ideas could be also extended to fog computing as well. Looking at the architecture of fog computing, nodes (i.e., sensors, devices and data collectors) come at the edge of the network. Applying diversification at the nodes stops the malicious activity right at the edge before it goes up to the fog.

As mentioned earlier, diversification techniques could be applied as a complementary security approach, to the measures environments already have. We are studying applying diversification in trusted environments. More specifically, we consider diversifying the containers and storing the diversification secret inside TPM. Another idea is diversifying enclaves in a trusted execution environment such as Intel SGX.

Applying obfuscation to mobile platforms is also a topic that has received some attention recently but still requires more research. For example, Android applications, distributed in Dalvik bytecode form, are vulnerable to reverse engineering. There has been some positive development in this front, such as the ProGuard obfuscation tool being packed along with Google’s Android Studio development environment and some recent publication on obfuscation in the context of mobile applications [35]. Most current studies on obfuscation in mobile environments, however, only highlight obfuscation techniques Android malware uses to evade detection. Therefore, with applications run on mobile platforms getting more prevalent, new obfuscation and diversification techniques to increase software security and research on their resilience are still needed.

To sum up, by systematically studying the two software security techniques, obfuscation and diversification, and finding the research gaps and research opportunities, we believe that both research community and industry should study and apply these techniques. There has already been some development in this direction. As we saw in Section 4.1.1, research on these techniques has drawn fair attention in software security community. This trend is also shown by several patent applications filed during the last ten years. For example, there are patents on ISR [9] and interface diversification [10]. There have also been some initiatives by Microsoft research center [119]. Still, we would like to encourage the practical application of these techniques even more – especially by industry.

5.3. Limitations of the study

As discussed earlier, a systematic literature review has a number of advantages compared to the traditional literature reviews, such as transparency, larger breadth of included studies, and reduction of research bias [4]. Nevertheless, this method has several different practical challenges and limitations as well, that we experienced in this work that made it difficult to put it into practice. First, it was very time-consuming and arduous work, due to the high number of included studies, in all the process of search, selection, data extraction, and also synthesis of the data and classification of them. Second, we selected the set of search strings manually. Thus, there is a concern that not all the materials in the field are captured. Third, the inclusion or exclusion of the studies might be biased based on the researcher’s knowledge. Moreover, because of high number of authors, sometimes, each author had slightly different interpretations of the research questions. To solve the inconsistency in the screening process, we solved the disagreement cases in the meetings with the presence of all authors. Fourth, in some cases the search flow was dictated by limitations of some of the databases. As an example, in SpringerLink digital library we had to make the search in the full-text of the studies, which resulted in a huge number false positives. For other databases, we had limited the search to the title, abstract, and keywords of the studies.

In addition, like any other security measure, despite of the security benefits that the two studied techniques have, they also have some limitations such as increase in the size of the obfuscated code, and consequently increase in the execution time. In diversification method propagation of the diversification throughout the whole system from the lower levels to the upper most layers, is still a challenge and needs improvement.

Moreover, not all the diversification techniques are well suited for the tiny resource constrained devices. In other words, the diversification method should be chosen with consideration. For example, embedded systems may not always have Memory Management Units (MMU), which makes applying ASLR less worthwhile. We can perform device-specific diversification with the layout, but the embedded system requires certain offsets to be static, which means since there is no MMU we cannot hide these offsets with ASLR, which could mean all the diversification was for nothing since the exploiting code can possibly crawl through the known offsets to the offsets it needs to function. Diversification can expand the size of the system, which means the space constraints of such tiny systems come at us faster. In our previous experiments, for example, applying layout shuffling and symbol diversification required some extra space [239]. In Thingsee OS the size of binaries expanded, but not excessively.

6. Conclusion and future work

Obfuscation and diversification are promising software security techniques that protect computers from harmful malware. The idea of these two techniques is not to remove the security vulnerabilities, but to make it challenging for the attacker to exploit them and perform a successful attack.

In this study we reviewed the studies that were applying/studying obfuscation and diversification for improving software security. For this purpose, we systematically collected and reviewed the related studies in this field, i.e., 357 articles (See Appendix A), published between 1993 and 2017. We reported the result of study in form of analysis and classification of the captured data, and also we managed to answer the formulated research questions. We found out that these two techniques have been utilized for various aims and mitigation of various types of attacks, they can be applied at different parts of the system and at different phases of software development life-cycle. Moreover, in the literature, there exists many different techniques to obfuscate/diversify the program that each present different levels of protection, but also overhead, which depending on the need of the program could be chosen.

Moreover, by studying the existing works we pinpointed the research gaps. We concluded that the major part of the existing research works were focusing on obfuscation, and there is still room for studies on software diversification. We discussed that there exist many different environments that could still benefit from these techniques and need more focus of research, such as virtual environments, IoT, fog computing, and trusted computing. For the discussed environments, we also presented some potential ideas.

As future works, we will apply diversification on the interfaces of containers, also diversify the codes inside enclaves in trusted execution environments.

Acknowledgment

The authors gratefully acknowledge Tekes – the Finnish Funding Agency for Innovation [grant number 3772/31/2014], DIMECC Oy, and the Cyber Trust research program for their support.
Appendix A. Selected studies

The following is the list of 357 studies reviewed in this SLR, sorted based on the publication year:

1993
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
References


[78] A. Majumdar, S.J. Drape, C.D. Thomborson, Slicing obfuscation: design,


A. Capiluppi, P. Falcarin, C. Boldyre X. Yao, J. Pang, Y. Zhang, Y. Yu, J. Lu, A method and implementation of control
M. Christodorescu, M. Fredrikson, S. Jha, J. Gi Y. Huang, A.K. Ghosh, Introducing diversity and uncertainty to create moving
S. Hosseinzadeh et al. S.W. Boyd, G.S. Kc, M.E. Locasto, A.D. Keromytis, V. Prevelakis, On the general
S. Hu, A. Nguyen-Tuong, C. Hsu, C. Wang, J.W. Davidson, IIC: where shou- ld my gadgets?
A. Gupta, M. Kirkpatrick, E. Bertino, A secure architecture design based on ap- plication isolation, code minimization and randomization, Proceedings of the IEEE


