



# Evaluation Methodologies in Software Protection Research

**BJORN DE SUTTER**, Electronics and Information Systems, Ghent University, Gent, Belgium

**SEBASTIAN SCHRITTWIESER**, Christian Doppler Laboratory for Assurance and Transparency in Software Protection, University of Vienna, Vienna, Austria

**BART COPPENS**, Electronics and Information Systems, Ghent University, Gent, Belgium

**PATRICK KOCHBERGER**, Institute of IT Security Research, St. Pölten University of Applied Sciences, St. Pölten, Austria

---

*Man-at-the-end* (MATE) attackers have full control over the system on which the attacked software runs, and try to break the confidentiality or integrity of assets embedded in the software. Both companies and malware authors want to prevent such attacks. This has driven an arms race between attackers and defenders, resulting in a plethora of different protection and analysis methods. However, it remains difficult to measure the strength of protections because MATE attackers can reach their goals in many different ways and a universally accepted evaluation methodology does not exist. This survey systematically reviews the evaluation methodologies of papers on obfuscation, a major class of protections against MATE attacks. For 571 papers, we collected 113 aspects of their evaluation methodologies, ranging from sample set types and sizes, over sample treatment, to performed measurements. We provide detailed insights into how the academic state of the art evaluates both the protections and analyses thereon. In summary, there is a clear need for better evaluation methodologies. We identify nine challenges for software protection evaluations, which represent threats to the validity, reproducibility, and interpretation of research results in the context of MATE attacks and formulate a number of concrete recommendations for improving the evaluations reported in future research papers.

CCS Concepts: • **Security and privacy** → **Software reverse engineering**;

Additional Key Words and Phrases: Survey, software protection, obfuscation, deobfuscation, diversification

## ACM Reference Format:

Bjorn De Sutter, Sebastian Schrittwieser, Bart Coppens, and Patrick Kochberger. 2024. Evaluation Methodologies in Software Protection Research. *ACM Comput. Surv.* 57, 4, Article 86 (December 2024), 41 pages. <https://doi.org/10.1145/3702314>

---

B. De Sutter and S. Schrittwieser share dual first authorship.

This research was funded by the Austrian Science Fund (FWF): I 3646-N31, by The Research Foundation - Flanders (FWO): 3G0E2318, and by the Cybersecurity Research Program Flanders. Further, the financial support by the Austrian Federal Ministry of Labour and Economy, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, and SBA Research (SBA-K1), a COMET center funded by BMK, BMAW, and the state of Vienna, is gratefully acknowledged. Likewise, the funding by Ghent University is appreciated.

Authors' Contact Information: Bjorn De Sutter, Electronics and Information Systems, Ghent University, Ghent, Belgium; e-mail: [bjorn.desutter@ugent.be](mailto:bjorn.desutter@ugent.be); Sebastian Schrittwieser, Christian Doppler Laboratory for Assurance and Transparency in Software Protection, University of Vienna, Vienna, Austria; e-mail: [sebastian.schrittwieser@univie.ac.at](mailto:sebastian.schrittwieser@univie.ac.at); Bart Coppens, Electronics and Information Systems, Ghent University, Ghent, Belgium; e-mail: [bart.coppens@ugent.be](mailto:bart.coppens@ugent.be); Patrick Kochberger, Institute of IT Security Research, St. Pölten University of Applied Sciences, St. Pölten, Lower Austria, Austria; e-mail: [patrick.kochberger@fhstp.ac.at](mailto:patrick.kochberger@fhstp.ac.at).



This work is licensed under a [Creative Commons Attribution International 4.0 License](https://creativecommons.org/licenses/by/4.0/).

© 2024 Copyright held by the owner/author(s).

ACM 0360-0300/2024/12-ART86

<https://doi.org/10.1145/3702314>

## 1 Introduction

The desire and need to protect software from analysis, reverse engineering, and tampering have always existed. Benign software vendors want to keep the exact implementation of their product and its embedded data, such as cryptographic keys, secret. They want to prevent the removal of copy protections, cheating in multi-player games, and so on. Malware authors also often employ **software protection (SP)** mechanisms to prevent detection and analysis of the malicious nature of their code.

Over the past three decades, a vast number of obfuscation, tamperproofing, and anti-analysis techniques for both goodware and malware have been introduced in the literature [73, 98, 148] and in industry to meet the described needs. The deployment of such protections has in turn triggered an arms race in which many new deobfuscation, tampering, and analysis techniques have been proposed. These are collectively called **man-at-the-end (MATE)** attacks, because the *human attackers* using the deobfuscation, tampering, and analysis techniques have full control over the end system on which they run, analyze, and alter the software and malware under attack. Their tools include emulators, debuggers, disassemblers, custom operating systems, fuzzers, symbolic execution, sandboxes, and instrumentation among a wide range of static and dynamic analysis and tampering tools. These give them white-box access to the software and malware internals, including their internal execution state. To a considerable degree, but not completely, benign software and malware analysis techniques overlap, and hence so do the deployed protections.

In the MATE attack model, and with the current state of the art in practical SPs, complete prevention of attacks on software and malware protections is impossible [133]. With enough resources and time to exploit their white-box access, attackers will always succeed. The goal of benign MATE SP is hence to make potential attackers' return on investment negative. The already mentioned SPs can help to achieve this by delaying the successful identification and engineering of attacks, and by limiting the exploitability of identified attacks. Software diversity also plays an important goal, as it can limit the attacker's use of *a priori* knowledge to identify an attack and hinder its wide-scale exploitation. The goal of malware protection is of course the inverse, namely to make the malware authors' return on investment positive by delaying analysis and detection to the point where the malware has generated enough money or caused enough damage.

These goals are much fuzzier than the goals in some other security domains such as cryptography, which builds on precisely defined mathematical foundations to protect against man-in-the-middle attacks. While cryptographic techniques and well-defined, cryptographically-sound concepts have also been developed in the domain of obfuscation, the results (be it negative [25] or positive [120]) are far from ready for widespread, general use in practice. Because of the mentioned fuzziness, and because MATE attackers have so many alternative attack strategies and techniques at their disposal, it is difficult to measure the strength of SPs and to make founded statements about them. As attackers can reach their goal in so many ways, there is not even a universally applicable and accepted evaluation methodology. For example, while there is a common understanding that the relevant aspects to evaluate include potency, resilience, stealth, and cost [57, 58], there is no universally agreed and applicable method to define, qualify, or quantify the former three.

Such issues were discussed extensively at the 2019 Dagstuhl Seminar on Software Protection Decision Support and Evaluation Methodologies [68]. Most participants felt that there is a lack of good benchmarks for evaluating SP, and that the significance and validity of evaluations of proposed methods remain unclear. The fact that this gut feeling and (collective) subjective concern based on anecdotal experience was not backed up with systematic research motivated us to perform this survey. Our overall goals are to validate whether the mentioned concerns really exist, to draw a clear picture of the currently used methods to evaluate SPs, and to identify challenges and outline directions for better comparability and reproducibility of publications in the field of SP.

To that extent, we systematically analyze the content of publications on SP and code analysis that target software confidentiality, i.e., that focus on the topics of obfuscation and deobfuscation and techniques to perform and prevent analysis of goodware and malware. With an initial set of 1,491 publications and an in-depth analysis of 571 papers, this survey is by far the largest literature study in this field to date. It is organized as follows: Section 2 presents the methodology we adopted to include, exclude, and categorize papers, and for extracting data from them. Section 3 analyzes the sample sets used for evaluating the papers' contributions, including the sample set sizes and the nature of the used samples. Section 4 discusses the treatment that the samples underwent as discussed in the papers, i.e., how the protected (and unprotected) samples were generated. Section 5 studies how measurements were performed on the samples. Section 6 reports our findings about experiments involving human subjects. Section 7 presents our recommendations for future research. In Section 8, we list related literature surveys and reviews. Finally, Section 9 concludes this work.

## 2 Scope and Methodology

This survey comprises *peer-reviewed* papers on *practical general-purpose SP* techniques that aim to protect the *confidentiality* of assets embedded in software (goodware or malware) by preventing or hindering software analysis and reverse engineering in the attack model.

The SPs in scope target software formats ranging from source code over bytecode/**intermediate representations (IRs)** to compiled software (binary code). Such SPs include various obfuscation techniques (e.g., opaque predicates) and preventive protections (i.e., that prevent or hinder the use of reverse engineering tools and analysis tools, such as anti-debugging techniques). Our SP scope also includes methods that use hardware to protect software (e.g., relying on physically unclonable functions). However, schemes that protect hardware (i.e., integrated circuits) are excluded. Furthermore, out-of-scope are SPs that prevent the exploitation of vulnerabilities (e.g., ASLR), as well as anti-tampering schemes that exclusively focus on protecting the integrity of embedded assets, such as code guards and remote attestation. In addition, methods for obfuscating data that is processed by, but not included in the distributed software are out of scope (e.g., network traffic and location data obfuscations), as well as software-based steganography (i.e., techniques to inject external data into a program in a hidden manner). Because our focus is on practical general-purpose SP techniques, we exclude papers focusing on special-purpose obfuscation (e.g., white-box cryptography [81] and obfuscations of point-functions, pseudo-random functions, and finite automata) as well as cryptographic obfuscation (e.g., indistinguishability obfuscation, virtual black box obfuscation, fully homomorphic encryption) [96] that we deem not ready for use in practice yet.

This survey includes papers that present (i) novel SPs in scope; (ii) reverse engineering tools and techniques with which those SPs in scope are attacked, identified, detected, located, circumvented, deobfuscated, analyzed, undone, bypassed, worked-around, and so on; (iii) practical or theoretical methods or models to reason about, evaluate, and deploy those SPs and attacks; (iv) as well as surveys on them. Pure position papers, non-peer-reviewed papers, blog posts, books, and posters are out of scope. Table 1 lists all our exclusion criteria.

### 2.1 Paper Retrieval and Selection Process

The starting point of our study are papers on the topic of obfuscation gathered from three online libraries. In the ACM Digital Library, IEEE Xplore, and SpringerLink we searched for paper titles including “obfuscation”, “obfuscate”, “obfuscating”, and other variants thereof, including variants of “deobfuscation”, from 2016 to 2022. From those years, we also added papers based on their title from the USENIX Security and **Network and Distributed System Security (NDSS)** symposia, two top security conferences that are not included in the aforementioned libraries. For papers older

Table 1. Inclusion and Exclusion Criteria

Inclusion 1	Obfusc* deobfusc* in paper title published between 2016–2022 and listed in ACM Digital Library, IEEE Xplore, SpringerLink or published at the USENIX Security or NDSS symposia
Inclusion 2	Paper is cited in [73, 98, 148]
Inclusion 3	Offensive or defensive paper on SP or malware analysis published before 2023 by any of a self-curated list of authors <sup>1</sup>
Exclusion 1	Not peer-reviewed
Exclusion 2	Posters, books, pure position papers
Exclusion 3	Targets obfuscation of something else than software, such as location or energy meter data
Exclusion 4	Topic is a form of cryptographic obfuscation
Exclusion 5	Main focus is special-purpose obfuscation such as white-box cryptography
Exclusion 6	Protection aims for mitigating vulnerability exploits, not on confidentiality of software
Exclusion 7	Focus of paper is software integrity protection rather than confidentiality
Exclusion 8	Focus is on steganography instead of software confidentiality
Exclusion 9	Evaluated or analyzed techniques do not include protections in scope

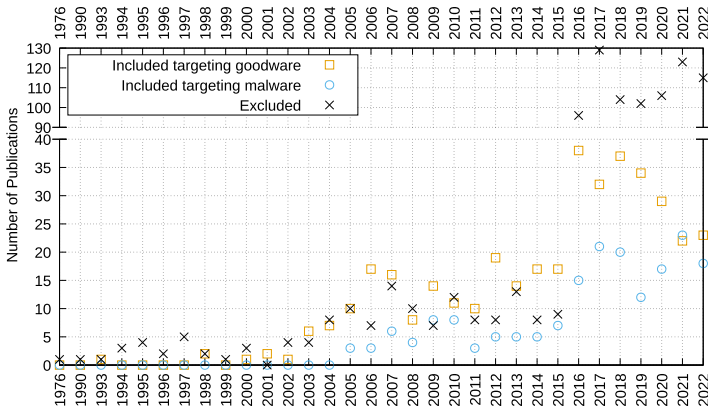


Fig. 1. Evolution of included goodware and malware papers, as well as excluded papers.

than 2016, we limited the selection to papers referenced in three well-known surveys related to this survey [73, 98, 148]. In addition, a self-curated list<sup>1</sup> of researchers working in the area of SP was used to manually identify additional papers. Table 1 lists these inclusion criteria. The preliminary list obtained with the described, rather mechanistic retrieval process totaled 1,491 papers. Many of those were false positives, however, so after the initial selection we manually filtered out papers that are out of scope as determined by the listed exclusion criteria. Eventually, 571 (38%) remained for in-depth analysis. This is hence by far the largest literature study on SP to date. By comparison, the aforementioned surveys only contain 203, 367, and 98 cited works.

Figure 1 depicts the publications included and excluded per publication year. About 60%  $\hat{=}$  343/571 of the papers are from the 2016–2022 period. This is in line with our ambition to shed a light on currently used evaluation methods in the domain of SP. We deem the inclusion

<sup>1</sup>C. Basile, A. Pretschner, A. Lakhotia, A. Francillon, B. Wyseur, C. Henderson, C. Collberg, C. Thomborson, C. Maurice, C. Mulliner, D. Boneh, J. Davidson, L. Goubin, M. Videau, M. Bailey, M. Franz, M. Dalla Preda, M. Ahmadvand, N. Stakhanova, N. Provos, N. Eyrolles, R. Giacobazzi, S. Debray, S. Banescu, S. Bardin, S. Katzenbeisser, T. McDonald, and Y. Gu.

Table 2. Categorization of the 571 Surveyed Papers

<b>Perspective</b>		
Obfuscation	360 $\hat{=}$ 63%	novel SPs, deployment and evaluation methods, insights into SP practices
Deobfuscation	102 $\hat{=}$ 18%	novel tools/methods for obtaining deobfuscated software representations
Analysis	166 $\hat{=}$ 29%	other analysis and (malware) classification techniques, evaluation thereof
<b>Application</b>		
Diversification	33 $\hat{=}$ 5.8%	diversification of SPs or as an SP
Tamperproofing	42 $\hat{=}$ 7.4%	use of SPs to strengthen tamperproofing of code, or attacks on them
Watermarking	12 $\hat{=}$ 2.1%	use of SPs for building watermarks, or attacks on them
<b>Target</b>		
Malware	183 $\hat{=}$ 32%	paper focuses on malware and/or includes malware evaluation samples
Goodware	388 $\hat{=}$ 68%	all papers not categorized as malware according to above criterion
Malware only	99 $\hat{=}$ 17%	malware paper focusing on techniques that are only useful for malware
Mobile	88 $\hat{=}$ 15%	paper targets Android or iOS apps
<b>Nature</b>		
Survey	38 $\hat{=}$ 4.4%	literature studies, surveys, meta-analysis papers
Theoretical	29 $\hat{=}$ 5.1%	theoretical approaches, e.g., abstract interpretation for strength evaluation
Data science	77 $\hat{=}$ 13 %	presented approaches rely on machine learning or artificial intelligence
Human Experiment	20 $\hat{=}$ 3.5%	paper presents experiments with humans performing tasks

Categories are non-exclusive.

of the other 40%  $\hat{=}$  230/571 of older papers important, however, because it enables us to identify potential historic trends and evolutions in how SPs are being evaluated.

## 2.2 Information Collection

For each paper, we collected 113 aspects ranging from the contribution area (obfuscation, deobfuscation, analysis, etc.), over types of targeted software, types of performed measurements, and used sample sets, to the discussed and deployed protection and analysis methods and tools. We use our own terms and definitions for all categories, which sometimes diverge from how authors used the same terms in their papers. Our definitions are listed in the supplemental material. Overall, 64,523 individual data points were gathered. We partitioned the papers among all authors of this survey. Care was taken not to finalize any paper's data collection before discussing any corner cases or doubts that surfaced and before obtaining consensus, i.e., agreeing unanimously on how to classify them. All classifications and information were collected in a spreadsheet. At times, additional categories or evaluation aspects were added and definitions of existing categories were updated or refined. At those occasions, potentially impacted, already reviewed papers were revisited.

We further conducted multiple rounds of double checking in the spreadsheet for consistency and performed various plausibility checks on the data to further improve its quality.

## 2.3 Top-level Paper Categorization

We categorized all papers non-exclusively using the top-level categories listed in Table 2. In doing so, we consider malware detection and malware classification papers to be analysis papers. Papers labeled as analysis papers are in addition labeled as deobfuscation papers if the presented analysis reveals the information necessary to deobfuscate protected code fragments. Examples are control flow analyses that identify opaque predicate-based bogus control flow as unrealizable, or techniques that reveal which library APIs are called at obfuscated call sites.

	388 goodwill papers								183 malware papers					
source language	N	M	S	N&M	S&N	S&M	S&N&M	-	N	M	S	N&M	S&N	-
	210	103	15	9	1	1	5	44	94	47	24	15	2	1

Fig. 2. Targeted languages: Managed, Natively compiled, and Scripts, plus language-agnostic papers (-).

If a paper’s contribution is some analysis or framework to assess the strength of an obfuscation, we consider this an obfuscation contribution, as it advances our knowledge about obfuscations and evaluation methods thereof. We only consider it an analysis contribution if it also advances knowledge about analysis techniques. Detection algorithms, such as for library code and cryptographic primitive detection, are framed by some authors as analysis research, and by others as deobfuscation research. We consider them analysis papers because they classify software components similarly to what malware classification does. We also label them as deobfuscation papers, however, because in the eyes of an attacker they summarize complex, obfuscated low-level code instances into abstract concepts that are void of any obfuscation.

All papers not categorized as *malware* using the criterion in Table 2 are categorized as *goodware* papers. Hence, those include papers that do not specifically target malware and that do not use malware samples, but of which the content can still apply to both benign and malicious software.

► *Our survey includes many more defensive papers than offensive ones.* In the context of malware, defensive papers focus on analysis, detection, and deobfuscation techniques to safeguard systems from malware. In the context of goodware, by contrast, defensive papers focus on obfuscation to protect the confidentiality of software assets against reverse engineering. A Venn diagram categorizing all papers is included in the supplemental material.

► *Our survey includes few multi-perspective papers, and even fewer papers (1.2%  $\hat{=}$  7/571) that contribute novel techniques and at once evaluate the impact of countermeasures that adversaries might adopt.* Table 3 lists the various ways in which multi-perspective papers contribute. This lack of multi-perspectivism is important because SP is an arms race between defenders and attackers, a.k.a. a cat and mouse game. In case evaluating a countermeasure against one’s own contribution requires a considerable research or engineering effort, it is acceptable to publish the initial contribution and the countermeasure separately. However, we think it occurs much more often that one can evaluate the impact of (small) adaptations by adversaries with relatively little effort. In such cases, it is warranted for reviewers and readers to expect and demand such an evaluation in the papers. This is all the more important because a crucial aspect of MATE attackers’ modus operandi is to find and exploit the attack-path-of-least-resistance. Whenever some novel SP is proposed and evaluated that impacts some path-of-least-resistance, it is important to evaluate whether there are no trivially similar, alternative paths-of-least-resistance remain unimpacted. A best practice for SP researchers that make a cat or mouse move is therefore to look ahead and evaluate at least the smallest next moves that can be anticipated. This best practice expectation was explicitly raised in Dagstuhl.

► *The literature is lacking with respect to multi-perspectivism and direct countermeasure evaluation.*

We also categorized the papers according to the categories of programming languages they target, i.e., the languages on which the paper authors demonstrate or evaluate their contributions. Figure 2 shows the results. In line with existing work [97], we considered four categories:

- *Native languages* such as C/C++ are compiled to and distributed as native code binaries.
- *Managed languages* such as Java or C# are typically compiled to and distributed as bytecode to be executed in a managed environment, for which enough symbolic information needs to be included, e.g., to support garbage collection, type correctness verification, and reflection.

Table 3. Multiperspective Papers and How They Combine Contributions in (De)Obfuscation and Analysis

<b>Obfuscation, Deobfuscation, and Analysis: 0.4% <math>\hat{=}</math> 2/571 papers</b>	
[88]	survey
[121]	presents interactive tool for all three tasks
<b>Obfuscation and Deobfuscation: 1.8% <math>\hat{=}</math> 10/571 papers</b>	
[24, 72]	theorize about both kinds of tasks
[9, 135, 144]	survey and/or evaluate both kinds of transformations
[1, 170]	present a novel deobfuscation technique to study the prevalence of obfuscation techniques in real-world samples
[52, 128]*	present a deobfuscation technique to defeat existing obfuscations and novel obfuscations as countermeasures
[147]*	presents new obfuscation techniques as well as novel deobfuscation techniques, with the latter outperforming the existing state-of-the-art while not succeeding (entirely) on the newly obfuscated samples
<b>Obfuscation and Analysis: 3.7% <math>\hat{=}</math> 21/571 papers</b>	
[61, 62]	present the use of abstract interpretation to assess obfuscations
[35, 42, 63, 92, 148, 157, 182]	survey, present, and evaluate malware detection and software analysis techniques as well as obfuscations as a counter-measure
[4, 17, 182]	discuss the use of analyses tools and techniques to evaluate the strength of obfuscations
[159]*	presents novel obfuscations and new models of attacks thereon
[171]*	presents novel obfuscations and improvements to existing analyses to counter those obfuscations
[70, 132, 146, 156, 176]	present novel detection techniques to study the prevalence of obfuscations in real-world software
[174]	presents empirical studies of the effort needed to attack software protected with specific obfuscations
[15, 19]*	evaluate state-of-the-art analyses on obfuscated code and present novel mitigating obfuscations
<b>Deobfuscation and Analysis: 3.9% <math>\hat{=}</math> 22/571 papers</b>	
[12, 23, 41, 80, 83, 160, 167, 180, 186, 194]	present or build on third-party (library) code (similarity) and cryptographic primitive detection algorithms
[50, 51, 54, 183]	present pre-pass deobfuscation for improving malware detection
[60, 86, 87]	present analysis tools that are demonstrated in manual deobfuscation uses cases
[7, 192]	present analyses and transformations that deobfuscate software as a side-effect
[27, 67]	present techniques to deobfuscate code as well as to detect code reuse from libraries and from earlier version
[189]	presents analyses of which the results are equivalent to deobfuscation but without actually deobfuscating any code, such as a data flow analysis that reconstructs the original data dependencies hidden with data flow obfuscation

Papers marked with an \* contribute novel techniques and at once evaluate the impact of countermeasures.

- *Scripts*, e.g., in JavaScript or PHP, are distributed as source code mostly for web applications.
- No *domain specific languages* are in scope as we exclude special-purpose obfuscation.

► *A significant number of papers (44 goodwill + 1 malware papers) are language-agnostic.* They do not target any particular type of programming language, either because they are surveys or theoretical papers, or because they present practical techniques in a language-independent way.

► *For goodwill, by far the most targeted languages are native languages at 54%  $\hat{=}$  210/388. Managed languages are considerably less popular at 27%  $\hat{=}$  103/388.* The latter mostly target Java and

C#. Their lower popularity is no surprise. The run-time management of these programs requires that the distributed bytecode adheres to stricter rules than required in native code, and that it is accompanied by quite some symbolic information. Protection tools for managed software have hence much less freedom to operate and to generate unconventional code than native language protection tools. The semantic gap between the managed language source code and the corresponding (protected) bytecode is hence also smaller than the semantic gap between native language source code and its corresponding (protected) assembly code. For these reasons (protected) managed bytecode is typically an easier target for reverse engineers than (protected) native code. Software developers caring for their assets and considering SP are hence incentivized to opt for a natively compiled language rather than for a managed language, and when they have the freedom to make that choice (given their other industrial requirements), they often do so in practice. It then also follows that programs written in native languages are more interesting targets for SP.

► *Only 2.3%  $\hat{=}$  9/388 of the papers study the combination of managed and natively compiled software. Given the market importance of the Android platform, we find this surprisingly few: only 1.3%  $\hat{=}$  5/388 of the papers explicitly target Android applications that contain both Java bytecode and native code [26, 67, 111, 131, 164]. We find this case important, because the aforementioned choice to embed the security-sensitive assets in the native libraries in Android applications does not fully void the need to also protect the Java part, in particular its interfaces to the native parts. If those are not protected, they can provide trivially exploitable attack vectors.*

► *Scripting languages are clearly the least popular targets of goodwill SP papers.* The reason is of course that they are in general even easier to reverse engineer than bytecode of managed languages. Only 3.9%  $\hat{=}$  15/388 of the goodwill papers specifically target them; 13 papers target JavaScript, two target PHP. JavaScript is hence clearly the most popular script language for obfuscation research.

Of the 1.3%  $\hat{=}$  5/388 goodwill papers that target all three types of programming languages, four are surveys that cover obfuscations for all types [18, 98, 148, 188], while one presents an obfuscation tool for all three types of languages [107].

Finally, one goodwill paper discusses not closely related techniques applicable to script code and techniques applicable to native code [90] and one paper focuses (mostly) on identifier renaming which is mostly applicable to both script and managed languages [11].

Overall, the prevalence of the three types of languages in our survey is similar to that reported in a previous survey [98]. For each type, the fractions we report are within 5% of theirs.

► *Of the malware papers, 51%  $\hat{=}$  94/183 target native malware. This is typically Windows malware. Next, 26%  $\hat{=}$  47/183 of the malware papers focus on code written in managed languages, mostly (Android) Java. Two papers target pure Java malware [49, 109], and 45 papers target the Java part of Android apps. In addition, 8.2%  $\hat{=}$  15/183 consider both the Java and the native part in Android apps. The 13%  $\hat{=}$  24/183 of the papers targeting script malware focus on JavaScript (11), PowerShell (5), PHP (3), Visual Basic (3), PDF scripts (1), and shell scripts (1). Finally, two papers discuss malware analyses for both script and native malware [74, 102].*

► *Finally, script languages are much more popular in malware papers than in goodwill papers, with 14%  $\hat{=}$  26/183 of the malware papers targeting script languages, while only 5.7%  $\hat{=}$  22/388 of the goodwill papers do so. Scripts language papers are then also the only type of language for which malware papers dominate the goodwill papers, with a ratio of 26:22. For native and managed languages, the ratios are 111:226 and 61:118, respectively. This is in line with what we discussed above for goodwill defenders being incentivized not to use managed and script languages if they care about their assets. In the case of malware, by contrast, the omnipresence of scripts in on-line applications and their exploitation in the real-world have effectively raised the attention of researchers, resulting in more focus on them in literature as well.*



## 2.4 Quality of Venues

We assessed the quality of the 571 papers' publication venues using the Australian CORE conferences and journal rankings [59]. The results are as follows: A\* 10.0%  $\hat{=}$  57/571, A 11%  $\hat{=}$  60/571, B 13%  $\hat{=}$  74/571, C 13%  $\hat{=}$  45/571, unranked 42%  $\hat{=}$  237/571, and workshop 14%  $\hat{=}$  81/571. In addition, 2.6%  $\hat{=}$  15/571 of the surveyed papers were published at national-only venues, and 0.4%  $\hat{=}$  2/571 at new venues not (yet) ranked by CORE. Importantly, these rankings are based on the latest available CORE databases (2020 for journals and 2021 for conferences) but venue rankings might change over time. The ranking of a venue may hence have been different at the time of publication.

A detailed study of the numbers revealed that while the distribution of papers over differently ranked venues varies from one year to another, we observed no major trends over the years. In particular, the distributions for the years 2016–2022 did not differ substantially from the distribution in the years up to 2015. In other words, the different methods we used for selecting papers up to 2015 (based on existing surveys and authors) and from 2016 onwards (additionally based on title keywords in online databases) did not have a noticeable impact on the quality of the papers included from those years as judged by the ranking of their publication venue.

► *The most important finding from our data is that papers published at top-tier venues (A\*) are predominantly from the malware category.* Although only 32% of all surveyed papers fall into the malware category, about 58% of all A\* papers are from this category. Goodware obfuscation papers are instead mostly published at specialized workshops.

► *Overall, the number of goodware SP papers at top-tier security venues is low:* USENIX Security: 5, ACM CCS: 4, IEEE S&P: 1, NDSS: 1. The cause of this can only be speculated. It coincides with the subjective feeling among many goodware SP researchers, including many participants of the Dagstuhl seminar, that it is hard to get goodware SP papers published at top-tier venues. Part of the reason could be that the evaluations and validations of the proposed techniques are less convincing to reviewers, e.g., because the used methodologies are ad hoc rather than standardized, and because the goals and provided security guarantees are much fuzzier than in other domains such as cryptography. With this survey, we shed more light on the used evaluations, such that the SP community can progress from a subjective feeling towards more objective observations.

## 3 Sample Sets

In the SP community, no consensus on the use of particular samples for the evaluation of SPs and code analysis methods exists [68]. Instead, used samples vary widely both in complexity of individual samples and overall sample set size. This results in a strikingly large number of distinct samples used in the SP literature, in stark contrast to other fields in computer science, such as in machine learning and computer vision with the omni-present data sets MNIST, CIFAR10, CIFAR100, ImageNet; in compilers with the SPEC, parsec, and DaCapo benchmark suites; in circuit design with ITC'99, MCNC'91, ISCAS'89 and ISCAS'85; and even in hardware obfuscation, where Trust Hub offers a set of labeled hardware obfuscation benchmarks [8].

This section presents the results of a fine-grained analysis of samples used in SP research. Methodologically, we included only samples used for evaluating solutions presented in a paper. Short demo programs or fragments of programs explaining an approach were omitted from the analysis. Overall, 84%  $\hat{=}$  481/571 of all papers have specified a sample set for evaluation. Among the other papers that have no sample set, there are 8.4%  $\hat{=}$  48/571 of which this is to be expected, because they are surveys, theoretic papers, and papers focusing specifically on SP tool features that do not need to be evaluated on samples, such as tool transparency for maintainers and users [40].

► *In the remaining 7.0%  $\hat{=}$  40/571 papers, which are all goodware papers and of which the vast majority present novel obfuscation techniques claimed to be practical, no samples are used for evaluating*

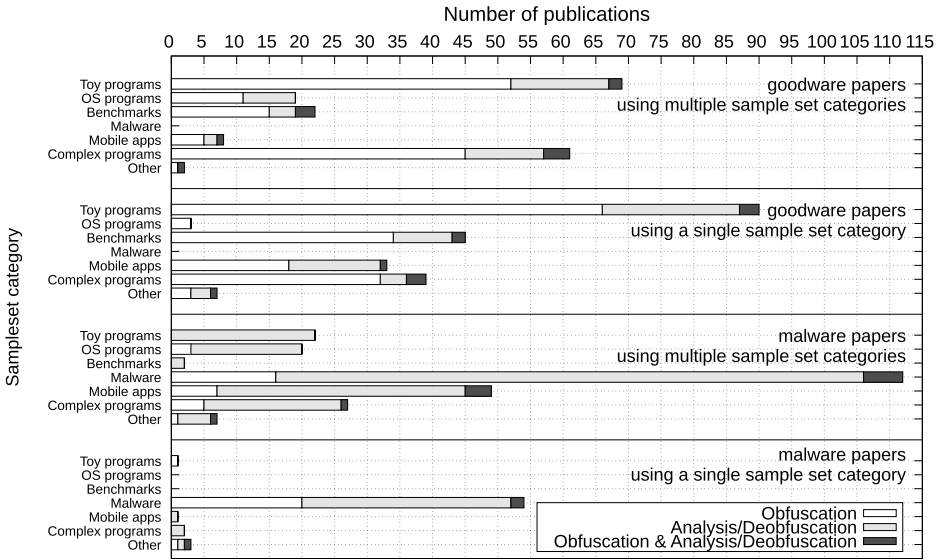


Fig. 3. Number of papers per category of sample set. The categories are non-exclusive.

the contributions. This lack of evaluation obviously is a major shortcoming of a significant number of papers. However, our data (as detailed in the supplemental material) indicates a downward trend.

### 3.1 Sample Categories

First, we analyzed which types of samples were used in the papers, including whether samples were reused, and we classified samples into distinct categories: *benchmarks*, *malware*, *toy programs*, *complex programs*, *OS programs*, *mobile apps*, *other*, and *unknown*. Figure 3 shows the distribution of sample set categories in the surveyed papers for each of four paper types: goodwill and malware papers that experiment with either one category of samples or multiple categories. Notice that some malware papers rely exclusively on non-malware samples. The (valid) reason is that these papers focus on techniques aimed at malware, but which are only evaluated on benign samples. This happens, e.g., for papers presenting repackaging detection techniques.

*Toy programs.* Toy programs are used in a strikingly high number of evaluations. Goodwill papers stand out in this regard, with  $52\% \hat{=} 159/305$  of their sample sets containing toy programs, and  $57\% \hat{=} 90/159$  of them only using toy programs for evaluation. Of all papers with samples,  $38\% \hat{=} 182/481$  include at least one toy program.

We define toy programs as small, mostly single-function programs that are not standardized, i.e., not from a known benchmark suite. They are often written for the sole purpose of performing evaluations. They include well-known algorithms for sorting and searching ( $14\% \hat{=} 26/182$ ); encoding, encrypting, and hashing ( $35\% \hat{=} 64/182$ ); and math functions such as Fibonacci and prime number generators and basic string operations ( $28\% \hat{=} 51/182$ ). Toy programs also include synthesized programs generated by tools such as Tigress and csmith ( $8.2\% \hat{=} 15/182$ ), and simple programs with a special structure such as nested loops or a certain way of branching ( $15\% \hat{=} 71/182$ ).

The use of toy programs does not always imply a threat to validity. Local protections such as **mixed Boolean-arithmetic (MBA)** can be evaluated locally within their function, isolated from the rest of a program. Thus, even simple samples can be the foundation for meaningful evaluations. However, we observe a more ambivalent picture, in which complex, non-local protections are also often evaluated with toy programs only. For example,  $26\% \hat{=} 24/91$  of all toy-program-only

papers deal with virtualization-based protection, which is not a local protection in real-world usage. Furthermore, 9 out of 10 goodwill papers that introduce techniques based on symbolic execution include toy programs in their sample set. Five even do so exclusively (the remaining paper uses a single unknown sample). It is known that symbolic execution becomes impractical as program sizes increase due to the challenge of path explosion, so when evaluations are done with small programs only, no statement about the applicability to real programs can be made.

► *The observed reliance on toy programs, in particular in goodwill papers, threatens the external validity of much research, showing it remains at the lowest **technology readiness levels (TRLs)**. The reliance on non-standardized toy programs of which the implementations are rarely published, also hinders the reproducibility and interpretation of presented research results.* For many categories of toy programs, there exist myriads of different implementations that differ significantly in terms of complexity and structure, such as when algorithms can be implemented with or without recursion. It should be a strict requirement that evaluation toy programs are made available as artifacts.

*OS programs.* Around  $8.7\% \hat{=} 42/481$  of all sample sets include OS programs, i.e., programs that can be attributed to an operating system. For Windows systems, these are bundled tools such as the calculator or notepad. For Linux, we included tools that are an integral part of a typical Linux distributions, such as the GNU core utilities. Almost all identified samples in this category have in common their comparatively low complexity and small binary size.

Samples of the GNU core utilities occur in  $33\% \hat{=} 14/42$  papers that rely on OS programs. Following the UNIX philosophy of small single-task utilities, the median **lines of code (LoC)** of the 126 utilities is 339, with the biggest program (ls) being 5626 LoC. Most real-world software that requires SP is magnitudes bigger. Core utilities hence are not representative.

► *Lacking representativeness, the use of Linux OS programs is as doubtful as that of toy programs.*

The situation is different for the  $43\% \hat{=} 18/42$  papers with Windows samples, which are typically more complex. For example, a Windows 7 calc.exe binary is almost a megabyte large. Another major difference between Linux and Windows OS samples is the code format on which obfuscations can be applied. Since their source code is not publicly available, Windows samples only have SPs applied on binary code, while Linux OS programs can be obfuscated at various build stages.

*Benchmarks.* Of all sample sets,  $14\% \hat{=} 69/481$  contain samples from benchmark suites. Almost all of them are originally intended for performance evaluations. The SPEC suites [158] are by far the most often used, at  $52\% \hat{=} 36/69$ . They include real programs with realistic inputs. SPEC CPU 2017, e.g., includes the GCC compiler, a weather forecasting program, a perl interpreter, and an x264 video compressor. The surveyed papers use samples from only two out of SPEC's current offering of 25 suites: the SPEC CPU suite (including SPECint to measure integer arithmetic performance) for binary code and the SPECjvm suite for Java bytecode evaluations. The second most used suite is MiBench, at  $12\% \hat{=} 8/69$ . This embedded benchmark suite's programs vary from single-function quicksort implementations to complex programs such as Ghostscript [89]. All other suites, such as DaCapo, SciMark, and the CompCert benchmarks are used by less than five papers.

We identified only one benchmark suite specifically compiled for SP research, from the Technical University of Munich [14]. It includes basic algorithms, other small toy programs, and programs automatically created by Tigress. Despite this benchmark suite being presented at an A-ranked conference in December 2016, only  $8.7\% \hat{=} 6/69$  papers from the surveyed corpus use it [15, 16, 71, 100, 104, 200], including four papers originating from other research institutions (with disjoint authors). This is surprisingly low, in particular considering that 64 papers published in the period 2018–2022, when everyone had long had the opportunity to learn about this obfuscation benchmark suite, still rely on other, non-standardized toy programs.

The widespread use of samples from performance benchmark suites is unsurprising, as they serve well for measuring the performance cost of SPs. However, the fact that 45 goodwill papers rely solely on sample programs from performance-oriented benchmarks and from a benchmark suite consisting only of toy programs, clearly constitutes a benchmark representativeness problem.

► *In conclusion, the community is missing opportunities to standardize their evaluation methodologies, and there is the lack of standardized, representative benchmarks for goodwill SP research.*

*Malware.* Of all sample sets,  $35\% \hat{=} 166/481$  include malware samples. Sample sets containing malware samples are hence on average the largest sets. The reason is straightforward:

► *Access to malware samples is easy with several publicly available malware repositories.* The Drebin [165] dataset with 5,560 samples from the years 2010–2012 is used most often in  $7.8\% \hat{=} 13/166$  of the papers, followed by MalGenome [190] (1,200 samples from the years 2010–2011) with  $4.2\% \hat{=} 7/166$  and Contagio [129] with  $1.8\% \hat{=} 3/166$ . Other used repositories are VirusShare.com [173], in which almost 50M malware samples are freely available, at  $5.4\% \hat{=} 9/166$ ; and AndroZoo [6] with 20M samples of goodwill and malware Android APKs, at  $3.0\% \hat{=} 5/166$ .

*Mobile apps.*  $19\% \hat{=} 91/481$  sample sets include mobile app samples. Similar to malware, mobile app samples also contribute to significantly larger sample sets than those from other sample categories.

► *There is a vast difference in the use of Android samples and iOS apps.* The easy, direct availability of Android mobile apps enables their use as evaluation samples. These apps can be batch-downloaded from various app stores in the form of APK files [79]. This is done for goodwill research, but also for malware detection/classification research, in which sample sets need to contain malicious as well as benign samples. By contrast, access to iOS app samples is much more restricted: there exists only one official app store for iOS from which apps can be obtained, and apps from the store are DRM protected and encrypted. Sharing sample sets with iOS apps is therefore both technically and legally challenging. Only two SP papers include iOS samples [176, 178].

*Complex programs.* Complex programs are included in  $27\% \hat{=} 129/481$  samples set. In this category, we collect all samples which are more complex than toy programs and do not fall into any of the other categories. These programs include well-known programs from various domains: compression or archiving programs ( $22\% \hat{=} 28/129$  papers), games ( $22\% \hat{=} 29/129$ ), chat and instant messaging applications ( $8.5\% \hat{=} 11/129$ ), media processing software ( $11\% \hat{=} 14/129$ ), webservers ( $5.4\% \hat{=} 7/129$ ), browsers, ( $4.7\% \hat{=} 6/129$ ), and databases ( $7.0\% \hat{=} 9/129$ ). Some papers also used less known programs from public repositories (e.g., GitHub) in their sample set ( $7.8\% \hat{=} 10/129$ ).

► *We found no evidence of considerable re-use of a complex sample. Moreover, the vast majority of papers do not mention the samples' specific version. Hence reproducibility is severely limited.*

A notable exception is observed in the  $4.9\% \hat{=} 19/388$  goodwill papers that present human experiments. Six of those do use the same two samples: a simple race car game and a chat client [44–46, 91, 93, 202]. These papers span 12 years, with the latter three cited papers being from different authors than the first three.

*Other.* This category unites samples that are not executable programs, which are used in  $4.0\% \hat{=} 19/481$  of all samples sets. It includes office documents, obfuscated expressions such as MBA expressions, malicious URLs, and the like.

► *In this category of samples, we do observe more reuse: MBA expression sample sets are shared and reused frequently [34, 66, 101, 114, 128, 143, 185].*

*Unknown.* Finally, we included a category unknown, which are included in  $1.9\% \hat{=} 9/481$  samples sets. We use this category for papers that clearly evaluate their contributions on executable

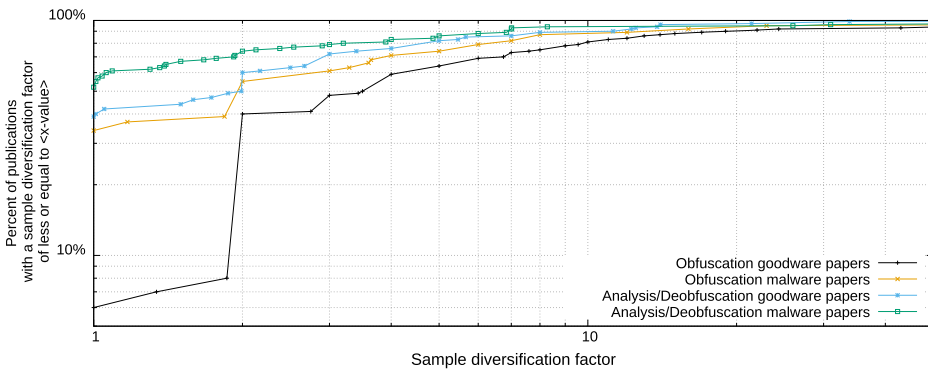


Fig. 4. Cumulative graph of ratio between total and original sample set sizes.

software, but that lack specificity to enable us to categorize that software. For example, some papers only state they use Windows applications, or applications downloaded from GitHub.

### 3.2 Sample Set Sizes

For each paper, we also counted two sample set sizes. The first is the *total sample set size*, which is the total number of different samples used in the paper’s evaluation, including variations of the same program, document, expression, and so on. The second is the *original sample set size*. This is the number of original programs/documents/expressions from which variations were generated and evaluated. In goodwill research, for example, the original samples are typically the vanilla, unprotected programs. The total sample set then contains those originals, plus a number of protected versions of each of them. When authors downloaded samples from app stores or malware repositories, we counted each downloaded sample as an original sample, even if they belong to the same malware family or if they happen to be repackaged versions.

► *The total sample set sizes range from a single sample to over two million. Large sample sets over 4,000 samples consist predominantly of the categories malware and mobile apps.* In addition, there exist a few large sample sets from samples of the *other* category, such as large sets of MBA expressions. Few papers use large sample sets from the remaining categories. For instance, Salem and Banescu [145] have automatically generated a large number of toy samples with the help of the *Tigress* obfuscator to create a sample set of over 11,000 samples.

► *By contrast, one third of all evaluations (36%) feature small total sample sets of at most 20 samples.* Eleven evaluations are based on only a single sample, which originated from one of four sample categories: *complex programs* (4), *toy programs* (3), *malware* (3), and *unknown* (1).

We further analyzed the ratio between the total sample set size and the original sample set size (i.e., how many variations of the original samples were generated). Figure 4 shows the ratios between the two sample set sizes for all four combinations of goodwill/malware and obfuscation/deobfuscation-analysis papers. Unsurprisingly, a majority (51%) of the malware analysis papers has a diversification factor of 1: these papers only evaluate original malware samples and original benign apps, but no variations of either of them, to evaluate malware detection rates. This number is in line with  $54 \approx 99/183$  of the malware papers being malware only papers. Many malware papers, in particular the ones focusing on techniques that can also be used on goodwill, also evaluate their techniques on samples they built themselves, and then evaluate more than one version, such as an unprotected one and a protected one. That is why the green line gradually rises from 51% to 100%. Noteworthy is the big step on the black curve at two for goodwill obfuscation papers. 9% of the papers feature a diversification factor of less than 2, meaning that they do not

even consider perform measurements on two versions (unprotected vs. protected) of all of their samples. An additional 31% of the goodwill obfuscation papers only reaches a diversification factor of 2, which in almost all cases corresponds to one unprotected and one protected version for each sample.

► *This means that 40% of the goodwill obfuscation papers do not evaluate diversified protection at all.* In other words, they do not evaluate the sensitivity of protection strength or costs to configuration parameters or to random seeds used in stochastic approaches, such as the parameters or seeds that might determine precisely where a particular protection is injected, which might be in a hot (i.e., frequently executed) part of a program or in a cold part, or the parameters that determine how sparsely or densely some transformation is deployed. We find this a major shortcoming, that plagues too many papers in this research domain.

► *For malware obfuscation research, where authors typically also generate their own samples, the situation is even worse, with close to 70% of those papers not evaluating diversified protection.*

### 3.3 Correlation with Publication Venue

We analyzed the relation between the quality of the evaluations (number of samples, number of different sample categories, sample complexity and diversification) and the rankings of the venues at which the works were published. For malware papers, we did not make interesting observations.

► *For goodwill papers, we did observe clear correlations between publication venue quality and sample set quality and size.* In this category, the total number of samples used in an evaluation of obfuscations correlates with the ranking of the venue. The median sample set sizes go from 51 for A\* down to 21 for C-rated venues. In the supplemental material, we analyze this relation in more detail. In addition, we observed that the share of toy programs is significantly higher in evaluations of goodwill publications in the C, unranked, and workshop categories (34–48%) compared to B-ranked or better (20–25%) venues.

► *The lack of diversification, in particular in papers that present novel obfuscation, is not limited to lower-rated publication venues. To the contrary: the problem appeared almost as frequently in A\*/A papers as it occurred in other papers.*

### 3.4 Identified Challenges

In our analysis of sample sets, we identified three relevant challenges.

- **Lack of real programs in the evaluations:** Simple toy samples are omnipresent in SP research. Samples from other categories such as the GNU core utilities are also of low complexity. Such samples seem unsuitable for measuring the strength of a SP or its applicability in real software, which typically is much more complex. The only exceptions are local protections such as MBA which can be evaluated independently of the program which they are intended to protect. However, our data clearly shows that even for non-local protections and analysis methods, often low-complexity samples are used. In top publication venues, this issue occurs less frequently, but still frequently enough to be worrisome.
- **Limited sample availability:** Depending on the sample set category, reproducibility in SP research is most often low, including in top publication venues. In contrast to other research domains such as machine learning, where large datasets are publicly available, in SP research no commonly-used sample sets exist. This makes it difficult to reproduce results and to compare different approaches. For toy programs, many different implementations are used (rather than the ones available in a benchmark suite [15]), and of complex programs the used versions are most often not specified, all of which makes it difficult to compare, interpret, and reproduce results. For OS programs, we observe more reuse,

primarily due to the easy availability of the GNU core utilities. Another exception is recent work on MBA, where reuse of expressions exists (e.g., the public sample set of Syntia [34]).

- **Limited diversification:** Evaluations by and large neglect diversification of the protection deployment. This issue plays in particular in goodware papers that present novel obfuscations, including in those published in A\*/A venues.

To tackle these challenges, we advocate for a collaborative compilation of open sample sets within the research community. A publicly available set of samples that takes into account the different motivations and goals for protections would assist researchers in evaluating new protections, contribute to the validity of research, and improve the comparability and reproducibility of research results. It should furthermore be of sufficient size to increase its credibility for A\*-rated conferences. (If necessary, authors can then still pick a small subset of this sample set to do exploratory research.) Moreover, the standardized sample set should come with guidelines on how to diversify the deployment of protections on the samples, e.g., by specifying small, medium, and large sets of functions in the samples that are candidates for obfuscation, similar to how the performance benchmark suites include testing, training, and references inputs.

## 4 Sample Treatment

The protected samples discussed above have been compiled and built using certain compilation tools. They feature combinations of protections that were deployed by protection tools that transformed one or more representations of those samples. In addition, the last phase of the samples' treatment is the application of code analysis, deobfuscation, and classification techniques on them. This section analyzes the representations of the samples that were used to deploy protections, the tools that were used to do so, the protections that were deployed with those tools, and the analysis methods used to reverse a protection and to reduce its protection potency and/or resilience.

### 4.1 Protection Code Representation

For categorizing papers according to the code representation (i.e., format) on which the studied protection transformations are applied, we used similar categories as in Section 2.3 for the types of programming languages. First, *Source code* can be rewritten to inject protections, e.g., with the C-to-C rewriter Tigress. Secondly, *Native code* in the form of assembler or binary object code can be transformed, e.g., with link-time binary rewriters such as Diablo. Finally, protections can be deployed on *Intermediate code* formats. These more symbolic formats can be either compiler intermediate representations such as LLVM's bitcode, or the bytecode distribution formats of managed language, such as Java bytecode or the **C Intermediate Language (CIL)**.

For the 79%  $\hat{=}$  308/388 goodware and 96%  $\hat{=}$  175/183 malware papers that report measurements on protected samples generated by transforming some code format, the middle bar in Figure 5 displays how many papers apply protections on the different formats, per the different programming language types. This section discusses only the most striking observations.

First, 10%  $\hat{=}$  40/388 of the goodware papers are not considering any specific language or protection code representation. These include theoretical papers (e.g., on using abstract interpretation to model protection strength), surveys, and a few papers on aspects of SP tools that do not need to be evaluated on samples, such as making a tool's transformation transparent for maintainers and users [40].

► *Another 10%  $\hat{=}$  40/388 of the goodware papers lack an evaluation on samples, despite the fact that 36 (90%) of them do present novel obfuscation techniques their authors claim to be practical.* This is a rather negative result that clearly shows how the field of goodware software protection often fails to maintain the highest standards. In malware research, that problem does not occur. The

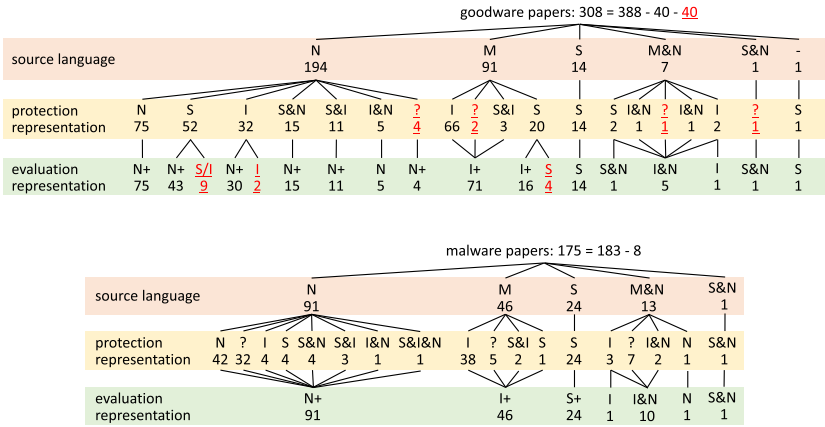


Fig. 5. Sample protection and evaluation languages. Same legend as in Figure 2, and I = Intermediate formats, ? = unknown, + = and more, & = and, / = or. Red underlined data indicate cases we consider problematic.

4.4%  $\hat{=}$  8/183 of malware papers that do not feature an evaluation on protected samples are all surveys and theoretical papers.

► 2.6%  $\hat{=}$  8/308 of the goodware papers fail to clarify which code format is being transformed for their evaluation, and hence lack this critical information for reproducibility. On the malware side, 25%  $\hat{=}$  44/175 of the papers similarly do not report a representation used to obfuscate the samples. This is to be expected and is okay, however, as malware papers typically use real-world malware samples which can be found online in various datasets for reproducing the research results, even if their provenance is not documented.

► Most importantly, goodware papers targeting natively compiled languages understudy the composition and layering of protections applied at different code representations, i.e., at different levels of abstraction.<sup>2</sup> Transformations on more abstract program representations such as source code early on in the build process can protect more abstract properties (e.g., invariants) and can be selected based on more abstract program features such as type information. Moreover, because the injected protections' code is then compiled together with the application code, it can be integrated more stealthily. The freedom to transform code is then restricted, however, by the conventions to which the higher-level representations need to adhere (language or IR specification compliance). Complementary thereto, obfuscating transformations applied on lower-level code formats can completely break the conventional mapping between higher-level software engineering constructs and lower-level software artifacts to mitigate decompilation and human comprehension and to prevent, e.g., that disassemblers produce correct CFGs. The detailed analysis presented below shows that such complementarity has hardly been researched despite its importance [110].

First, only 7.7%  $\hat{=}$  15/194 goodware papers targeting natively compiled languages consider both source-level and binary-level transformations (S&N). Among them are two papers that only compare the strength of their source-level transformation against other binary-rewriting-based alternatives [147, 177]. Among the five papers that actually compose source-level and native-level transformations, one paper evaluates return-oriented-programming obfuscation through binary rewriting on programs already obfuscated at source-level by Tigress [37]. The other four papers

<sup>2</sup>For this analysis, we consider multiple protections to be composed/layered if they are deployed together in at least one sample. Papers never applying multiple protections together on at least a sample are hence not considered to be composing or layering protections.



originate from the ASPIRE research project in which both source-level and binary-level transformations are composed on the same samples [2, 3, 22, 141]. The remaining eight papers all have a deobfuscation/analysis perspective and rely on samples that are either protected with source-level obfuscators such as Tigress or with binary-level tools such as VMProtect or Themida, but never with both. We thus identified only two research efforts that layered source-level and native-level obfuscations on C/C++ goodwill.

All 5.7%  $\hat{=}$  11/194 papers that consider both source-to-source rewriting and compile-time transformations (S & I) for native languages use Tigress and Obfuscator-LLVM to generate samples [15, 69, 85, 136, 161, 162, 166, 168, 169, 199, 200]. Only 2 of those 11 papers compose protections of the two tools on the same samples [85, 136].

Of the 2.6%  $\hat{=}$  5/194 papers that consider compiler-level and binary-level obfuscation, only two actually compose them: One composes compiler-based control flow obfuscation with assembly-level code layout randomization [78], and one combines a high-level obfuscation based on multi-threading implemented in a compiler with packing applied at the binary level [152].

► *So overall, only 4.6%  $\hat{=}$  9/194 of papers on native goodwill from only five projects/teams explicitly target tool flows that deploy protections at multiple software abstraction levels. This gap between published research results and best practices in industry [110] is a major shortcoming of this field.*

## 4.2 Deployed Protections

For each of the papers, we collected the types of protective transformations and features thereof such as relying on aliasing, that were deployed or theorized about beyond merely discussing them as related work. We classified them in 31 different classes, of which descriptions are provided in the supplemental material, where we also discuss some subtleties of our information gathering process. Our classification was derived from previous work by Collberg et al. [57], and from three surveys [73, 98, 148].

For each protection, we distinguish between theoretical considerations as found in surveys or qualitative security analyses on the one hand and practical implementations that actually get deployed on samples on the other hand. In the remainder of this section, we only consider the practical implementations. For the interested reader, the supplemental material presents a number of tables with detailed counts of occurrences of protection combinations.

► *The popularity (i.e., the relative deployment frequency) of the considered obfuscation classes differs between malware and goodwill and between different types of programming languages.*

Overall, data encoding/encryption (138) is the most frequently used SP technique, followed by identifier renaming (126), junk code insertion (117), opaque predicates (116), code diversity (112), and packing/encryption (105). Some often-studied SPs such as data encoding/encryption, packing/encryption, junk code insertion, and also repackaging are more popular for malware than for goodwill. Vice versa, opaque predicates are relatively more frequently researched in goodwill than in malware, as are control flow flattening, class-based transformations, data flow transformations, aliasing-based techniques, MBA, code mobility, loop transformations, server-side execution and hardware-supported obfuscation. This is in line with the distinct goals for which SPs are used: for preventing reverse engineering, code comprehension, and tampering in goodwill; and for preventing detection and classification of malware.

Few protections are popular for all types of programming languages: Only data encoding/encryption, junk code insertion, code diversity, packing/encryption, control flow transformation, and function transformations are all deployed in 10% or more of the papers on each type of language. For the interested reader, the supplemental material presents a more extensive analysis of which protections are popular for which languages, complementary to the above.

► *Aliasing is rarely exploited (explicitly), even though this SP feature is often cited as increasing the resilience of SPs due to alias analysis being an intractable problem [58].*

Only 4.0%  $\hat{=}$  20/495 of papers that deploy protections exploit aliasing to complicate analyses and to make analysis results less precise. This feature is much more popular in managed languages than in natively compiled, let alone script languages. The reason is that the computation of precise call graphs for Java bytecode depends heavily on type inference and points-to set analysis. Various transformations have been presented to increase points-to set sizes and to hamper type inference and call graph reconstruction. Most script languages are dynamically typed, and no C++-specific obfuscations exist. Hence, transformations that aim for points-to set increases are not researched for those types of languages. What remains is the use of conventional pointer aliasing to hamper data and control flow analysis, but very few papers target this.

► *For most protections, we identified a balance between papers that present protections from an obfuscation perspective vs. papers that consider and evaluate them from an analysis/deobfuscation perspective, meaning that comparable attention is given to those protections from the two perspectives.* However, we also found that some protections are much more likely to be studied from the perspective of protection than deobfuscation or analysis, and vice versa. The biggest imbalance towards a protection perspective can be seen for hardware-assisted protection (10:0). While 10 papers present and implement new hardware-assisted protection techniques, none target deobfuscation or analysis of this type of protection. Other techniques with a big imbalance towards the obfuscation perspective are code mobility (12:2), aliasing (16:5), server-side execution (8:3), data-to-code conversion (23:11), and class-based transformations (32:15). Large imbalances towards deobfuscation and analysis occur less often. The biggest one is observed for repackaging (12:24). Some of the imbalances can be explained by taking into account that the number of defensive papers (453) in our survey is much higher than the number of offensive papers (151), and the fact that certain protections are used relatively more frequently for malware than for goodware or vice versa. Notable exceptions are virtualization, which is included 59 times in defensive studies and 38 times in offensive studies, and MBA is included 11 times in defensive research, and 12 times in offensive research. We conjecture that this is due to the fact that these obfuscations were at some point considered among the strongest ones in the past and widely deployed in practice, which made them both relevant and challenging cases for (academic) offensive research.

► *Few papers explicitly layer protections on each other. SP research, in particular for goodware, is clearly lacking in this regard.* While quite some papers consider and evaluate multiple classes of protections, most of them deploy the different protections on different samples, e.g., to compare the protections' strength, rather than evaluating their combined/layered deployment within the same samples. For both goodware and malware, about one quarter of the papers only deploy a single protection. Moreover, about 72%  $\hat{=}$  278/388 of all goodware papers and about 61%  $\hat{=}$  112/183 of all malware papers deploy three or fewer protections, and about 85%  $\hat{=}$  484/571 deploy five or fewer protections, on mostly disjunct sets of samples as discussed. On average, few protections are hence considered in the surveyed papers. Slightly more are considered in papers from A\*/A-rated venues, but the difference to lower-rated venues is small. This is the case despite the common wisdom that strong protection can only be obtained if multiple protections are combined and layered. For example, commercial tools such as Cloakware [99] and Dexguard [84] support and explicitly recommend the combination of multiple different protection techniques in so-called protection recipes in their user manuals and whitepapers in order to mitigate various attack strategies.

► *Only in a small minority of the papers, contributions get evaluated in protected samples that are representative of real-world SP usage.* This lack of mature evaluation is a shortcoming for deobfuscation and analysis papers that are evaluated on unrealistically weak protected samples, as well as for obfuscation papers, many of which apparently do not evaluate the complementarity of their

contributions over existing ones. In particular, in goodwill research, where the authors create protected samples themselves and thus can provide clear descriptions of the deployed protections, the numbers are appalling. Obviously, not all papers that limit the evaluation to a single or few protections are problematic. In particular, for all fundamental research at low technology readiness levels (TRLs), it can be perfectly reasonable to evaluate techniques on unrealistically simple use cases. Our numbers do indicate, however, that the field of goodwill SP is running behind in demonstrating that the presented techniques are capable of achieving a high TRL.

Two protections stand out as being deployed in isolation in analysis/deobfuscation papers: virtualization for goodwill and packing/encryption for malware. As for the latter, this is mostly due to the papers not containing all ground-truth / provenance information: Many malware papers mention that their real-world malware samples are packed, either because they specifically target packed malware or because an unpacking step is included in their tool flow, but do not mention any other techniques that are deployed on those samples because the authors do not know which ones have been applied and because their analysis is (sufficiently) orthogonal to other SPs.

As for the high frequency with which virtualization is studied in isolation in goodwill analysis/deobfuscation papers, there are certainly a number of authors that simply have opted to target programs protected with only virtualization. But to some extent, authors also were forced to do so: on natively compiled code, the early virtualization tools, both academic (e.g., based on Strata) and commercial, transformed binary code. For the commercial ones, such as VM protect, it is not obvious which additional SPs they deploy on top of virtualization, and hence it is hard to report on those in papers. Moreover, binary rewriting tools are not easily combined with other obfuscation tools to layer SPs. Up to 2015, this definitely impacted research.

From 2015 onwards, this changed. Of the goodwill papers on natively compiled code, 40%  $\hat{=}$  63/194 deploy virtualization, of which 40%  $\hat{=}$  25/63 use Tigress, the source-rewriting obfuscator that allows one to compose virtualization with a range of other SPs and became available around 2015. Of those 25 papers, only two are virtualization-only papers, one of which is a Systematization of Knowledge paper on automated virtualization deobfuscation [103]. In other words, once the necessary tool was available to compose virtualization with other SPs, researchers effectively started doing so.

► *This example of how tool availability improved the research methodology stresses the need for reusable tools in SP research, an aspect that will be discussed further in Section 4.4.*

### 4.3 Employed Analysis Methods

Similar to how we analyzed the papers' deployment of protections, we classified their use of code analysis methods. These are used by MATE attackers and malware analysts in practice and can hence be used by researchers to evaluate their obfuscations' practical strength. In total, we considered 28 different methods and features thereof, which were again derived from previous work [57, 73, 98, 148]. A list of all analysis methods and short descriptions is provided as supplemental material. In the collected data, we again differentiate between theoretical discussions and actual deployment on samples. Here, we only report on the latter.

► *A major observation is that more than half (54%  $\hat{=}$  133/248) of the goodwill obfuscation papers that deploy protections on samples do not evaluate those protections' strength by assessing how analysis methods fare against them. Moreover, 22%  $\hat{=}$  54/248 only evaluate one analysis method, leaving only 25%  $\hat{=}$  61/248 that evaluate more than one method. These numbers are inflated by lower-quality papers of lower-quality publication venues, but the observation is certainly not limited to such papers: 27%  $\hat{=}$  11/41 of the goodwill obfuscation papers published in A\*/A rated venues do not deploy any analysis methods. This is strikingly low, given that (layered) SP is all about mitigating the attack paths (plural) of least resistance, including attacks on the deployed*

protections. While it is perfectly acceptable for authors presenting a new SP to aim to mitigate only one existing attack strategy, we do think it deserves a recommendation to evaluate the impact of the SP on multiple attacks, in particular attacks on the new SP itself. In other words, beyond the potency of an obfuscation to hamper one attack strategy, also its resilience against other attacks [48] should be evaluated. This is clearly not done in papers in which no or only one analysis method is evaluated.

In the obfuscation malware papers, the numbers are better, with  $81\% \hat{=} 38/47$  of them reporting on at least one analysis method, but still  $45\% \hat{=} 21/47$  of them consider only one such method.

In the analysis/deobfuscation papers, there, of course, is no issue of papers not evaluating analyses. The average number of analysis methods per paper (with at least one analysis method) is also significantly higher in those papers (3.3) than in obfuscation papers (2.0).

Overall, control flow analysis (96), pattern matching (96), tracing (79), statistical analysis (72), and machine learning (66) are the top five methods, the latter being a feature rather than an analysis.

► *Similar to what we noticed for deployed protections, we observe significant differences between the types of analysis methods used in obfuscation papers and in deobfuscation/analysis papers.* For example, pattern matching is used in 72 analysis/deobfuscation papers. This clearly indicates that in some scenarios, this rather generic technique is considered a highly relevant alternative to more targeted analyses. In stark contrast, only 33 papers that introduce new protections evaluate their strength against pattern matching. Similar observations can be made for statistical analysis (16:60) and machine learning (15:55). To some extent, these effects are driven by some analyses' predominance in malware research. When we exclude malware papers from the analysis, the imbalance between obfuscation and deobfuscation/analysis papers is less pronounced. However, these observations still add to our concern as to whether protections are sufficiently being evaluated against path-of-least-resistance attack and analysis methods.

As for pattern matching, in goodwill papers the ratio is still 9:22. Schrittwieser et al. [148] previously observed that pattern matching is easily mitigated, even with very simple obfuscations. However, the scope of their observation is limited to the use of pattern matching to identify artifacts/assets in the original programs, excluding its use to identify deployed protections. In other words, they observed that even simple obfuscations can be *potent* against pattern matching but did not discuss the *resilience* of obfuscations against pattern matching-based deobfuscation. Our observation that pattern matching is among the more popular techniques in analysis/deobfuscation goodwill papers confirms it is considered a viable technique to attack SPs, which makes it all the worse that pattern matching is hardly being evaluated in obfuscation goodwill papers.

► *This confirms our earlier point on defensive goodwill research lacking in evaluations of resilience.*

A second observation by Schrittwieser et al. [148] was that dynamic analyses are significantly more effective against dynamic protections than static analyses. They defined static protections as ones that execute exactly the code present in the static binary, while dynamic protections perform additional code transformations at run time (e.g., unpacking). Following up on their observation and to verify whether it still applies to our more recent data, we analyzed how virtualization, the most popular dynamic protection method in the surveyed papers, is being targeted by analyses. Our findings indicate that against virtualization, indeed, predominantly dynamic analyses such as trace-based techniques and symbolic execution are used, sometimes supplemented by human analysis. However, our data also reveals that static analysis techniques like lifting, control flow analysis, data flow analysis, normalization, and slicing are also frequently utilized in the analysis and deobfuscation of virtualization-based protections. A crosstab analysis (for which the data is available in the supplemental material) provides a robust explanation, namely that these static

Table 4. The 39 Tools used in the Experimental Evaluation in Six or More Papers

Tool	History	\$	O	F	B	A	P	T	Tool	History	\$	O	F	B	A	P	T
LLVM		●	●	●	71	22	57	73	Allatori		●	○	○	-	-	10	10
IDA Pro		●	○	●	-	63	-	63	DashO		●	○	○	-	-	10	10
GCC		●	●	●	52	3	1	52	Jad		●	●	●	-	10	-	10
Tigress		●	○	●	-	-	37	37	Sandmark		●	●	●	-	-	10	10
OLLVM		●	●	●	-	-	35	35	UPX		●	●	●	-	-	10	10
Diablo		●	●	●	-	3	24	24	objdump		●	●	●	-	9	-	9
VirusTotal (AV)		●	○	○	-	24	-	24	Syntia		●	●	●	-	9	-	9
PIN		●	○	●	-	20	-	20	Triton		●	●	●	-	9	-	9
ProGuard		●	○	●	-	-	19	19	BinDiff		●	○	○	-	8	-	8
Soot		●	●	●	-	14	6	19	Dex2Jar		●	●	●	-	8	-	8
Clang		●	●	●	17	-	1	18	TXL		●	○	○	-	-	8	8
VMProtect		●	○	○	-	-	18	18	Z3		●	●	●	-	8	1	8
KLEE		●	●	●	-	16	-	16	ACTC		●	●	●	7	-	7	7
Eclipse		●	●	●	2	8	6	15	AndroGuard		●	●	●	-	7	-	7
Visual Studio		●	○	○	14	4	1	15	Arybo		●	●	●	-	7	-	7
Themida		●	○	○	-	-	14	14	McAfee (AV)		●	○	○	-	7	-	7
angr		●	●	●	-	13	-	13	scikit-learn		●	●	●	-	7	-	7
OllyDbg		●	○	○	-	13	-	13	Zelix Klassmaster		●	○	○	-	-	7	7
APKTool		●	●	●	8	4	-	12	QEMU		●	●	●	-	6	-	6
Code Virtualizer		●	○	○	-	-	12	12									

Columns A = Analysis, B = Build, and P = Protection list how many papers use a tool for each purpose. T = Total gives the number of publications using the tool. The \$ column indicates whether the tool is ● = free to use, ● = has a demo version, ○ = paid version only. Column O describes source-code access: ● = open-source, ○ = upon request, ○ = closed-source. The F column describes how flexible/adjustable the tool is: ● = adaptable/extendable, ● = plugins, ○ = configurable, ○ = rigid. Column History = Usage per year since 2012 (left to right).

techniques are often used in conjunction with the identified dynamic techniques. For instance, symbolic execution is often combined with lifting, normalization, and data flow analysis.

► *We, therefore, conclude that the claim by Schrittwieser et al. [148] about dynamic analysis techniques in the face of dynamic protections should be refined somewhat. Dynamic analyses, indeed, are stronger, but they seem to complement static analyses rather than replace them.*

#### 4.4 Used Tools

As already hinted, the existence and availability of tools influence how authors treat their samples. This section discusses the tool use in papers with sample sets, covering tools used for building, protecting, and analyzing samples. Our analysis includes tools that are newly introduced in a paper, as well as tools that are re-used as is or built upon. For example, when a work deploys **Obfuscator-LLVM (OLLVM)** on samples, we count this towards OLLVM as well as towards the LLVM framework on which OLLVM builds. A caveat is that not all authors are specific and complete in describing their tool use. For interested readers, the supplemental material provides additional information on the tools, including recommendations regarding their use for evaluating SPs.

► *In general, authors do not share or reuse research artifacts much.* At 768, we counted more tools than papers, and 74%  $\hat{=}$  532/768 of them are used in only one paper. Only the 5.7%  $\hat{=}$  39/768 listed in Table 4 are used in at least five papers. We discuss several interesting aspects in more detail below.

*Tool Popularity.* LLVM (71 papers) and GCC (52) are by far the most popular build tools, followed by Clang (16), Visual Studio (14), and ACTC (7). Beyond those, the only additional compilers being mentioned are CompCert [31, 33, 149], g++ [53, 130], and tinycc and TenDRA [149].

► *Too few authors mention the compilation tools they use to build their samples.* Only 126 papers provide this information, of which 111 target natively compiled code. So 45%  $\hat{=}$  91/202 of the goodwill papers that target at least natively compiled code do not mention which compiler they used to generate samples, let alone how their compiler was configured, e.g., with respect to the

optimization level. Given how strongly optimized code can differ from non-optimized code, this lack of information in so many papers is astonishing and an important issue.

Unlike GCC, which is only used once to deploy protections [78], LLVM is frequently used for this purpose (57 papers). Its more modular compilation pipeline design enables obfuscating IR transformations at various stages. The popular OLLVM obfuscator builds in this, (35), and so do other obfuscators developed on top of LLVM (22). While the Clang compiler typically serves as a front-end for LLVM IR, it is also used to inject protections with source-to-source rewriting [2].

In addition, LLVM is used in the popular KLEE dynamic symbolic execution engine (16 papers). This demonstrates that core software analysis and transformation infrastructure can be re-used for building, protecting, and analyzing software. Other multi-purpose tools are the Java bytecode rewriting framework Soot (e.g., in the Dava decompiler (4 papers)), and Visual Studio and Eclipse, which are used for building programs as well as for dynamic analysis (debugging). Eclipse's code refactoring support is also used to protect programs [196, 197]. Finally, beyond being used for compilation, GCC is used to diversify software (e.g., by compiling at multiple optimization levels), and as an attack tool to undo source-level obfuscations [32, 127, 128, 134].

► *LLVM and Tigress are currently the only popular protection tools.* The *History* data in Table 4 shows that other SP tools are declining in popularity: only a few recent papers still use ProGuard after an initial peak of activity; a similar fate has befallen Zelix Klassmaster, Sandmark, and Diablo.

► *Table 4 hints for two reasons for tool popularity: their origin and their low cost.* As for the cost, the only commercial SP tools used at least five times (VMProtect, Themida, Code Virtualizer, DashO, Allatori, and Zelix Klassmaster) all offer a demo version. All other SP tools in Table 4 are freely available. Ten of them come from academia: LLVM, OLLVM, Tigress, Diablo, Soot, Clang, KLEE, Sandmark, Obfuscapk, and DroidChameleon. Relatively absent are the *advanced* commercial offerings from, e.g., Irdeto, Digital Ai (formerly known as Arxan), and GuardSquare. The latter offers the free but limited-functionality ProGuard and the commercial DexGuard. ProGuard is used in 19 papers, DexGuard in only 3. Of the other tools commonly used to protect software, VMProtect and Code Virtualizer only offer specific obfuscations, namely virtualization. Themida virtualizes and encrypts code fragments, and the injected execution engine embeds several techniques to prevent analysis, but it performs no further transformations of the original application code.

The popularity of academic protection tools raises a question: Is their popularity due to their widespread adoption across the research community, or are their own authors prolific publishers?

► *A detailed analysis found that Tigress and OLLVM have seen widespread take-up in the community:* only 3 out of 37 Tigress papers have a connection with Tigress' developers, and only 1 out of 35 OLLVM papers have a connection with the OLLVM authors. This is in stark contrast to Diablo, the third most popular academic protection tool, for which 18 out of 24 papers originate from within the team that developed Diablo. Another binary rewriter, PLTO, suffers a similar fate: It has only four uses in our dataset, of which two originate from its own research group. We conjecture that this lack of reuse of (post-)link-time rewriters like Diablo and PLTO is, to a large, degree due to the difficulty of using and maintaining such binary-rewriting tools.

► *We conclude with the major problem that malware research involves very little deployment of commercial-grade protection tools, even in papers in A\*/A publication venues.* It implies that the strength of commercial SP software is obscured by a lack of publicly documented evaluations thereof. The root cause of this problem is, of course, that commercial protection tool vendors do not allow independent researchers to evaluate their tools and publish about it, e.g., by means of restrictions in their end-user license agreements.

On the analysis tool side, by contrast, the most popular tool is commercial: the interactive binary disassembler IDA Pro. Being relatively cheap and having a free demo version might contribute to its popularity. The competition that has surfaced in recent years with Binary Ninja (released in

2016) and Ghidra (released in 2019) has not yet made much of a dent in IDA Pro's popularity: Ghidra was only used one time in the surveyed papers, Binary Ninja two times. The most popular academic binary code analysis tool is angr. In 2016, this tool has been put forward as a unifying binary analysis framework [154]. In practice, however, in the period 2018–2022, well after the publication of angr, IDA Pro was used in 19 papers, angr in 11, Triton in 7, Pin in 8, and so on.

► *So clearly, many different analysis tools and frameworks keep being used. There is no sign of overall unification or standardization in their use for evaluating the strength of SPs on native binaries.*

When angr is used, it is most often for symbolic execution of native code. KLEE is another tool for symbolic execution of natively compiled code that has remained almost as popular as angr, with nine deployments in the years 2018–2022. KLEE operates on LLVM IR code, which is sometimes obtained by lifting binary code [36, 104], but almost always by compiling source code. Fortunately, most authors seem to understand that one can question how accurately such KLEE results reflect real-world attacks on binary code, and at least from 2018 onwards, they all complement their KLEE results with experimental data obtained with angr, IDA Pro, or other binary analysis tools.

*Tool Versions and Configurations.* Some tools have been in use for a long time. Some did not evolve significantly over that time, such as OLLVM, which saw only one release. Other tools evolved considerably, in which case reproducibility requires that the used versions are reported.

► *We observed that all too many papers lack in the reporting of used versions and configuration.* For example, for IDA Pro, 23%  $\hat{=}$  14/63 papers specified the version number; for the source-to-source obfuscator Tigress, 23%  $\hat{=}$  8/37 did so. For Tigress, the situation is improving over time, as 28%  $\hat{=}$  5/18 papers report a version in the period 2020–2022. All papers using Tigress mention at least which SPs were deployed with it, but in many cases, the authors omit the used configuration options, of which Tigress offers a wide variety. Some mention they used default configurations, but as these evolve over time, that is insufficient for reproducing the research and for interpreting the results.

The worst tool with respect to version reporting is VirusTotal. Users provide software samples to this online service, which are then scanned by a set of malware detection engines. Both the deployed engines and their versions have evolved over the years. In quite some papers, VirusTotal is used to measure the extent to which concrete obfuscations hinder malware detection. To interpret the result in those papers, it is paramount to know which version of VirusTotal was used, i.e., when the service was accessed. However, only 8.3%  $\hat{=}$  2/24 of the papers using VirusTotal do so.

*Tool Flexibility.* Some tools offer their users more flexibility than others. This does not imply, however, that this flexibility is exploited in research. In fact, we observed quite the contrary.

► *SP researchers mostly use tools as is, even the flexible ones, rather than customizing them the way attackers do.* Consider the Tigress obfuscator, of which the developers share its source code with colleagues in academia on demand. Still, we observed that no outsider papers (i.e., not involving members of Christian Collberg's team behind Tigress) discuss or evaluate extensions of improvements of Tigress' transformations. At most, other authors combine Tigress protections with their own or with other existing tools. Despite OLLVM being open source, we made similar observations, albeit to a lesser degree: 7/35 OLLVM papers extend OLLVM.

As another example, consider IDA Pro. This closed-source tool is extensible through a plugin and scripting interface. Like all disassemblers that are by and large developed to reverse engineer non-obfuscated binaries, IDA Pro can easily be thwarted with control flow obfuscations [112, 171]. It has also been shown, however, that it is fairly easy to improve those tools' handling of obfuscated code with custom extensions [171]. That is also how professional penetration testers customize them [44, 47, 48]. However, of the authors using IDA Pro, only 32%  $\hat{=}$  20/63 extended its

functionality with custom plug-ins or reused ones. Of those, 22%  $\hat{=}$  14/63 only extend the functionality with BinDiff, a special-purpose analysis tool to compare binaries. BinDiff merely builds on IDA Pro's disassembly to match code across different binaries rather than improve the disassembly of IDA Pro itself. This leaves only 9.5%  $\hat{=}$  6/63 papers extending or customizing IDA Pro in one way or another. Notably, of the 24%  $\hat{=}$  15/63 papers that use IDA Pro and that specifically target disassembly and CFG reconstruction, only two papers extend IDA Pro's functionality with regards to disassembly [21, 171]. Others, such as the seminal static disassemblers thwarting paper [112], make no such attempt.

► *We consider this discrepancy between published SP research and industrial SP practice a major shortcoming of the research in this field.* It is linked to our earlier remark in Section 4.2 about SP being a cat and mouse game in which researchers fail all too often in anticipating even the simplest custom extensions of the functionality they propose to thwart with a new SP.

#### 4.5 Identified Challenges

In our analysis of sample sets, we identified six relevant challenges.

- **Gap between analysis methods used in analysis papers and evaluations of newly proposed protections:** The used analysis methods differ significantly between analysis/deobfuscation papers and obfuscation papers. While several simple analysis methods are frequently used in analysis/deobfuscation papers, they are much less common in obfuscation papers. This is the case even in A\*/A papers. This leaves open the question of whether appropriate analysis methods are chosen for the evaluation of newly presented protections.
- **Gap between tools used in published research and the commercial, industrial state of the art:** Our analysis of tools used in the surveyed papers across all publication venue rankings identified a concerning absence of commercial state-of-the-art protection tools. While in analysis papers, well-known commercial tools (e.g., IDA Pro) are heavily used, commercial protection tools such as Irdeto, Arxan, and GuardSquare are rarely applied, even in papers in top publication venues. And even the used analysis tools are most often not exploited in the way practitioners do so, e.g., by means of plug-ins. These are obvious threats to the validity of research, as the effectiveness of proposed protections is not compared against the commercial state-of-the-art and real-world practices.
- **Limited exploration of obfuscation combinations:** Our analysis reveals that in the examined papers, protections are rarely combined; a significant majority investigates protections in isolation rather than layered or combined with other protections, even in the papers in A\*/A-rated venues. This limited exploration of multi-layered/combined SPs in research may not fully capture the complexity of protections deployed in real-world applications, which often utilize a combination of techniques.
- **Evaluations with no analysis method:** In the malware obfuscation category, too many papers (even in A\*/A venues) lack an evaluation with analysis methods from an attacker's toolbox, meaning they do not evaluate real-world potency and resilience.
- **Evaluations with just a single analysis method:** Still in the malware obfuscation category, but only in B, C, and lower-rated publication venues, even the papers that do evaluate the SPs with analysis methods still heavily lean towards using a single analysis method. This sharply contrasts with real-world scenarios, where attackers typically employ a multitude of analysis methods in various combinations to undo, bypass, and work-around protections. It also implies that either potency or resilience is evaluated, but not both.



- **Lack of specificity in experimental setups:** All too many papers, especially in lower-rated publication venues, do not specify which versions and configurations of protection tools, analysis tools, and build tools were used.

To overcome the latter challenges, we simply invite researchers to provide more details in the description of their experimental setups. Regarding the use of commercial protection tools, we invite the vendors to offer their tools at reduced prices to academics, with the permission to evaluate their tools and to publish their results. With regards to the combination of obfuscations, we invite all academic tool developers to open up their tools to enable others to build on them and experiment more freely with them, and we invite all researchers to exploit the already available flexibility that tools such as Tigress and OLLVM offer to compose protections.

As for the insufficient use of appropriate analysis methods, we do not think we can ask researchers to deploy more tools to evaluate their SP contributions. Instead, more work is needed to ease the reuse of analysis tools. We will come back to that in the next section, which analyzes which forms of measurements are reported in the papers, including measurements with analysis tools.

## 5 Measurements

This section analyzes how papers measure their contributions in the form of new protections and/or analysis/deobfuscation methods. We first focus on the evaluation representation, i.e., the software format on which measurements and other evaluations are performed as reported. Next, we analyze which aspects of strength and cost are measured and what measurements are used thereto.

### 5.1 Evaluation Representation

For all papers performing measurements on software, we recorded the **evaluation representation**, i.e., the program format from which the experimental evaluations in the paper start and from which measurements are performed.

► *In most papers, most experimental evaluations are performed on software in the format in which it is being distributed to customers, i.e., the format on which real-world attackers get their hands.*

The bottom bars in Figure 5 show the representations of samples on which the papers report performed measurements. Some papers include additional measurements on a second, higher-level representation to obtain ground truth information or to evaluate the applicability of protections. Furthermore, some papers, such as those presenting MBA-deobfuscation techniques [75, 115], evaluate their tools on MBA expressions in their source code format.

On top of the 80 goodware papers that do not report any experimental evaluation, 4.9%  $\hat{=}$  15/309 goodware papers fail to perform an experimental evaluation on software in the format on which attackers get their hands, as shown by the red underlined numbers in the bottom bar of Figure 5.

This poses no problem for papers focusing on the obfuscation of source code of natively compiled languages (e.g., [64, 175, 198]), for surveys, and for papers that evaluate decision tool support rather than protection or analysis strength (e.g., [142]). For other papers, it does pinpoint a problem.

► *We observed that for most papers that have an experimental evaluation but lack one on the format on which attackers get their hands, that lack coincides with a lack of layered protection.* This coincidence makes sense: If no additional protection is deployed to widen the semantic gap between a higher-level representation on which the evaluated protection is deployed and a lower-level representation on which attackers get their hands, i.e., if there is a one-to-one and hence reversible mapping from higher-level obfuscation code constructs to lower-level ones, strength measurements on the richer, more abstract format can serve as reasonable approximations of the protection's real strength against attackers that only get their hands on the lower-level

representation. In other words, it is fine for researchers to use ground-truth information available on the higher-level representations to estimate practical strength when attackers can easily reconstruct the used information. For example, the practical potency of anti-slicing obfuscations can be evaluated accurately on static slices in C source code [122, 123, 163] if no additional obfuscation has been applied that prevents the reconstruction of those slices from assembly code. Similarly, the practical potency of control flow or data flow obfuscations can be measured on Java source code instead of on bytecode [91, 155] if no anti-decompilation obfuscations have been applied that prevent reconstruction of the obfuscated flows from the bytecode.

However, we still do consider performing evaluations of protections in isolation a threat to their real-world relevance and validity because in practice only layered protections provide real protection [110], and in practically useful compositions one or more layers do exploit the semantic gap between higher-level and lower-level software formats, such as source code vs. native code, in order to hamper code understanding and reconstruction of the original code structure. Even if such additional obfuscations are deployed, it is often unclear to what extent metrics computed on higher-level representations are representative of practical protection strength. For example, in the mentioned example of anti-slicing obfuscations it is not clear whether a potency metric computed on slices that are statically and accurately computed on source code (i.e., using the ground truth information that defenders have access to) provides an upper bound or a lower bound on the practical potency in cases where additional obfuscations prevent the accurate static computation of slices by attackers starting from the binary code.

► *We hence consider protections getting evaluated in isolation a bigger threat to validity than evaluations happening on a higher-level representation than the one attackers can attack in practice.*

Unfortunately, that threat to validity is not limited to the relatively few papers that completely lack any evaluation on the appropriate format. Further analysis revealed that of all goodware papers performing at least some evaluation on samples written in natively compiled languages,  $32\% \hat{=} 64/(194+7+1)$  actually only measure overhead (compilation time, run time, memory consumption, and power consumption) and applicability (i.e., to what percentage of a program is some transformation applicable). They do not measure any potency, resilience, stealth, or other form of protection strength on actual binaries. Another  $8.9\% \hat{=} 18/(194+7+1)$  papers do measure strength-related features, but they do so on other representations.

► *We consider this lack of adequate strength measurements a major shortcoming and lack of maturity in obfuscation research for natively compiled goodware.*

## 5.2 Measurement Aspects and Categories

We identified ten *protection strength measurement categories* that are used to evaluate three strength aspects of SPs, namely potency,<sup>3</sup> resilience<sup>3</sup>, and stealth. We define the potency of an SP as the extent to which it can prevent, confuse, or hamper some analysis of the asset the SP protects. With this definition, SPs can have different potencies, each of which is specific to the considered analysis. Importantly, that can be a manual human analysis such as code comprehension, but also an automated analysis such as disassembling or malware detection. Our definition of resilience of an SP then is its capability to resist targeted attacks on the SP itself (instead of on the protected asset) to undo it or otherwise nullify or reduce its impact. Finally, our definition of stealth concerns

<sup>3</sup>The original definitions of potency and resilience by Collberg et al. [57, 58] unnecessarily aimed to discriminate between human and automated analysis, in an unsatisfactory manner. Collberg's revised definition of potency fixed this by unifying the two aspects under the umbrella of potency 2.0, which he then defined as a protection's effect of making at least one analysis harder to perform, and no analysis easier [133]. We find this revised definition insufficiently discriminative, however, as it does not discriminate between analyses of the protected assets and analyses of the protections. For this reason, we refine Collberg's potency 2.0, and complement it with a revised definition of resilience.

Table 5. Number of Papers that Report Measurements for Different Strength and Cost Criteria of SPs

	Deobf./Ana. Goodware	Deobf./Ana. Malware	Obfuscation Goodware	Obfuscation Malware	Total
Papers with protection implementation	87	137	248	47	495
<b>Aspect of Protection Measured</b>					
Costs	29	20	197	11	245
Potency	13	63	97	36	199
Resilience	43	43	40	3	122
Stealth	14	29	21	3	73
<b>Cost Measurement Categories</b>					
Static program size	18	17	111	8	149
Execution time	10	2	137	5	148
Compilation/protection time	1	2	17	3	22
Dynamic memory consumption	0	0	18	0	18
Dynamic power consumption	0	0	4	0	4
Other costs	2	2	4	0	8
<b>Strength Measurement Categories</b>					
<i>Papers with 0 strength measurements</i>	13	16	92	9	125
Other precision, recall, F-score, ...	40	50	31	4	118
Automated attack/analysis time	35	63	20	4	117
Malware detection precision	0	49	0	30	76
Code complexity	10	2	48	2	59
Deltas / similarity	7	13	16	3	39
Applicability (Code Coverage)	5	3	16	2	24
Human analysis (effort, success rate)	5	4	15	2	24
Opcodes distribution	2	4	16	0	21
Entropy	2	3	5	1	10

the ability of an SP to remain unidentified, unrelated to whether or not the SP helps in hiding the presence of the asset protected by the SP. Note that there hence exists no one-to-one mapping between the measurement categories and the three aspects of protection strength. For example, precision metrics can be used to evaluate stealth in obfuscation classification papers, to evaluate potency in malware detection papers, and to evaluate resilience in goodwill deobfuscation papers.

In addition to the strength measurement categories, we identified six *cost measurement categories* that are used to measure the cost of protections. Table 5 lists all measurement categories, with counts of their occurrences in different (non-exclusive) types of papers. A complete list of all categories, including descriptions, is provided in the supplemental material.

► *A first observation is that the costs of applying SPs are much more frequently measured than their strengths.* This is particularly the case in obfuscation goodwill papers, where cost measurements (79%  $\hat{=}$  197/248) are reported twice as often as potency measurements (39%  $\hat{=}$  97/248), the most popular measured aspect of strength. The vast majority of the cost measurements focus on static program size and execution time, which is not unexpected.

► *Still, the other 21% of the obfuscation goodwill papers fail to report the costs of the protections they discuss.* This lack of cost evaluation is unfortunately not limited to lower-rated publication venues: also in A\*/A obfuscation goodwill papers, we observed that (20%  $\hat{=}$  8/41) of the papers that do feature protection implementations lack cost measurements.

Of the SP strength measurements, stealth is clearly the least frequently evaluated (15%  $\hat{=}$  73/495).

In the deobfuscation/analysis types of papers, the balance between potency and resilience measurements matches the balance between goodware and malware papers: Analysis papers typically measure potency, while deobfuscation papers measure resilience.

► *In obfuscation papers, potency is measured much more often than resilience.* For obfuscation malware papers, this is to be expected, as malware obfuscation aims by and large at preventing detection, not at preventing deobfuscation. For goodware obfuscation papers, the fact that potency measurements (39%  $\hat{=}$  97/248) are more than twice as popular than resilience measurements (16%  $\hat{=}$  40/248) is in line with our earlier observations (see Sections 2.3 and 4.4) that few researchers study how attackers might adapt their strategies to the SPs they propose. Among researchers that publish in A\*/A venues, this appears not to be the case, however: 37%  $\hat{=}$  18/49 A\*/A of the goodware obfuscation papers present potency measurements, while only slightly less (33%  $\hat{=}$  16/49) present resilience measurements. Only 10%  $\hat{=}$  5/49 present both.

In defensive malware papers, analysis time (46%  $\hat{=}$  63/137) and malware detection precision (36%  $\hat{=}$  49/137) and other precision metrics (36%  $\hat{=}$  50/137) are by far the most popular measurement categories. Delta/similarity measurements (9.5%  $\hat{=}$  13/137) is the only remaining category that is somewhat popular in those papers. In offensive malware research, malware detection precision is also by far the most popular measurement category (64%  $\hat{=}$  30/47). The dominance of these measurement categories for malware research is not surprising.

In goodware papers, many more strength measurement categories have some popularity. In offensive goodware papers, precision (46%  $\hat{=}$  40/87) and automated attack/analysis time (40%  $\hat{=}$  35/87) are the dominant measurement categories. In defensive, obfuscation goodware papers, code complexity is the most popular measurement, being reported in 19%  $\hat{=}$  48/248 of the papers. The only other category measured in more than 10% of the obfuscation goodware papers is precision (13%  $\hat{=}$  31/248).

► *Perhaps the most striking result is that 25%  $\hat{=}$  125/495 of the papers with protection implementations report no strength measurements.* This is mostly due to 37%  $\hat{=}$  92/248 of the obfuscation goodware papers reporting no strength measurements on the obfuscations they present, an astonishingly high number. While there is a correlation between this number and the quality of the publication venue, this problem is certainly not limited to lower-rated venues: 22%  $\hat{=}$  9/41 of the A\*/A obfuscation goodware papers that feature implemented obfuscations report no strength measurements. This trend of lacking adequate strength measurements in obfuscation goodware papers also shows in the average number of such measurements per paper: 1.0 overall, and still only 1.4 in A\*/A papers.

We do not know for sure what the reason behind this lack of strength measurements is. We see two possible reasons. First, authors of goodware obfuscation might not consider such measurements relevant. We doubt that this is the case. Secondly, performing strength measurements might be considered too complex and/or too time-consuming to be worthwhile. We can think of several reasons why authors might think so. First, for many types of SPs, there is no consensus on which metrics to use. Theoretical complexity metrics from the field of software engineering often suffer from issues when used on obfuscated software, and practical metrics of the performance of concrete tools are most often ad hoc. Moreover, using real-world tools such as the IDA Pro disassembler (and its plug-in capabilities) or symbolic execution engines is rather difficult. Secondly, the relevant measurements can require experiments mimicking attack strategies involving multiple analyses. This is the case, e.g., for “binary” protections that completely prevent certain types of analysis tools (such as anti-debuggers, or code mobility preventing static disassembly). Measuring the impact of such protections requires comparing attack strategies with those tools to alternative attack strategies without those tools rather than evaluating the performance of individual analyses.

While tools/frameworks have been proposed in the past to ease the use of reverse engineering tools in research [4, 17, 154, 161] and to move into the direction of a commonly accepted set of criteria [9], our analysis of the literature in this section and in Section 4.4 on the use of tools reveals that further work is needed to convince authors to include more strength measurements and to facilitate the use of tools to obtain relevant measurements.

### 5.3 Identified Challenges

In our analysis of measurements, we identified one major challenge.

- **Focus on cost measurements, lack of strength measurement:** In the surveyed papers, the most often measured aspects of protections are costs. This is unsurprising, as cost measurements and their interpretation are rather straightforward. By contrast, the lack of established methods for evaluating the strength of protections is clearly evident in the limited number of strength measurements in the surveyed papers. This is particularly the case in obfuscation goodware papers, including those from top venues with A\*/A ranking.

Compared to the challenges related to samples and treatments, which might be solvable by a community consensus on shared sample sets and by more sharing of analysis tools, the strength measurement challenges are probably much more difficult to address as the weaknesses of the evaluations arise from both the choice of measurement types and tools used. Clearly, it is not useful to define a standardized methodology for all evaluations, as different protections have very different motivations and goals, and thus the choice of appropriate measurements can also differ significantly. Still, we strongly believe these challenges need to be addressed. The development of a flexible and easily re-usable protected software analysis toolbox, to which researchers can contribute their own measurement techniques and scripts, could help to improve the situation. A large community effort is needed to advance beyond the fragmental past efforts [4, 17, 154, 161].

## 6 Experiments with Human Subjects

In the large body of literature studied in this survey, few papers present experiments in which the performance of human subjects deploying SP or attacks on protected software is evaluated. Table 6 summarizes our findings on these papers. A more complete table can be found in the supplemental material. Notice that this list excludes papers that merely discuss how a human performs some analysis or uses some tool. In the papers listed in Table 6, the performance of the humans is analyzed and evaluated to gather knowledge about the strength of SPs.

The first observation is that few validations of SPs have been performed involving human subjects. Important to know, in [48] Ceccato et al. report on a superset of the experiments reported in [47]. Together with [125] and [137], these are the only experiments in which more than two protections are layered on top of each other. Moreover, only three experiments involved professional SP experts [47, 48, 91, 137], and only five papers have experiments lasting at least one working day [47, 48, 106, 193, 199]. While it is understandable that few research groups have the budget to hire professionals for multiple days, these results do indicate that few experiments have been performed that are representative of real-world MATE attacks. For all the other experiments, it is also understandable that they are performed on toy programs or small “complex” programs. This enables the student participants, which can in practice not be required or demanded to participate to longer running experiments, to finish their assignments in the experiments’ limited time frames.

► *It is an open question whether results from such short running experiments with non-expert subjects and toy program lacking SP layering can be extrapolated to real-world attack scenarios.*

Table 6. Papers Reporting Experiments with Human Subjects Performing MATE Protection and Attack Tasks

Paper	Year	Subjects	Samples	# Protections	Language	Format	Time
[199]	2021	5 ○	Toy	1,1	Native	Native	12h
[28]	2021	14 ●	Toy	Vanilla,2	Managed	Intermediate	?
[174]	2020	87 ○	Complex	Vanilla,1	Native	Source	2h
[91]	2020	22 ○ ● ○	Complex	Vanilla, ?	Managed	Source	1h
[48]	2019	6 ● 1 ○	*Mobile, Toy	9,8,7,3,2,1,1,1	Native	Native	30d
[29]	2019	14 ●	Toy	Vanilla, 2	Managed	Intermediate	?
[193]	2019	4 ●	Mobile	1	Managed	Intermediate	40h
[93]	2018	2 ○ 64 ○	Complex	Vanilla,1,1	Managed	Intermediate	1.5h
[106]	2018	2 ○ 13 ○	Complex, Toy	2	Native	Native	72h
[181]	2018	63 ●	Mobile	?	Managed	Intermediate	?
[117]	2017	10 ○ 10 ○	Complex	?	Script	Source	?
[47]	2017	6 ●	*Mobile	9,8,7	Native	Native	30d
[116]	2016	20 ●	Complex	?	Script	Source	?
[125]	2016	1 ●	Complex, Toy	3,2,2	Native	Native	?
[175]	2016	1 ○ 14 ○	Complex	Vanilla,1	Native	Source	3.5h
[202]	2014	12 ○	Complex	Vanilla,1,1,2	Managed	Source	1h
[44]	2014	22 ○ 52 ○	Complex	Vanilla,1,1	Managed	Intermediate	4h
[140]	2009	6 ●	Malware	1	Native	Native	?
[46]	2009	22 ○ 10 ○	Complex	Vanilla,1	Managed	Intermediate	4h
[45]	2008	8 ○	Complex	Vanilla,1	Managed	Intermediate	4h
[137]	2007	5 ●	*Complex	?+5	Native	Native	80m

“Subjects” indicates how many subjects participated of various levels of expertise: ○ bachelor and master students, ○ PhD and postgraduate students that are not experts in SP or reverse engineering, ○ students and amateurs with considerable experience in SP or reverse engineering, ● professional programmers, and ● professional security experts and pen testers. “Samples” indicates the handled type of samples. Asterisks mark samples that are real-world programs rather than just “Complex” programs somewhere between toy and real-world programs. “# Protections” indicates the number of protections (if any) composed in different samples. Commas separate different samples; “Vanilla” means unprotected. “Language” indicates the targeted type of programming language. “Format” indicates the format from which the software was reverse engineered. “Time” indicates how long the experiments lasted. Question marks indicate the information is not available.

## 7 Recommendations for Goodware Obfuscation Research

Previous sections identified a number of evaluation methodology challenges that the field of software obfuscation is facing. Most of those challenges concern goodware obfuscation/deobfuscation/analysis research. This section follows up on that by formulating a number of recommendations for improving the evaluations reported in future research papers.

Importantly, we explicitly do not propose moving in the direction of one universal evaluation methodology for SPs and countermeasures. In goodware SP, the different types of obfuscations are deployed to counter different attack strategies. Hence, the obfuscations should be evaluated differently: with different forms of measurements, and by evaluating their impact on different analysis tools from the attacker’s toolbox. In short, different evaluation methodologies can and should be used for different SPs. What all evaluations should have in common, such as including an evaluation of the potency and of the resilience, using concrete attack tools, on code obfuscated with state-of-the-art tools and layered protections, is precisely captured in our recommendations.

*Multiperspectivism.* When presenting a new obfuscation/analysis contribution that aims to counter some existing analysis/obfuscations, do not only evaluate how the contribution fares against those existing analyses/obfuscations as is. Also evaluate how it fares against adversaries that adapt their strategy. In other words, try to think as your adversary, at least considering what their immediate reactions might be. For example, for a new deobfuscation technique, propose some potential countermeasure obfuscations [128], and when evaluating novel obfuscations to defeat the function reconstruction heuristics of commercial disassemblers, do so using plug-ins that to some degree override the default heuristics with ones that aim at defeating your new obfuscations [171].

*Complete strength evaluation.* When presenting a new SP, evaluate its potency, i.e., its strength in terms of protecting the assets it is supposed to protect; its resilience, i.e., its strength for resisting attacks on the SP itself rather than on the protected asset; and, if relevant, its stealth, i.e., how easy it is to detect the presence and configuration of the SP in a program. Ideally, you evaluate these aspects with respect to multiple analysis techniques, including static and dynamic ones that are known, from the scientific literature and other sources, to be deployed by real-world MATE attackers. Pattern matching is a prime example of a popular analysis to be considered. Provide convincing arguments for your choice of analyses, and why you exclude other popular, capable analyses, if any. If you, for some reason, cannot include experimental results with samples for some relevant analysis, at least present a theoretical security assessment thereof.

*Layered SP deployment.* Deploy multiple, layered SPs on your samples, similar to how they are deployed in the real world. For a novel SP, evaluate its marginal value when combined with existing (popular) SPs, rather than its value when used in isolation. For managed and script languages, at least identifier renaming and string encryption should be included in the composed protections. Natively compiled software samples should, at the very least, be stripped.

*Concrete attacks evaluation.* To evaluate an SP, do not solely rely on complexity metrics computed on ground truth data. Instead, measure the SP's actual impact on analyses executed with concrete analysis tools. For example, evaluate the impact on their run times and on the precision of their results. If the tools are flexible and support plug-ins, as is the case for many disassemblers, consider using plug-ins (available online) rather than only the base tool. If possible, preference should be given to tools that attackers might use in the real world. For example, when evaluating how symbolic execution performs on protected code, tools that operate on binary code such as BINSEC/SE [65] or angr [154] are to be preferred over the use of KLEE on IR produced with LLVM. Alternatively, KLEE can, of course, also be deployed on IR lifted from binaries [77]. The survey by Schrittwieser et al. on obfuscations vs. analysis techniques [148] and the data presented in the supplemental work can help researchers to select the most relevant analyses for evaluating a new obfuscation's strengths, or to select the best obfuscations to stress-test a novel analysis technique.

*State-of-the-art SP tools.* The most advanced, commercial SP tools such as the offerings by Irdeto, Digital Ai, and GuardSquare are unavailable to most if not all researchers. So instead, evaluate analyses and deobfuscation techniques on samples generated with what comes closest to state-of-the-art SP tools. Several tools are available for free or affordable prizes, including Tigress, VMProtect, and Themida. Some older tools such as OLLVM and ProGuard cannot be considered today's state of the art. In addition, configure the tools properly such that your deployment is representative of real-world deployment. This can require quite some effort, e.g., to decide which SPs are combined and layered on which parts of the sample programs.

*Setup specificity.* For all tools used to build, protect, and analyze samples, specify the used versions and configurations. If you use a default configuration, specify what that entails. This is


particularly important for tools of which the default settings evolve over time, such as Tigress where the default configurations of individual SPs evolved with different releases.

*Tool availability.* Make your research tools available as artifacts for reuse and reproducibility.

*Sample complexity.* Include in your data set a sufficient number of sample programs of sufficient complexity to be representative of real-world use cases. Which concrete complexity metrics are relevant and which levels of complexity should be included depends entirely on the claims for which the evaluation is supposed to provide evidence. For example, when claiming that some deobfuscation technique can fail even on the simplest of samples, experiments with simple samples suffice [153]. By contrast, if a technique requires identifying the relevant fragments in program traces, large programs that generate long traces should be included [34]. In addition, the samples build process should be representative of how real-world software is being built. For example, do not include native binaries compiled at `-O0`, i.e., without any compiler optimizations enabled.

*Sample availability.* Make the samples in your dataset available as artifacts for reuse and reproduction by others. As for commercial programs, make sure you mention the exact versions.

*Sample diversification.* When performing an evaluation on protected samples that you generate yourself with some configurable obfuscation and/or with an obfuscation of which the behavior is randomized, include multiple obfuscated versions for varying configurations and random seeds and deploy accepted statistical techniques to aggregate the obtained measurements. For measuring the performance overhead, include multiple obfuscated samples where the protection has been deployed on a range of program points with varying degrees of execution frequencies.

These recommendations are concrete and realistic, as evidenced by the fact that there exist quite a few papers that already implemented them in the past. To help readers find good examples, Table 7 lists the extent to which A\* goodwill papers in our survey implement our recommendations. The sample diversification column lists multiple numbers for papers with multiple, distinct subsets of samples with varying levels of diversification. Each reported number is the rounded ratio between the total sample set size of such a subset and its original sample set size. In the state-of-the-art SP tools column, we marked Javascript minifiers such as Google's Closure compiler that do not deploy more advanced obfuscations with . Apart from that, it should be clear that our judgments are subjective: they are our interpretation of our recommendations and of the papers. Still, we hope this table can provide guidance for future research in software obfuscation.

For inspiration as to how to instantiate some of these generic recommendations for concrete SPs or analyses, we refer to the report of the Dagstuhl seminar that motivated us for this survey [68]. The report's Section 4 lists concrete recommendations for research into anti-disassembly SPs and into trace-based analyses. In addition, the supplemental material contains concrete recommendations for the usage of some of the most popular research tools. In addition, the supplemental material presents some potential sources of inspiration for selecting relevant combinations of layered protections.

## 8 Related Work

While ours is the largest survey in the field of SP to date, it is not the first. We identified 25 in-scope surveys and several non-survey related works, which we classify in this section. A complete list of these papers including short descriptions is available in the supplemental material.

We identified several literature surveys [10, 73, 97, 98, 108, 139, 148, 187, 195], which — similar to our work — provide a broad overview of SP research from different perspectives. Another category of publications similar to our work are taxonomies, introductions, tutorials and other theoretical publications [5, 18, 30, 56, 94, 95, 118, 126, 150, 151, 188, 197]. These papers do meta descriptions such as categorization or classification of SP techniques, attack scenarios, methods for measuring



Table 7. Goodware Papers Published in A\* Venues, Scored for their Implementation of the Recommendations from Section 7

Paper	Obfuscation	Deobfuscation	Analysis	Multiperspectivism	Complete strength eval.	Layered SP deployment	Concrete attacks eval.	State-of-the-art SP tools	Setup specificity	Tool availability	Sample Complexity	Sample availability	Sample diversification	Year	Venue	Authors
[147]	✓	✓	●	●	●	●	●	●	●	●	●	●	175	2022	Usenix Security	Schloegel et al.
[153]		✓	○	n/a	●	n/a	●	●	●	○	●	●	3	2022	NDSS	Shijia et al.
[128]	✓	✓	●	●	●	●	●	○	●	●	●	●	1; 2; 3	2021	ACM CCS	Menguy et al.
[201]	✓		○	●	●	●	○	○	●	●	●	●	10	2020	IEEE/ACM ACE	Zhou et al.
[179]	✓		○	●	○	○	○	○	●	○	○	○	2; 4	2019	IEEE INFOCOM	Wang et al.
[69]			✓	○	n/a	●	n/a	●	●	●	●	●	211	2019	IEEE S&P	Ding et al.
[156]			✓	●	n/a	●	n/a	○	n/a	●	●	○	1; 12	2019	WWW Conf.	Skolka et al.
[176]	✓			●	●	●	○	n/a	○	○	○	○	n/a	2018	ACM/IEEE ICSE	Wang et al.
[178]	✓			○	●	●	○	○	○	○	○	○	2	2018	ACM/IEEE ICSE	Wang et al.
[76]	✓			○	●	○	○	○	○	○	○	○	6	2018	IEEE ToC	Fyrbiak et al.
[34]		✓		○	n/a	●	n/a	●	○	●	○	●	1; 2	2017	Usenix Security	Blazytko et al.
[117]	✓			○	●	○ <sup>4</sup>	n/a	○	○	○	○	○	3	2017	ACM/IEEE ICSE	Liu et al.
[119]			✓	○	n/a	●	n/a	○	○	○	○	○	1; 5; 6	2017	ACM FSE	Luo et al.
[172]		✓		○	n/a	○	n/a	○	n/a	●	●	○	1	2017	ACM FSE	Vasilescu et al.
[16]	✓			○	●	●	○	●	●	●	○	●	6	2017	Usenix Security	Banescu et al.
[12]		✓	✓	○	n/a	○	n/a	○	○	●	●	○	1	2016	ACM CCS	Backes et al.
[184]	✓			○	●	○	○	○	n/a	○	○	○	201	2010	ACM CCS	Wu et al.
[138]	✓			○	●	○	○	○	●	○	○	○	2	2007	Usenix Security	Popov et al.
[105]			✓	○	n/a	○	n/a	○	●	○	○	○	1	2004	Usenix Security	Kruegel et al.
[112]	✓			○	●	○	○	○	○	○	○	○	6	2003	ACM CCS	Linn and Debray
[58]	✓			○	●	○	○	○	n/a	n/a	○	n/a	n/a	1998	ACM POPL	Collberg et al.

● means well done, ○ not so; n/a means the recommendation is not applicable.

the strength of SPs. Some papers survey available tools and techniques for SP. Besides some early works, which look at the SP domain in general [13, 56] there exist multiple publications which take a more narrow focus: Different languages, e.g., Java [43], analysis avoidance [39, 135, 182], malware [113, 144, 191], specific SPs, e.g., control flow obfuscations [124], call-flow [197], instruction substitution [197], self modifying code [126], and indistinguishability obfuscation [20, 96]. Three publications survey the opposing side of SP – analysis and deobfuscation of SP [63, 74, 103]. Several papers assess uses of SPs in practice for Android apps [26, 70, 82, 92, 181], iOS apps [176], or malware [38]. Collberg and Proebsting published a large-scale study [55] on reproducibility and repeatability of experiments in computer science. For only 37% of the surveyed publications code was made available by the authors, which is in line with our findings.

### 9 Conclusions

This survey on software obfuscation is based on the largest collection of papers ever studied in the SP domain. While the mix of protection targets is similar to what we observed in other surveys, in

terms of types of programming languages targeted, our distinct focus on the measurements performed in the surveyed papers is novel. In the aftermath of the 2019 Dagstuhl Seminar on Software Protection Decision Support and Evaluation Methodologies, where participants expressed subjective worries about evaluation methodologies in SP research [68], we systematically searched for evidence of these concerns and indeed found a number of issues.

We see a concerning focus on cost measurements while the strength of SPs is way less often measured. Evaluation sample sets suffer from major shortcomings regarding availability, diversification, and complexity. Furthermore, we identified a troubling number of papers using no or just a single analysis technique for evaluation, and we observed a limited exploration of obfuscation combinations. Next, an identified gap between analysis methods used in analysis papers and evaluations of newly proposed protections raises the concern whether appropriate analysis methods are used for the evaluation of newly proposed protections. We also identified a gap between tools used in published research and the commercial state of the art. This casts a shadow on the real-world relevance of the published research, and confirms the continuing reliance on security-through-obscurity in this domain. Finally, the reproducibility of research results is low as too many papers do not specify versions and configurations of used tools. Interestingly, most of these issues are prevalent even in A and A\* publications, although there they tend to occur less frequently.

In summary, our work serves as a robust confirmation of the worries expressed in Dagstuhl. The prevalence of these issues across different tiers of publications underscores the urgent need for a broad reconsideration of evaluation methodologies in the SP research field. To that end, we formulated concrete recommendations, with a focus on goodware SP research. Beyond those recommendations that individual research teams can implement, we strongly advocate for the development of a community consensus on shared sample sets, akin to practices in other research domains. In particular the Trust Hub set of labeled hardware obfuscation benchmarks [8] can serve as an example. Furthermore, the creation of a flexible and easily re-usable protected software analysis toolbox could benefit both the reproducibility and comparability of results across studies, thereby driving the field forward and establishing a solid foundation for future explorations.

## Acknowledgments

For the purpose of open access, the authors have applied a CC BY public copyright license to any Author Accepted Manuscript version arising from this submission.

The authors would like to acknowledge the help of Armin Huremagic.

## References

- [1] Moataz AbdelKhalek and Ahmed Shosha. 2017. JSDES: An automated de-obfuscation system for malicious JavaScript. In *ARES*.
- [2] Bert Abrath, Bart Coppens, Jens Van den Broeck, Brecht Wyseur, Alessandro Cabutto, Paolo Falcarin, and Bjorn De Sutter. 2020. Code renewability for native software protection. *ACM Trans. Priv. Secur.* 23, 4 (2020).
- [3] Bert Abrath, Bart Coppens, Stijn Volckaert, Joris Wijnant, and Bjorn De Sutter. 2016. Tightly-coupled self-debugging software protection. In *ACM SSPREW*. 7:1–7:10.
- [4] Deepak Adhikari, J. Todd McDonald, Todd R. Andel, and Joseph D. Richardson. 2022. Argon: A toolbase for evaluating software protection techniques against symbolic execution attacks. In *SoutheastCon*. 743–750.
- [5] Mohsen Ahmadvand, Alexander Pretschner, and Florian Kelbert. 2019. A taxonomy of software integrity protection techniques. In *ADCOM*. Vol. 112. 413–486.
- [6] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting millions of Android apps for the research community. In *MSR (MSR'16)*. ACM, New York, NY, USA, 468–471. DOI: <https://doi.org/10.1145/2901739.2903508>
- [7] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. 2020. BinRec: Dynamic binary lifting and recompilation. In *EuroSys*.

- [8] Sarah Amir, Bicky Shakya, Xiaolin Xu, Yier Jin, Swarup Bhunia, Mark Tehranipoor, and Domenic Forte. 2018. Development and evaluation of hardware obfuscation benchmarks. *J. Hardw. Syst. Secur.* 2, 2 (2018), 142–161.
- [9] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. 2007. Program obfuscation: A quantitative approach. In *ACM QoP*. 15–20.
- [10] Claudio Agostino Ardagna, Qing Wu, Xueling Zhu, and Bo Liu. 2021. A survey of Android malware static detection technology based on machine learning. *Mobile Information Systems* (2021).
- [11] Eran Avidan and Dror G. Feitelson. 2015. From obfuscation to comprehension. In *IEEE ICPC*. 178–181.
- [12] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable third-party library detection in Android and its security applications. In *ACM CCS*. 356–367.
- [13] Arini Balakrishnan and Chloe Schulze. 2005. Code Obfuscation Literature Survey. CS701 Construction of Compilers.
- [14] Sebastian Banescu. 2016. *GitHub — A Set of Programs used for Benchmarking the Strength of Obfuscation*. <https://github.com/tum-i4/obfuscation-benchmarks>
- [15] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *ACM ACSAC*. 189–200.
- [16] Sebastian Banescu, Christian Collberg, and Alexander Pretschner. 2017. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *USENIX Security*. 661–678.
- [17] Sebastian Banescu, Martin Ochoa, and Alexander Pretschner. 2015. A framework for measuring software obfuscation resilience against automated attacks. In *IEEE/ACM SPRO*. 45–51.
- [18] Sebastian Banescu and Alexander Pretschner. 2017. A tutorial on software obfuscation. In *Advances in Computers*. Vol. 108. 283–353.
- [19] Sebastian Banescu, Samuel Valenzuela, Marius Guggenmos, Mohsen Ahmadvand, and Alexander Pretschner. 2021. Dynamic taint analysis versus obfuscated self-checking. In *ACM ACSAC*. 182–193.
- [20] Boaz Barak. 2016. Hopes, fears, and software obfuscation. *Commun. ACM* 59, 3 (2016), 88–96.
- [21] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-bounded DSE: Targeting infeasibility questions on obfuscated codes. In *IEEE S&P*. 633–651.
- [22] Cataldo Basile, Daniele Canavese, Leonardo Regano, Paolo Falcarin, and Bjorn De Sutter. 2019. A meta-model for software protections and reverse engineering attacks. *J. Sys. and Softw.* 150 (2019), 3–21.
- [23] Richard Baumann, Mykolai Protsenko, and Tilo Müller. 2017. Anti-ProGuard: Towards automated deobfuscation of Android apps. In *SHCIS*. 7–12.
- [24] Philippe Beaucamps and Éric Filiol. 2007. On the possibility of practically obfuscating programs — Towards a unified perspective of code protection. *J. in Comput. Virol.* 3, 1 (2007), 3–21.
- [25] Mihir Bellare, Iqbal Stepanovs, and Brent Waters. 2016. New negative results on differing-inputs obfuscation. In *EUROCRYPT*. 792–821.
- [26] Stefano Berlato and Mariano Ceccato. 2020. A large-scale study on the adoption of anti-debugging and anti-tampering protections in Android apps. *J. Inf. Sec. and App.* 52 (2020), 102463.
- [27] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical deobfuscation of Android applications. In *ACM CCS*. 343–355.
- [28] Mohammed H. Bin Shamlan, Alawi S. Alaidaroos, Mansoor H. Bin Merdhah, Mohammed A. Bamatraf, and Adnan A. Zain. 2021. Experimental evaluation of the obfuscation techniques against reverse engineering. In *ICACIn*. 383–390.
- [29] Mohammed H. Bin Shamlan, Mohammed A. Bamatraf, and Adnan A. Zain. 2019. The impact of control flow obfuscation technique on software protection against human attacks. In *ICOICE*. 1–5.
- [30] Fabrizio Biondi, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, and Jean Quilbeuf. 2018. Tutorial: An overview of malware detection and evasion techniques. In *ISoLA*. 565–586.
- [31] Sandrine Blazy and Rémi Hutin. 2019. Formal verification of a program obfuscation based on mixed Boolean-arithmetic expressions. In *ACM CPP*. 196–208.
- [32] Sandrine Blazy and Stéphanie Riaud. 2014. Measuring the robustness of source program obfuscation: Studying the impact of compiler optimizations on the obfuscation of C programs. In *ACM CODASPY*. 123–126.
- [33] Sandrine Blazy and Alix Trieu. 2016. Formal verification of control-flow graph flattening. In *ACM CPP*. 176–187.
- [34] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the semantics of obfuscated code. In *USENIX Security*. 643–659.
- [35] Jean-Marie Borello and Ludovic Mé. 2008. Code obfuscation techniques for metamorphic viruses. *J. in Comput. Virol.* 4 (2008), 211–220.
- [36] P. D. Borisov and Yu. V. Kosolapov. 2020. On the automatic analysis of the practical resistance of obfuscating transformations. *Autom. Control Comput. Sci.* 54, 7 (2020), 619–629.
- [37] Pietro Borrello, Emilio Coppa, and Daniele Cono D’Elia. 2021. Hiding in the particles: When return-oriented programming meets program obfuscation. In *IEEE/IFIP DSN*. 555–568.

- [38] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. 2012. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-VM technologies. *Black Hat* (2012).
- [39] Murray Brand. 2010. *Analysis Avoidance Techniques of Malicious Software*. School of Computer and Security Science.
- [40] Pierrick Brunet, Béatrice Creusillet, Adrien Guinet, and Juan Manuel Martinez. 2019. Epona and the obfuscation paradox: Transparent for users and developers, a pain for reversers. In *ACM SPRO*. 41–52.
- [41] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. 2012. Aligot: Cryptographic function identification in obfuscated binary programs. In *ACM CCS*. 169–182.
- [42] Gerardo Canfora, Andrea Di Sorbo, Francesco Mercaldo, and Corrado Aaron Visaggio. 2015. Obfuscation techniques against signature-based detection: A case study. In *MST*. 21–26.
- [43] Mariano Ceccato, Andrea Capiluppi, Paolo Falcarin, and Cornelia Boldyreff. 2015. A large study on the effect of code obfuscation on the quality of Java code. *Empir. Softw. Eng.* 20, 6 (2015), 1486–1524.
- [44] Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. 2014. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empir. Softw. Eng.* 19, 4 (2014), 1040–1074.
- [45] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. 2008. Towards experimental evaluation of code obfuscation techniques. In *ACM QoP*. 39–46.
- [46] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. 2009. The effectiveness of source code obfuscation: An experimental assessment. In *IEEE ICPC*. 178–187.
- [47] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Bart Coppens, Bjorn De Sutter, Paolo Falcarin, and Marco Torchiano. 2017. How professional hackers understand protected code while performing attack tasks. In *IEEE ICPC*. 154–164.
- [48] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. 2019. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *Empir. Softw. Eng.* 24, 1 (2019), 240–286.
- [49] Aziem Chawdhary, Ranjeet Singh, and Andy King. 2017. Partial evaluation of string obfuscations for Java malware detection. *Formal Aspects of Computing* 29, 1 (2017), 33–55.
- [50] Yun-Chung Chen, Hong-Yen Chen, Takeshi Takahashi, Bo Sun, and Tsung-Nan Lin. 2021. Impact of code deobfuscation and feature interaction in Android malware detection. *IEEE Access* 9 (2021), 123208–123219.
- [51] Binlin Cheng, Jiang Ming, Erika A. Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. 2021. Obfuscation-resilient executable payload extraction from packed malware. In *USENIX Security*. 3451–3468.
- [52] Xiaoyang Cheng, Yan Lin, Debin Gao, and Chunfu Jia. 2019. DynOpVm: VM-based software obfuscation with dynamic opcode mapping. In *ACNS*. 155–174.
- [53] Seongje Cho, Hyeoung Chang, and Yookun Cho. 2008. Implementation of an obfuscation tool for C/C++ source code protection on the XScale architecture. In *IFIP SEUS*. 406–416.
- [54] Mihai Christodorescu, Somesh Jha, Johannes Kinder, Stefan Katzenbeisser, and Helmut Veith. 2007. Software transformations to improve malware detection. *J. in Comput. Virol.* 3 (2007), 253–265.
- [55] Christian Collberg and Todd A. Proebsting. 2016. Repeatability in computer systems research. *Comm. ACM* 59, 3 (2016), 62–69.
- [56] Christian Collberg and Clark Thomborson. 2002. Watermarking, tamper-proofing, and obfuscation – tools for software protection. *IEEE Trans. Softw. Eng.* 28, 8 (2002), 735–746.
- [57] Christian Collberg, C. Thomborson, and Douglas Low. 1997. *A Taxonomy of Obfuscating Transformations*. Technical Report 148. University of Auckland.
- [58] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *ACM POPL*. 184–196.
- [59] Computing Research and Education Association of Australasia, CORE Inc. [n.d.]. *CORE*. <https://www.core.edu.au>
- [60] Kevin Coogan and Saumya Debray. 2011. Equational reasoning on x86 assembly code. In *IEEE SCAM*. 75–84.
- [61] Mila Dalla Preda and Roberto Giacobazzi. 2005. Control code obfuscation by abstract interpretation. In *IEEE SEFM*. 301–310.
- [62] Mila Dalla Preda and Roberto Giacobazzi. 2005. Semantic-based code obfuscation by abstract interpretation. In *ICALP*. 1325–1336.
- [63] Mila Dalla Preda and Federico Maggi. 2016. Testing Android malware detectors against code obfuscation: A systematization of knowledge and unified methodology. *J. Comput. Virol. and Hack. Tech.* 13, 3 (2016), 209–232.
- [64] Siddhartha Datta. 2021. DeepObfusCode: Source code obfuscation through sequence-to-sequence networks. In *Intelligent Computing*. 637–647.
- [65] Robin David, Sébastien Bardin, Thanh Dinh Ta, Josselin Feist, Laurent Mounier, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *IEEE SANER*.
- [66] Robin David, Luigi Coniglio, and Mariano Ceccato. 2020. QSynth - a program synthesis based approach for binary code deobfuscation. In *BAR*.

- [67] Robbe De Ghein, Bert Abrath, Bjorn De Sutter, and Bart Coppens. 2022. ApkDiff: Matching Android app versions based on class structure. In *ACM CheckMATE*. 1–12.
- [68] Bjorn De Sutter, Christian Collberg, Mila Dalla Preda, and Brecht Wyseur. 2019. Software protection decision support and evaluation methodologies (Seminar 19331). *Dagstuhl Reports* 9, 8 (2019), 1–25. <https://drops.dagstuhl.de/opus/volltexte/2019/11682>
- [69] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *IEEE S&P*. 472–489.
- [70] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. 2018. Understanding Android obfuscation techniques: A large-scale investigation in the wild. In *SecureComm*. 172–192.
- [71] Weiyu Dong, Jian Lin, Rui Chang, and Ruimin Wang. 2022. CaDeCFF: Compiler-agnostic deobfuscator of control flow flattening. In *Internetware*. 282–291.
- [72] Dmitriy Dunaev and Laszlo Lengyel. 2012. Complexity of a special deobfuscation problem. In *ECBS*. 1–4.
- [73] Shouki A. Ebad, Abdulbasit A. Darem, and Jemal H. Abawajy. 2021. Measuring software obfuscation quality—a systematic literature review. *IEEE Access* 9 (2021), 99024–99038.
- [74] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM CSUR* 44, 2 (2012).
- [75] Ninon Eyrolles, Louis Goubin, and Marion Videau. 2016. Defeating MBA-based obfuscation. In *ACM SPRO*. 27–38.
- [76] Marc Fyrbiak, Simon Rokicki, Nicolai Bissantz, Russell Tessier, and Christof Paar. 2018. Hybrid obfuscation to protect against disclosure attacks on embedded microprocessors. *IEEE Trans. Comp.* 67, 3 (2018), 307–321.
- [77] Peter Garba and Matteo Favaro. 2019. SATURN - software deobfuscation framework based on LLVM. In *ACM SPRO*. 27–38.
- [78] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. 2005. Control flow based obfuscation. In *ACM DRM*. 83–92.
- [79] Franz-Xaver Geiger and Ivano Malavolta. 2018. Datasets of Android applications: A literature review. *arXiv preprint arXiv:1809.10069* (2018).
- [80] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. 2017. CodeMatch: Obfuscation won't conceal your repackaged app. In *ESEC/FSE*. 638–648.
- [81] Louis Goubin, Pascal Paillier, Matthieu Rivain, and Junwei Wang. 2020. How to reveal the secrets of an obscure white-box implementation. *J. Crypt. Eng.* 10, 1 (2020), 49–66.
- [82] Pierre Graux, Jean-Francois Lalande, and Valérie Viet Triem Tong. 2019. Obfuscated Android application development. In *CCEC*.
- [83] Felix Gröbert, Carsten Willems, and Thorsten Holz. 2011. Automated identification of cryptographic primitives in binary programs. In *RAID*. 41–60.
- [84] GuardSquare. 2024. *Dexguard*. <https://www.guardsquare.com/dexguard>
- [85] S. Guelton, A. Guinet, P. Brunet, J. M. Martinez, F. Dagnat, and N. Sziifierski. 2018. Combining obfuscation and optimizations in the real world. In *IEEE SCAM*. 24–33.
- [86] Yoann Guillot and Alexandre Gazet. 2009. Semi-automatic binary protection tampering. *J. in Comput. Virol.* 5, 2 (2009), 119–149.
- [87] Yoann Guillot and Alexandre Gazet. 2010. Automatic binary deobfuscation. *J. in Comput. Virol.* 6, 3 (2010), 261–276.
- [88] Runsheng Guo, Qichao Liu, Man Zhang, Ning Hu, and Hui Lu. 2022. A survey of obfuscation and deobfuscation techniques in Android code protection. In *IEEE DSC*. 40–47.
- [89] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE WWC*. IEEE, 3–14.
- [90] Kimaya Hajarnis, Jash Dalal, Rupali Bawale, Jibi Abraham, and Ashwini Matange. 2021. A comprehensive solution for obfuscation detection and removal based on comparative analysis of deobfuscation tools. In *SMART GENCON*. 1–7.
- [91] Salsabil Hamadache and Malte Elson. 2020. Creative manual code obfuscation as a countermeasure against software reverse engineering. In *AISC*. 3–8.
- [92] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products. In *ICSE*. 421–431.
- [93] Norman Hänsch, Andrea Schankin, Mykolai Protsenko, Felix Freiling, and Zinaida Benenson. 2018. Programming experience might not help in comprehending obfuscated source code efficiently. In *SOUPS*. 341–356.
- [94] Irfan Ul Haq and Juan Caballero. 2021. A survey of binary code similarity. *ACM CSUR* 54, 3 (2021). Issue June 2021.
- [95] Muhammad Hataba and Ahmed El-Mahdy. 2012. Cloud protection by obfuscation: Techniques and metrics. In *3PG-CIC*. 369–372.
- [96] Máté Horváth and Levente Buttyán. 2020. *Cryptographic Obfuscation: A Survey*. Springer.

- [97] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. 2016. A survey on aims and environments of diversification and obfuscation in software security. In *CompSysTech*. 113–120.
- [98] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. 2018. Diversification and obfuscation techniques for software security: A systematic literature review. *Inf. Softw. Technol.* 104 (2018).
- [99] Irdeto. 2019. *Cloakware by Irdeto*. <https://irdeto.com/cloakware-by-irdeto>
- [100] Zeliang Kan, Haoyu Wang, Lei Wu, Yao Guo, and Guoai Xu. 2019. Deobfuscating Android native binary code. In *ICSE Companion*. 322–323.
- [101] Seoyeon Kang, Jeongwoo Kim, Eun-Sun Cho, and Seokwoo Choi. 2022. Program synthesis-based simplification of MBA obfuscated malware with restart strategies. In *ACM CheckMATE*. 13–18.
- [102] Keshav Kaushik, Harshpreet Singh Sandhu, Neelesh Kumar Gupta, Naman Sharma, and Rohit Tanwar. 2022. A systematic approach for evading antiviruses using malware obfuscation. In *ETBS*. 29–37.
- [103] Patrick Kochberger, Sebastian Schrittwieser, Stefan Schweighofer, Peter Kieseberg, and Edgar Weippl. 2021. SoK: Automatic deobfuscation of virtualization-protected applications. In *ARES*.
- [104] Yury Kosolapov and Petr Borisov. 2020. Similarity features for the evaluation of obfuscation effectiveness. In *DASA*. 898–902.
- [105] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *USENIX Security*. 255–270.
- [106] Kaiyuan Kuang, Zhanyong Tang, Xiaoqing Gong, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. 2018. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. *Comput. & Secur.* 74 (2018), 202–220.
- [107] Akshay Kumar and Seema Sharma. 2019. Design and implementation of obfuscating tool for software code protection. In *LNME*. 665–676.
- [108] Renuka Kumar and Anjana Mariam Kurian. 2018. A systematic study on static control flow obfuscation techniques in Java. *arXiv preprint arXiv:1809.11037* (2018).
- [109] Renuka Kumar and Anand Raj Essar Vaishakh. 2016. Detection of obfuscation in Java malware. *Procedia Computer Science* 78 (2016), 521–529.
- [110] Clifford Liem, Yuan Xiang Gu, and Harold Johnson. 2008. A compiler-based infrastructure for software-protection. In *ACM PLAS*. 33–44.
- [111] Kyeonghwan Lim, Jaemin Jeong, Seong-je Cho, Jongmoo Choi, Minkyu Park, Sangchul Han, and Seongtae Jhang. 2017. An anti-reverse engineering technique using native code and obfuscator-LLVM for Android applications. In *RACS*. 217–221.
- [112] Cullen Linn and Saumya Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *ACM CCS*. 290–299.
- [113] Cătălin Valeriu Liță, Doina Cosovan, and Dragoș Gavriluț. 2018. Anti-emulation trends in modern packers: A survey on the evolution of anti-emulation techniques in UPa packers. *J. Comput. Virol. and Hack. Tech.* 14, 2 (2018), 107–126.
- [114] Binbin Liu, Weijie Feng, Qilong Zheng, Jing Li, and Dongpeng Xu. 2021. Software obfuscation with non-linear mixed Boolean-arithmetic expressions. In *ICISC*. 276–292.
- [115] Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. 2021. MBA-Blast: Unveiling and simplifying mixed Boolean-arithmetic obfuscation. In *USENIX Security*. 1701–1718.
- [116] Han Liu. 2016. Towards better program obfuscation: Optimization via language models. In *ICSE Companion*. 680–682.
- [117] H. Liu, C. Sun, Z. Su, Y. Jiang, M. Gu, and J. Sun. 2017. Stochastic optimization of program obfuscation. In *ICSE*. 221–231.
- [118] T. Long, L. Liu, Y. Yu, and Z. Wan. 2010. Assure high quality code using refactoring and obfuscation techniques. In *FCST*. 246–252.
- [119] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Trans. Softw. Eng.* 43, 12 (2017), 1157–1177.
- [120] Benjamin Lynn, Manoj Prabhakaran, and Amit Sahai. 2004. Positive results and techniques for obfuscation. In *EUROCRYPT*. 20–39.
- [121] Matias Madou, Ludo Van Put, and Koen De Bosschere. 2006. LOCO: An interactive code (De)Obfuscation tool. In *ACM PEPM*. 140–144.
- [122] Anirban Majumdar, Stephen Drape, and Clark Thomborson. 2007. Metrics-based evaluation of slicing obfuscations. In *IAS*. 472–477.
- [123] Anirban Majumdar, Stephen Drape, and Clark Thomborson. 2007. Slicing obfuscations: Design, correctness, and evaluation. In *ACM DRM*. 70–81.

- [124] Anirban Majumdar, Clark Thomborson, and Stephen Drape. 2006. A survey of control-flow obfuscations. In *ICISS*. 353–356.
- [125] Ramya Manikyam, J. Todd McDonald, William R. Mahoney, Todd R. Aniel, and Samuel H. Russ. 2016. Comparing the effectiveness of commercial obfuscators against MATE attacks. In *ACM SSPREW*.
- [126] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. 2011. A taxonomy of self-modifying code for obfuscation. *Comput. Secur.* 30, 8 (2011), 679–691.
- [127] J. Todd McDonald, Ramya Manikyam, Sébastien Bardin, Richard Bonichon, and Todd R. Aniel. 2021. Program protection through software-based hardware abstraction. In *SECRYPT*. 247–258.
- [128] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Cauim de Souza Lima. 2021. Search-based local black-box deobfuscation: Understand, improve and mitigate. In *ACM CCS*. 2513–2525.
- [129] Parkour Mila. 2024. *Contagio*. <https://contagiodump.blogspot.com>
- [130] Jiang Ming, Fangfang Zhang, Dinghao Wu, Peng Liu, and Sencun Zhu. 2016. Deviation-based obfuscation-resilient program equivalence checking with application to software plagiarism detection. *IEEE Trans. Reliability* 65, 4 (2016), 1647–1664.
- [131] Alireza Mohammadinodooshan, Ulf Kargén, and Nahid Shahmehri. 2019. Robust detection of obfuscated strings in Android apps. In *ACM AISec*. 25–35.
- [132] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically detecting JavaScript obfuscation and minification techniques in the wild. In *IEEE/IFIP DSN*. 569–580.
- [133] Jasvir Nagra and Christian Collberg. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional.
- [134] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. How to kill symbolic deobfuscation for free (or: Unleashing the potential of path-oriented protections). In *ACM ACSAC*. 177–189.
- [135] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. Obfuscation: Where are we in anti-DSE protections? (A First Attempt). In *ACM SSPREW*.
- [136] Colby B. Parker., J. Todd McDonald., and Dimitrios Damopoulos. 2021. Machine learning classification of obfuscation using image visualization. In *SECRYPT*. 854–859.
- [137] Ugo Piazzalunga, Paolo Salvaneschi, Francesco Balducci, Pablo Jacomuzzi, and Cristiano Moroncelli. 2007. Security strength measurement for dongle-protected software. *IEEE Security & Privacy* 5, 6 (2007), 32–40.
- [138] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. 2007. Binary obfuscation using signals. In *USENIX Security*. 275–290.
- [139] Junyang Qiu, Jun Zhang, Wei Luo, Lei Pan, Surya Nepal, and Yang Xiang. 2020. A survey of Android malware detection with deep neural models. *ACM CSUR* 53, 6 (2020).
- [140] Daniel A. Quist and Lorie M. Liebrock. 2009. Visualizing compiled executables for malware analysis. In *IEEE VizSec*. 27–32.
- [141] L. Regano, D. Canavese, C. Basile, and A. Lioy. 2017. Towards optimally hiding protected assets in software applications. In *QRS*. 374–385.
- [142] Leonardo Regano, Daniele Canavese, Cataldo Basile, Alessio Viticchié, and Antonio Lioy. 2016. Towards automatic risk analysis and mitigation of software applications. In *WISTP*. 120–135.
- [143] Benjamin Reichenwallner and Peter Meerwald-Stadler. 2022. Efficient deobfuscation of linear mixed Boolean-arithmetic expressions. In *ACM CheckMATE*. 19–28.
- [144] Kevin A. Roundy and Barton P. Miller. 2013. Binary-code obfuscations in prevalent packer tools. *ACM CSUR* 46, 1 (2013).
- [145] Aleieldin Salem and Sebastian Banescu. 2016. Metadata recovery from obfuscated programs using machine learning. In *ACM SSPREW*.
- [146] Shaown Sarker, Jordan Jueckstock, and Alexandros Kapravelos. 2020. Hiding in plain site: Detecting JavaScript obfuscation through concealed browser API usage. In *ACM IMC*. 648–661.
- [147] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2022. Loki: Hardening code obfuscation against automated attacks. In *USENIX Security*. 3055–3073.
- [148] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM CSUR* 49, 1 (2016), 4:1–4:37.
- [149] Sebastian Schrittwieser, Patrick Kochberger, Michael Pucher, Caroline Lawitschka, Philip König, and Edgar R. Weippl. 2022. Obfuscation-resilient semantic functionality identification through program simulation. In *NordSec*. 273–291.
- [150] S. A. Sebastian, S. Malgaonkar, P. Shah, M. Kapoor, and T. Parekhji. 2016. A study & review on code obfuscation. In *WCFTS (Startup Conclave)*. 1–6.

- [151] Serhii Semenov, Viacheslav Davydov, and Denys Voloshyn. 2019. Obfuscated code quality measurement. In *MMA*. 1–6.
- [152] Zihan Sha, Hui Shu, Xiaobing Xiong, and Fei Kang. 2022. Model of execution trace obfuscation between threads. *IEEE Trans. Dep. Sec. Comp.* 19, 6 (2022), 4156–4171.
- [153] Li Shijia, Jia Chunfu, Qiu Pengda, Chen Qiyuan, Ming Jiang, and Gao Debin. 2022. Chosen-instruction attack against commercial code virtualization obfuscators. In *NDSS*.
- [154] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) the art of war: Offensive techniques in binary analysis. In *IEEE S&P*.
- [155] Praveen Sivadasan and P. Sojan Lal. 2011. Suggesting potency measures for obfuscated arrays and usage of source code obfuscators for intellectual property protection of Java products. In *ICINT*.
- [156] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Anything to hide? Studying minified and obfuscated code in the web. In *WWW*. 1735–1746.
- [157] Yingbo Song, Michael E. Locasto, Angelos D. Keromytis, and Salvatore J. Stolfo. 2010. On the infeasibility of modeling polymorphic shellcode. *Mach. Learn.* 81, 2 (2010), 179–205.
- [158] Standard Performance Evaluation Corporation (SPEC). 1988. *SPEC — Standard Performance Evaluation Corporation*. <https://spec.org>
- [159] Jon Stephens, Babak Yadegari, Christian Collberg, Saumya Debray, and Carlos Scheidegger. 2018. Probabilistic obfuscation through covert channels. In *IEEE EuroS&P*. 243–257.
- [160] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Baishakhi Ray. 2018. Obfuscation resilient search through executable classification. In *ACM MAPL*. 20–30.
- [161] Anjali J. Suresh and Sriram Sankaran. 2020. A framework for evaluation of software obfuscation tools for embedded devices. In *ATIS*. 1–13.
- [162] Anjali J. Suresh and Sriram Sankaran. 2020. Power profiling and analysis of code obfuscation for embedded devices. In *IEEE INDICON*. 1–6.
- [163] M. Talukder, S. Islam, and P. Falcarin. 2019. Analysis of obfuscated code with program slicing. In *Cyber Security*. 1–7.
- [164] Xiaoxiao Tang, Yu Liang, Xinjie Ma, Yan Lin, and Debin Gao. 2017. On the effectiveness of code-reuse-based Android application obfuscation. In *ICISC*. 333–349.
- [165] Technische Universität Braunschweig. 2016. *The Drebin Dataset*. <https://www.sec.cs.tu-bs.de/~danarp/drebin/download.html>
- [166] Zhenzhou Tian, Hengchao Mao, Yaqian Huang, Jie Tian, and Jinrui Li. 2022. Fine-grained obfuscation scheme recognition on binary code. In *ICDF2C*. 215–228.
- [167] Dennis Titze, Michael Lux, and Julian Schuette. 2017. Ordol: Obfuscation-resilient detection of libraries in Android applications. In *IEEE Trustcom/BigDataSE/ICSS*. 618–625.
- [168] Ramtine Tofighi-Shirazi, Irina Asăvoae, Philippe Elbaz-Vincent, and Thanh Ha Lê. 2019. Defeating opaque predicates statically through machine learning and binary analysis. *arXiv preprint arXiv:1909.01640* (2019).
- [169] Ramtine Tofighi-Shirazi, Irina Măriuca Asăvoae, and Philippe Elbaz-Vincent. 2019. Fine-grained static detection of obfuscation transforms using ensemble-learning and semantic reasoning. In *ACM SSPREW*.
- [170] Denis Ugarte, Davide Maiorca, Fabrizio Cara, and Giorgio Giacinto. 2019. PowerDrive: Accurate de-obfuscation and analysis of PowerShell malware. In *DIMVA*. 240–259.
- [171] Jens Van den Broeck, Bart Coppens, and Bjorn De Sutter. 2021. Obfuscated integration of software protections. *Int. J. Inf. Secur.* 20, 73–101 (2021).
- [172] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *ESEC/FSE*. 683–693.
- [173] VirusShare.com. 2024. *VirusShare.com*. <https://virusshare.com>
- [174] Alessio Viticchié, Leonardo Regano, Cataldo Basile, Marco Torchiano, Mariano Ceccato, and Paolo Tonella. 2020. Empirical assessment of the effort needed to attack programs protected with client/server code splitting. *Empir. Softw. Eng.* 25, 1 (2020), 1–48.
- [175] Alessio Viticchié, Leonardo Regano, Marco Torchiano, Cataldo Basile, Mariano Ceccato, Paolo Tonella, and Roberto Tiella. 2016. Assessment of source code obfuscation techniques. In *IEEE SCAM*. 11–20.
- [176] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. 2018. Software protection on the go: A large-scale empirical study on mobile app obfuscation. In *ICSE*. 26–36.
- [177] Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu. 2016. Translingual obfuscation. *IEEE EuroS&P* (2016), 128–144.
- [178] Pei Wang, Dinghao Wu, Zhaofeng Chen, and Tao Wei. 2018. Protecting million-user IOS apps with obfuscation: Motivations, pitfalls, and experience. In *ICSE-SEIP*. 235–244.



- [179] Yongzhi Wang, Yulong Shen, Cuicui Su, Ke Cheng, Yibo Yang, Anter Faree, and Yao Liu. 2019. CFHider: Control flow obfuscation with Intel SGX. In *IEEE INFOCOM*. 541–549.
- [180] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. 2018. ORLIS: Obfuscation-resilient library detection for Android. In *MOBILESoft*. 13–23.
- [181] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Bradley Reaves, Patrick Traynor, and Sascha Fahl. 2018. A large scale investigation of obfuscation use in Google play. In *ACM ACSAC*. 222–235.
- [182] Carsten Willems and Felix C. Freiling. 2012. Reverse code engineering – state of the art and countermeasures. *Information Technology* 54, 2 (2012), 53–63.
- [183] P. Wrench and B. Irwin. 2016. Detecting derivative malware samples using deobfuscation-assisted similarity analysis. *SAIIE Africa Research J.* 107, 2 (2016), 65–77.
- [184] Zhenyu Wu, Steven Gianvecchio, Mengjun Xie, and Haining Wang. 2010. Mimimorphism: A new approach to binary code obfuscation. In *ACM CCS*. 536–546.
- [185] Dongpeng Xu, Binbin Liu, Weijie Feng, Jiang Ming, Qilong Zheng, Jing Li, and Qiaoyan Yu. 2021. Boosting SMT solver performance on mixed-bitwise-arithmetic expressions. In *ACM PLDI*. 651–664.
- [186] D. Xu, J. Ming, and D. Wu. 2017. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *IEEE S&P*. 921–937.
- [187] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R. Lyu. 2017. On secure and usable program obfuscation: A survey. *arXiv preprint arXiv:1710.01139* (2017).
- [188] Hui Xu, Yangfan Zhou, Jiang Ming, and Michael Lyu. 2020. Layered obfuscation: A taxonomy of software obfuscation techniques for layered security. *Cybersecurity* 3, 1 (2020), 9.
- [189] Babak Yadegari, Jon Stephens, and Saumya Debray. 2017. Analysis of exception-based control transfers. In *ACM CODASPY*. 205–216.
- [190] Zhou Yajin and Jiang Xuxian. 2012. *Android Malware Genome Project*. <http://www.malgenomeproject.org>
- [191] Ilsun You and Kangbin Yim. 2010. Malware obfuscation techniques: A brief survey. In *IEEE BWCCA*. 297–300.
- [192] Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2013. Obfuscation resilient binary code reuse through trace-oriented programming. In *ACM CCS*. 487–498.
- [193] Qiang Zeng, Lannan Luo, Zhiyun Qian, Xiaojiang Du, Zhoujun Li, Chin-Tser Huang, and Csilla Farkas. 2019. Resilient user-side Android application repackaging and tampering detection using cryptographically obfuscated logic bombs. *IEEE Trans. Dep. Sec. Comp.* (2019), 1–1.
- [194] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. 2019. LibID: Reliable identification of obfuscated third-party Android libraries. In *ACM ISSTA*. 55–65.
- [195] Xiaolu Zhang, Frank Breitinger, Engelbert Luechinger, and Stephen O’Shaughnessy. 2021. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *FSI: Digital Investigation* 39 (2021). DOI: <https://doi.org/10.1016/j.fsidi.2021.301285>
- [196] Xuesong Zhang, Fengling He, and Wanli Zuo. 2008. An inter-classes obfuscation method for Java program. In *ISA*. 360–365.
- [197] Xuesong Zhang, Fengling He, and Wanli Zuo. 2010. Theory and practice of program obfuscation. *Convergence and Hybrid Information Technologies* (2010), 426.
- [198] Xiaochuan Zhang, Jianmin Pang, and Xiaonan Liu. 2018. Common program similarity metric method for anti-obfuscation. *IEEE Access* 6 (2018), 47557–47565.
- [199] Yujie Zhao, Zhanyong Tang, Guixin Ye, Xiaoqing Gong, Dingyi Fang, and Zhiyuan Tan. 2021. Input-output example-guided data deobfuscation on binary. *Security and Communication Networks* 2021 (2021).
- [200] Yujie Zhao, Zhanyong Tang, Guixin Ye, Dongxu Peng, Dingyi Fang, Xiaojiang Chen, and Zheng Wang. 2020. Semantics-aware obfuscation scheme prediction for binary. *Comput. & Secur.* 99 (2020), 102072.
- [201] Hao Zhou, Ting Chen, Haoyu Wang, Le Yu, Xiapu Luo, Ting Wang, and Wei Zhang. 2020. UI obfuscation and its effects on automated UI analysis for Android apps. In *IEEE/ACM ASE*. 199–210.
- [202] Yan Zhuang, Mykola Protsenko, Tilo Muller, and Felix C. Freiling. 2014. An(other) exercise in measuring the strength of source code obfuscation. In *DEXA*. 313–317.

Received 18 July 2023; revised 3 October 2024; accepted 10 October 2024