Evaluation Methodologies in Software Protection Research — Supplemental Material

BJORN DE SUTTER*, Computing Systems Lab, Ghent University, Belgium
SEBASTIAN SCHRITTWIESER*, Christian Doppler Laboratory for Assurance and Transparency in Software Protection, Faculty of Computer Science, University of Vienna, Austria
BART COPPENS, Computing Systems Lab, Ghent University, Belgium
PATRICK KOCHBERGER, St. Pölten University of Applied Sciences, Austria

This document is the supplemental material of the main paper.

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

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

1 Overview

This supplemental material categorizes all papers according to their perspective (Section 2), lists our definitions of software protection methods (Section 3), analysis methods (Section 4), and measurements aspects and categories (Section 5). Furthermore, Section 6 provides additional information about the use of samples in papers, and Section 7 on the correlation of sample sets with publication venues. Sections 8 and 9 provide additional data on the papers' deployment of protection methods and analysis methods, respectively. Section 10 describes the actual tools that were most commonly used in the surveyed papers and provides some recommendations on their use in evaluations. Section 11 provides some additional information on the protections layered in experiments involving human subjects. Finally, Section 12 presents a more extensive discussion of other surveys in the domain of software protection, complementary to the shorter related work discussion in the main paper.

The first 202 references in the reference list at the end of this supplemental material are the same as the 202 references in the main paper, in the same order, such that they have the same number. The additional references give a complete picture of all 571 surveyed papers.

2 Paper Categorization along Perspectives

This Venn diagram in Figure 1 includes references to all 571 papers of our survey. As discussed in Section 2.3 of our main paper, the survey includes many more defensive papers than offensive ones. This is confirmed by the dominance of the blue goodware papers in the obfuscation part of the Venn diagram and the dominance of the red malware papers in the analysis part.

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

Bjorn De Sutter, bjorn.desutter@ugent.be, and Bart Coppens, bart.coppens@ugent.be, Computing Systems Lab, Ghent University, Technologiepark-Zwijnaarde 126, 9052, Gent, Belgium; Sebastian Schrittwieser, sebastian.schrittwieser@univie.ac.at, Faculty of Computer Science, University of Vienna, Kolingasse 14-16, 1090, Wien, Austria; Patrick Kochberger, patrick.kochberger@fhstp.ac.at, St. Pölten University of Applied Sciences, Campus-Platz 1, 3100, St. Pölten, Austria.

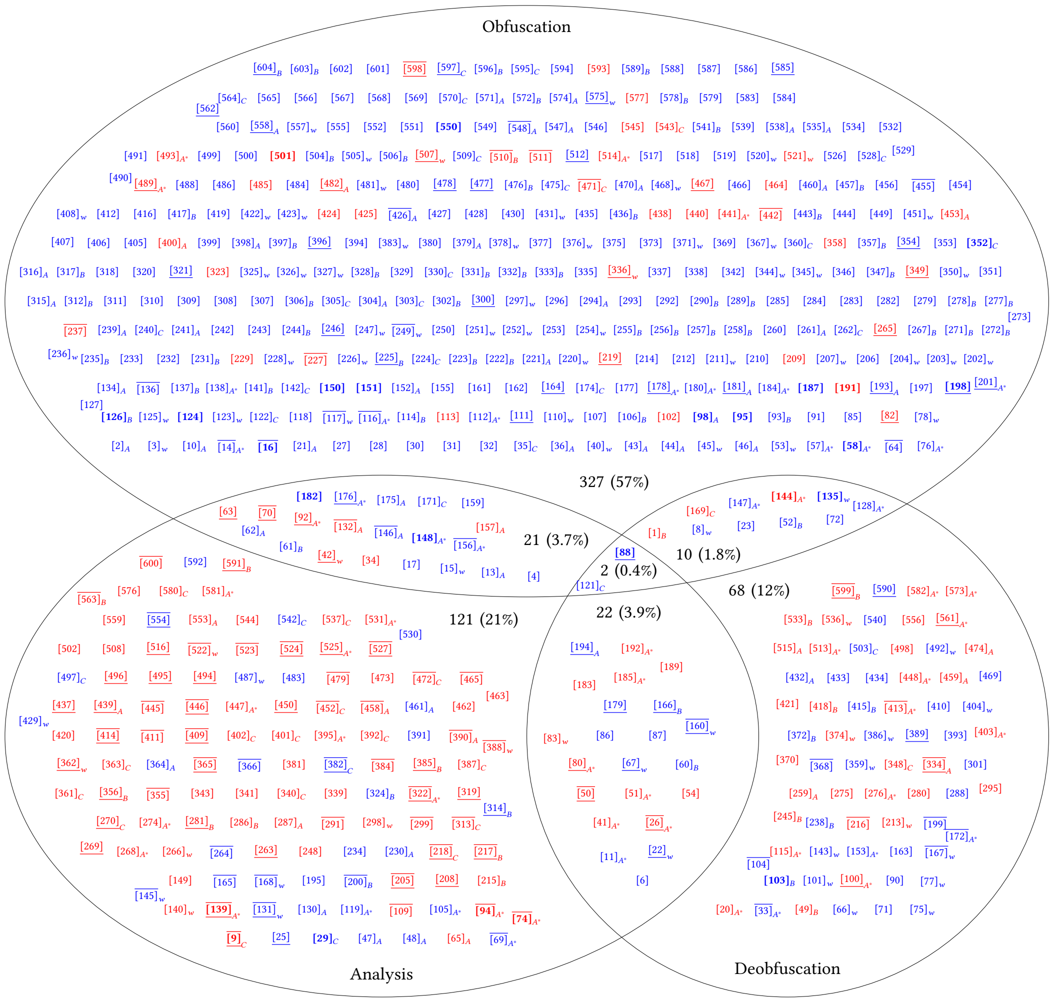


Fig. 1. Venn diagram of the 571 surveyed goodware and malware papers. Papers targeting mobile platforms are underlined. Overlined papers use data science methods such as machine learning and deep learning. Bold references are **surveys and literature and meta analyses**. A subscript index indicates the CORE ranking of the paper's publication venue, ranging from A^*, A, B, to C, and w for workshop. An absent index denotes that the venue is national, new, or unranked.

3 Protection Definitions

For each surveyed paper, we recorded which types of novel protections were presented, which protections were included in the samples of their experimental evaluation, which protections were discussed in their (theoretical) security analysis, and which protections were surveyed (in the case of surveys). So for each paper in scope, one or more of the protections listed below were marked.

Importantly, concrete protection transformations, as presented in papers, can match multiple classes. For example, a tool can inject opaque predicates based on aliasing pointer computations as a means to inject spurious data dependencies. Such a protection is classified as "opaque predicates", "aliasing", and "data flow transformation". Some classes are conceptually subclasses of others. For example, "control flow indirections" such as branch functions are a specific type of "control flow transformations". Whenever some protection discussed in a paper matches a more concrete subclass, we only mark that subclass, not the superclass. Furthermore, if some transformation is merely a side-effect of another one, we do not mark it. For example, "parallelization" inherently implies control flow and data flow transformations. However, we only mark the latter if the parallelization is used as an enabler of a specific flow transformation, for example when implicit data flow is implemented by measuring timing differences between fast and slow threads [128].

Furthermore, it is essential to raise the issue of potential bias. We collected data by interpreting the information and claims in the papers. Some papers present explicitly and exhaustively which protections the authors deployed; others do not. For instance, many malware papers only mention where they got their malware samples but do not discuss in detail how those samples were generated or obfuscated, in many cases because that ground-truth information is not public. Furthermore, some authors detail how they configured publicly available obfuscation tools; others do not. Some such tools have minimal or default configurations; others do not. For some (malware) data sets, descriptions of the deployed protections are available in the literature that we are aware of; for others, this is not the case. For each paper, we only marked protections if we could determine without reasonable doubt that they had been deployed on samples. Our discovery process mostly involved the papers themselves, any references in them to other papers discussing the used data sets, and publicly available descriptions of the used tools. When individual malware samples or families were mentioned without discussing the protections used in them, we did not, however, go as far as looking up external reverse engineering blogs or analysis reports on those individual samples. As a result, the reader should be aware that for quite some papers, in particular malware analysis papers, protections deployed on their samples might not be recorded as such. For goodware papers, in particular papers in the obfuscation perspective (for which authors typically generated their protected samples themselves), this is less of an issue.

Data encoding/encryption (DEN). Encrypting or changing the type, encoding, and bit representation of data in a program. This includes replacing static data (e.g., strings) with compressed or encrypted versions. It also excludes the special case of white-box cryptography in which the keys are replaced by other data or code.

Static data to code conversion (D2C). Replacing constant data available in a static program representation by computations that reproduce the data at run time. If those computations merely involve decryption or decompression, we consider it data encoding/encryption, not static data to code conversion.

White-box cryptography (WBC). Implementation of cryptographic primitives such that stored and used keys no longer occur in plain text in the binaries or in the program state during execution.

Mixed Boolean-arithmetic (MBA). Code and data obfuscation technique in which computations are replaced by more complex mixes of boolean and arithmetic operations.

Data flow transformation (DFT). Introducing fake, covert, or implicit data dependencies with the goal of obfuscating the data dependencies. Includes anti-taint protections (over-tainting as well as under-tainting), but not MBA which has its own category.

Data transformation (DTR). Splitting, merging, and reordering of structured data; changing the shape or layout with which data is stored in memory.

Aliasing (ALI). Obfuscations building on or targeting potential aliases among pointers, e.g., to make data analyses imprecise or to yield larger points-to sets. Includes replacing direct data accesses with indirect ones, e.g., through reflection.

Class based transformations (CBT). Class hierarchy transformations, class type hiding, class refactoring, transformations with overriding methods. This includes adding fake/dead methods to classes to hamper class-based analyses.

Code reordering (CRE). Reordering of instructions, functions, basic blocks, expressions, or other code elements; changing the alignment/padding of binary code fragments.

Code diversity (DIV). Replacing instructions or sequences thereof with other equivalent ones in the same ISA (e.g., CALL replaced by PUSH & JMP). Diversity can be spatial (diversified instances exist at the same point in time), or temporal (code within a software instance is diversified over time).

Control flow flattening (CFF). Obscuring structured control flow graphs by replacing them with a flat structure and a data-controlled dispatcher that controls the order in which code fragments are executed. The goal is to hide the original structure.

Opaque predicates (OPP). Using boolean-valued expressions whose values or other invariant properties are known at obfuscation time but difficult for an attacker to figure out, e.g., to steer execution around inserted bogus control flow transfers. The goal is to inflate the complexity of the code's CFGs.

Control flow indirections (CFI). Replacing direct control flow transfers by indirect ones, which can be, e.g., indirect calls or jumps, faulting instructions and exception handlers, uses of reflection. The goal is to hide the call graph's edges or the control flow graph's edges from static analysis, not to change those graphs.

Control flow transformation (CFT). All other ways of obfuscating the control flow, except control flow flattening, opaque predicates, and control flow indirections. The goal here is to change the control flow graph, not to hide edges from static analyses.

Function transformation (FUT). Transformations at the granularity of functions, such as wrapping, cloning, splitting, merging, inlining, and outlining of functions or methods. This excludes protections based on adding fake/dead methods to classes to hamper class-based analyses.

Identifier renaming (IRE). Renaming identifiers such as the names of variables, sections, functions, methods, classes, registers, etc. to remove information useful for an attacker or to complicate analyses.

Junk code insertion (JCI). Inserting junk bytes, dead code (which can be executed but without impact on the program semantics), or unreachable code (which cannot be executed).

Library hiding (LIH). Hiding which library functions are called, e.g., by making calls indirect or by inlining library functions. If this is achieved by means of specific transformations, such as encrypting the names of the functions, those transformations are also marked.

Loop transformations (LOT). Transformation of loops, including fusion, unrolling, splitting, reordering iterations, as well as altering counters, exit conditions, etc.

Overlapping Code (OLC). Overlapping instructions or functions. Native overlapping instructions share bytes in the binaries or in text segments. Overlapping functions share instructions, i.e., some instructions are executed as part of multiple functions.

Parallelizing (PAR). Injecting additional threads or processes or other forms of parallelism into a program in order to complicate analysis, i.e., to make it harder to use analysis tools and to obtain precise models of the software's data flow and control flow.

Repacking (RPA). Recompilation, reassembly, or repackaging of code taken from one software component into another.

Anti-debugging (ADB). Techniques for preventing the use of debuggers and debugging techniques.

Code mobility (CMO). Removing static code in a distributed program by equivalent code that is downloaded dynamically from a (secure) server to hamper static analysis.

Server side execution (SSE). Removing code fragments from a distributed program and instead having equivalent code executed on a remote (secure) server to hamper static and dynamic analysis.

Dynamic code modification (DCM). Just-in-time compilation, self-modifying code, custom dynamic class loading, and use of functionality such as JavaScript's `eval()` function that allows to execute code extracted from (dynamically generated) strings. This excludes packing/encryption techniques, as discussed below.

Packing/encryption (ENC). Packing/encryption/compression of software components at different levels of granularity (binaries, text sections, functions, basic blocks, ...) to be unpacked/decrypted/decompressed at run time.

Environmental requirements (ERE). Checks to identify being emulated, checks for path variables, and for other properties of the host system on which the software runs to detect and block the use of certain tools. Excludes anti-debugging techniques that check for the presence of a debugger or try to prevent being debugged. Includes techniques to delay execution to avoid detection in short-running dynamic analysis.

Hardware-assisted protection (HWO). Hardware-assisted protections such as USB-dongles, SGX enclaves, hardware-supported software encryption, etc.

Virtualization (VIR). Virtualization-based obfuscation, i.e., replacing code in a native, well-known real or virtual ISA by code in a custom virtual ISA, and an interpreter thereof. This includes lightweight forms in which only opcodes are remapped from their standard values to a custom numbering.

4 Analysis Definitions

Similar to how we analyzed the deployment of protections, we classified the papers' use of code analysis methods and features thereof to evaluate the strength of obfuscations against attackers' toolboxes.

Earlier remarks on our evaluation of deployed protections in the previous section regarding sub-/superclasses of methods and possible biases also apply here.

Abstract interpretation (AI). General theory of the sound approximation of a program's semantics. Parts of the program are abstracted (simplified) and then interpreted step-by-step.

Constraint based analysis (CBA). Defining the specifications of a program analysis in a constraint language and using a constraint solver to automate the implementation of the analysis.

Control flow analysis (CFA). Analyzing a program's control flow on the basis of reconstructed control flow graphs or call graphs.

Cryptanalysis (CRA). Custom analyses techniques of cryptographic algorithms, this includes algebraic attacks.

Disassembly / CFG reconstruction (DIS). Techniques to disassemble a program (i.e., identify its instructions) and to (re)construct the functions and their control flow graphs. This is only marked for publications that specifically aim for hampering or improving the disassembly process and the CFG reconstruction, not for papers that merely consider reconstructed control flow graphs or disassembled code as a starting point for further analysis.

Diffing (DIF). Similarity detection; use of diffing tools; looking for similar patterns or corresponding code fragments in one or more software versions. This excludes the identification of a priori determined patterns, in which case the technique would be classified as pattern matching.

Data flow analysis (DFA). Analyzing the data flow and data dependencies in a program. This includes performing alias analysis. Excludes slicing.

Dynamic event monitoring (DEM). Monitoring of events during the execution of a program. This includes API call and system call monitoring, function hooking (e.g., via interposers or detours), code breakpoints, and data watches in debuggers, etc.

Fuzzing (FUZ). Analyzing the operation or behavior of a program when executing it on numerous inputs, which might be invalid, unexpected, or random data. This includes brute-forcing.

Human analysis (HUA). Manual analysis conducted by a human, manual reverse engineering activities such as studying code fragments to comprehend them, or browsing through lists of filtered fragments.

Library dependency analysis (LIB). Analyses to identify used/imported/exported libraries and invocations of their APIs.

Lifting (LIF). Decompilation of code in a concrete low-level representation to a more abstract (intermediate) representation. This excludes the mere disassembling of code.

Machine Learning (ML). The use of machine learning for optimizing or training an analysis. Can be combined with other analyses. For example, when a pattern matcher is optimized using machine learning techniques, the paper is classified as using both pattern matching and machine learning.

Memory dumping (MD). Obtaining one or more snapshots of (part of) the memory state of a program under execution for analysis.

Model checking (MCH). Using formal methods to prove or disprove the correctness of a certain system. Excludes the use of model checking techniques to identify path conditions as part of symbolic execution or fuzzing.

Network analysis (NEA). Sniffing of network traffic, looking at the network data from outside of the binary.

Normalization (NOR). Performing a code normalization/canonicalization step.

Out-of-context execution (OOC). Executing parts of a program on chosen inputs without executing the whole program as is. This can be achieved with a debugger by injecting a custom "main" function into an existing program, by interposing function calls, by extracting code from a program and repacking it into another program, etc.

Pattern matching (PAT). Signature-based analysis, pattern matching on, e.g., data, call sequences, instruction sequences, regular expressions obtained somehow, etc.

Sandboxing (SAN). Emulating or simulating a program, or running it in a virtualized environment to observe its execution.

Slicing (SLI). Generating a subset of a program that includes all code statements that might affect the value of a variable at a certain statement for all possible inputs.

Statistical analysis (SAN). All kinds of program analyses that rely on statistics. This includes, e.g., entropy measurements and counting the occurrences of certain instructions or values or events, etc. Importantly, this only includes the use of statistical methods during an attack or analyses, but excludes the a priori use of statistics during the learning phases of analyses based on machine learning. Furthermore, a paper is not classified as using statistical analyses if statistics are merely used to evaluate a presented analysis technique in the paper.

Symbolic execution (SYM). Exploring many possible execution paths by interpreting the execution of code on symbolic rather than concrete inputs. Analyzing which paths can be executed and under which conditions.

Taint analysis (TNT). Marking and tagging variables/input/output or other data of a program and tracking which data and computations depend on the tagged data or on which the tainted data depends.

Tampering (TAM). Statically or dynamically altering a program or its state. In this survey, we only consider tampering with the goal of enabling some analysis (e.g., to circumvent anti-debugging techniques or to revert obfuscations), not tampering to obtain a program with altered functionality. Furthermore, tampering excludes transforming an obtained program representation without altering the program or its state itself.

Theorem proving (THE). Theorem proving includes SAT/SMT analysis. We do not classify a paper as relying on theorem proving if it is only used to determine path execution conditions during symbolic execution or fuzzing.

Tracing (TRA). Generating or analyzing a trace of a program's execution, i.e., a list of the executed operations and possibly of other features, such as the data on which they operate.

Type analysis (TYP). This includes type inference and analyses of the class hierarchy.

5 Measurements Definitions

For each surveyed paper, we also recorded which aspects of samples, which effects of protections on samples, and which properties of analyses and analysis results were measured in the experimental evaluation, and why those were measured. We consider four possible reasons to perform measurements: to quantify stealth, potency, resilience, and costs. The first three, relating to protection effectiveness, can share concrete metrics, depending on the considered analyses and protections. In other words, the same or similar concrete measurements can be used to evaluate stealth, potency, or resilience, depending on the perspective and context. For costs, there is less ambivalence: all measurements of different forms of costs serve only one purpose: measuring cost.

Stealth. The stealth of a protection is a measure of how difficult it is to detect that the concrete protection, its class, or particular aspects of are present in a program and where. Protections in malware are stealthy if the presence of the protection is hard to detect, not if the malware is hard to classify as malware. We mark a paper as measuring stealth if we interpret the paper as evaluating it through some measurement, independent of whether or not the authors explicitly mention stealth.

Potency. The potency of a protection is the effect that the protection has to make some analysis harder, meaning the analysis will require more time or resources to reach the same result or that the result will be less precise or less useful. Potency is typically obtained by increasing the (apparent) complexity of the object to be analyzed and/or by lowering its suitability as input for an analysis. Potency is hence always measured or defined with respect to one or more concrete or conceptual analyses or classes thereof. Historically, only human, manual analysis are considered for potency, but we extend it to any analysis. For example, when facing malware detection techniques, a deployed protection is potent if it succeeds in thwarting one or more (state-of-the-art) malware detection and classification techniques. We mark a paper as measuring potency if we interpret the paper as evaluating potency through some measurement.

Resilience. The resilience of a protection is a measure of how difficult it is to counter-attack the protection, i.e., to lower its potency with respect to some analysis. Counter-attacks can come in the form of adaptations to that analysis or in the form of analyses and transformations that can be executed a priori. Resilience can also be measured with respect to one or more analyses. For example, a potent protection used to protect malware against detection is also resilient if it is hard to come up with improved or alternative detection and classification techniques to mitigate the protection's potency or if that potency can only be mitigated partially. We mark a paper as measuring resilience if we interpret the paper as evaluating resilience through some measurement, even in case the paper does not explicitly mentions resilience.

Classification statistics. These are the traditional metrics derived from false rates and true rates of identification techniques. We mark papers in this category if the present precisions, recalls, F-scores, etc. This excludes malware classification, for which we have a separate category.

Malware classification statistics. When a paper reports classification metrics for the binary classification of samples as malware or goodware, we put it in this category.

Deltas/similarity. Does the paper measure how similar two or more samples or derivations thereof are, such as original, unprotected programs vs. deobfuscated protected programs?

Entropy. Does the paper measure or discuss entropy (or related statistical properties) of some artifacts of a sample?

Size. Does the paper measure or discuss the impact of a protection on the size of a sample (e.g., file size)?

Manual analysis. Does the paper discuss or evaluate the complexity of a manual, human analysis?

Controlled human experiment analysis. Does the paper present the result of a controlled experiment with human subjects (e.g., with students, reverse engineers, developers, penetration testers, etc.) related to software protection?

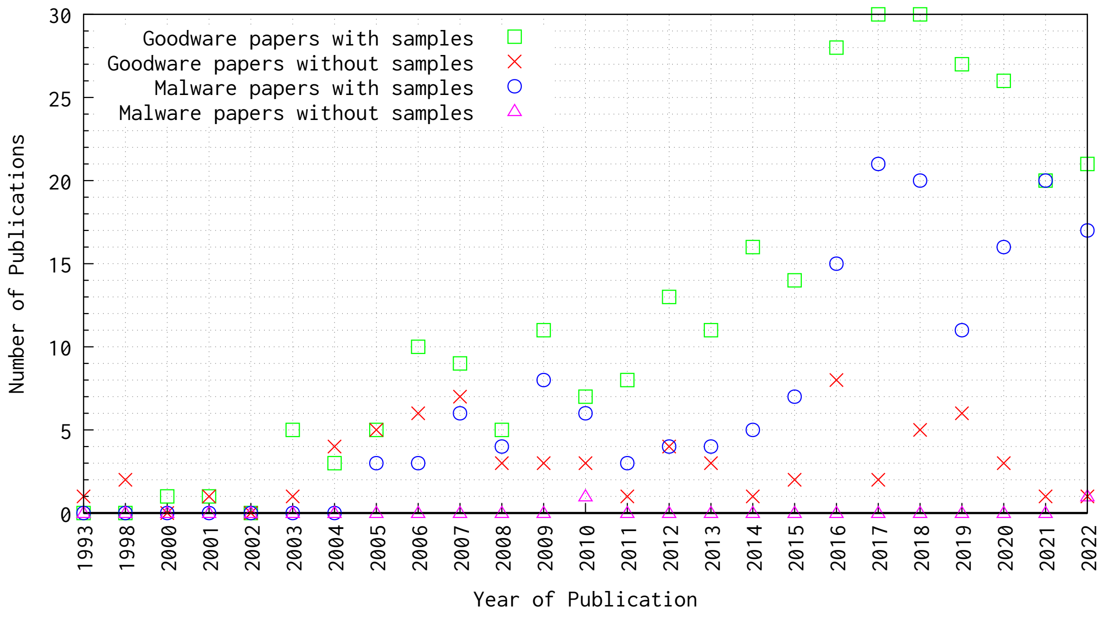


Fig. 2. Number of papers per year with and without samples. Although the number of papers increases because of the methodology as discussed in Section 2.1 of the main paper), the number of papers without samples in total per year does not increase.

Memory usage. Does the paper report or discuss the impact of a protection on the memory consumption of a sample? Measurements on memory usage of a performed analysis were put into the *Other costs* category.

Opcode distribution. Does the paper evaluate the distribution of opcodes/instructions in a sample?

Code complexity. Does the paper report code complexity measurements, such as branching complexity, cyclomatic complexity, points-to set sizes, etc.

Applicability (code coverage). Does the paper report on what fraction of the samples' code a protection can be deployed?

Power consumption. Does the paper evaluate a protection's effect on power consumption? Measurements on power consumption of a performed analysis were put into the *Other costs* category.

Attack/analysis overhead time. Does the paper evaluate how much (more) time certain attacks or analyses require (except for manual effort) on samples?

Compilation time. Does the paper evaluate how long it takes to add a protection to samples?

Performance overhead time. Does the paper report the run-time overhead of protection?

Other costs. Does the paper present measurements of any other costs of a protection/analysis not listed above?

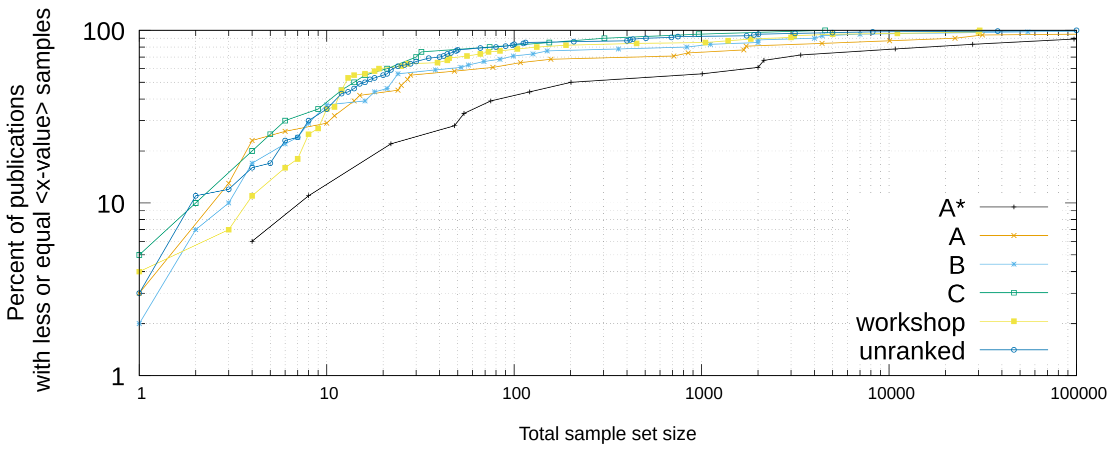


Fig. 3. Cumulative graph of the sample size for goodware publications. On the x-axis we have the sample set size. On the y-axis we have the percentage of papers lower or equal to the current x-value.

6 To Sample or not to sample

Figure 2 shows, per paper category, how many papers perform an evaluation on samples and how many lack such an evaluation. This information complements that of Figure 3 in the main paper. The low number of malware papers without samples is not surprising, given that one of the rules we used to categorize papers as malware papers is specifically that the paper uses malware samples in its evaluation, as discussed in Section 2.3 of the main paper. Over the years, we do not see a general trend in the absolute number of papers without samples. The relative numbers (i.e., comparing papers without samples to total number of papers per year) indicate a downward trend.

7 Correlation between Publication Venue Ranking and Sample Set Sizes

The cumulative graph for total sample set sizes in goodware papers in Figure 3 further illustrates the observation from Section 3.3 from the main paper on the correlation between sample total set size and publication venue ranking. In particular, for A* venues, but also for A and B venues, it is clear that their papers consider larger sample sets than lower ranked venues.

8 Deployment of Protections

Figure 4 visualizes how many papers feature implementations of protections. It features six crosstabs for different categories of papers. Next to each category inside the crosstab, the total number of papers reported in the crosstab is given, i.e., how many papers in that category had at least one protection implemented, as well as the top value of the color scale, i.e., the largest number occurring in the crosstab, which differs for each crosstab. Note that we included the eleven goodware papers and seven malware papers with both obfuscation and deobfuscation/analysis contributions in both of the corresponding crosstabs. These low numbers of multi-perspective papers with protection implementations confirm our earlier discussion in Section 2.3 of the main paper about few papers presenting both cat and mouse moves in the ongoing war between defenders and attackers.

Each cell in the left part of each crosstab on Figure 4 reports the number of papers that deploy at least the protection classes mentioned in its column and row. The diagonal shows how many papers deploy each type of protection; the other cells show in how many papers at least two protection classes were deployed.

For each row, the right part of each crosstab in Figure 4 presents the distribution of the number of protections deployed in the papers that also deploy that row's protection. The top row of the right part sums up the number of papers that deploy 1, 2, ..., up to "7 and more" different protections.

The studied protections differ from one type of programming language to another. This can be observed by comparing the refinements of Figure 4, where similar crosstabs are presented in Figures 5 to 7 per type of programming language: script, managed, and natively compiled.

8.1 Individual Protections

Complementary to the main observations discussed in Section 4.2 in the main paper, this section analyzes the popularity of different protections for different types of languages.

Identifier renaming is popular in script and managed languages such as JavaScript and Java, obviously because symbolic information such as class, field, and method names cannot be stripped from such software, they can only be obfuscated. It is not popular in native languages, although some papers do research the obfuscation of C source code for protecting it when it is distributed as source code.

For script languages, few other protections are frequently studied. Only data encoding/encryption, data-to-code-conversion, and data transformation are deployed frequently on scripts, typically for obfuscating strings by encoding (as data or code) and splitting them. And in fact many protections have not been researched at all for script languages. For some techniques, this is probably the case because they are not (directly) applicable to script languages. Hardware-based obfuscation and overlapping code are examples. For other techniques such as parallelization, which are not frequently researched for any type of language, the reason for those not showing up in script obfuscation papers is probably the relatively low sample size, i.e., the low number of papers. One clear exception is virtualization, which is frequently researched for natively compiled languages and even a couple of times for managed languages, but not once for script languages. We consider this a useful opportunity for future research, complementary to existing techniques, which, e.g., compile JavaScript to WebAssembly as a form of obfuscation [493].

Almost all papers that consider class-based transformations target managed languages, as do the vast majority of the papers that consider repackaging. In the case of repackaging, the reason is that it is mostly used in the context of Android malware apps, which consist mostly of Java bytecode. In the case of class-based transformations, this is due to the object-oriented nature of Java and the fact that papers targeting natively compiled code focus on obfuscations that can be deployed on imperative C code. This is the case, e.g., for all papers that rely on the Tigress, Obfuscator-LLVM, and Diablo-based obfuscators. While the latter two can, to some extent, also protect IR and binary code originating from C++ source code, they feature no protections specifically targeting classes.

Regarding control flow obfuscations, we notice that opaque predicates, control flow indirections, and the more general category control flow transformation are less popular for script languages than for managed and native languages. The one exception is control flow flattening, which is far less popular for managed languages than for script, let alone native languages. We can only speculate on the reason for this, taking into account that managed language software is obfuscated mainly by transforming bytecode, in particular, Java bytecode (and the DEX-variant thereof) as we observed in Section 4.1 in the main paper. While we see no fundamental issue for implementing global (i.e., function-wide) obfuscating transformations such as control flow flattening for Java or DEX bytecode, we do think that it is more

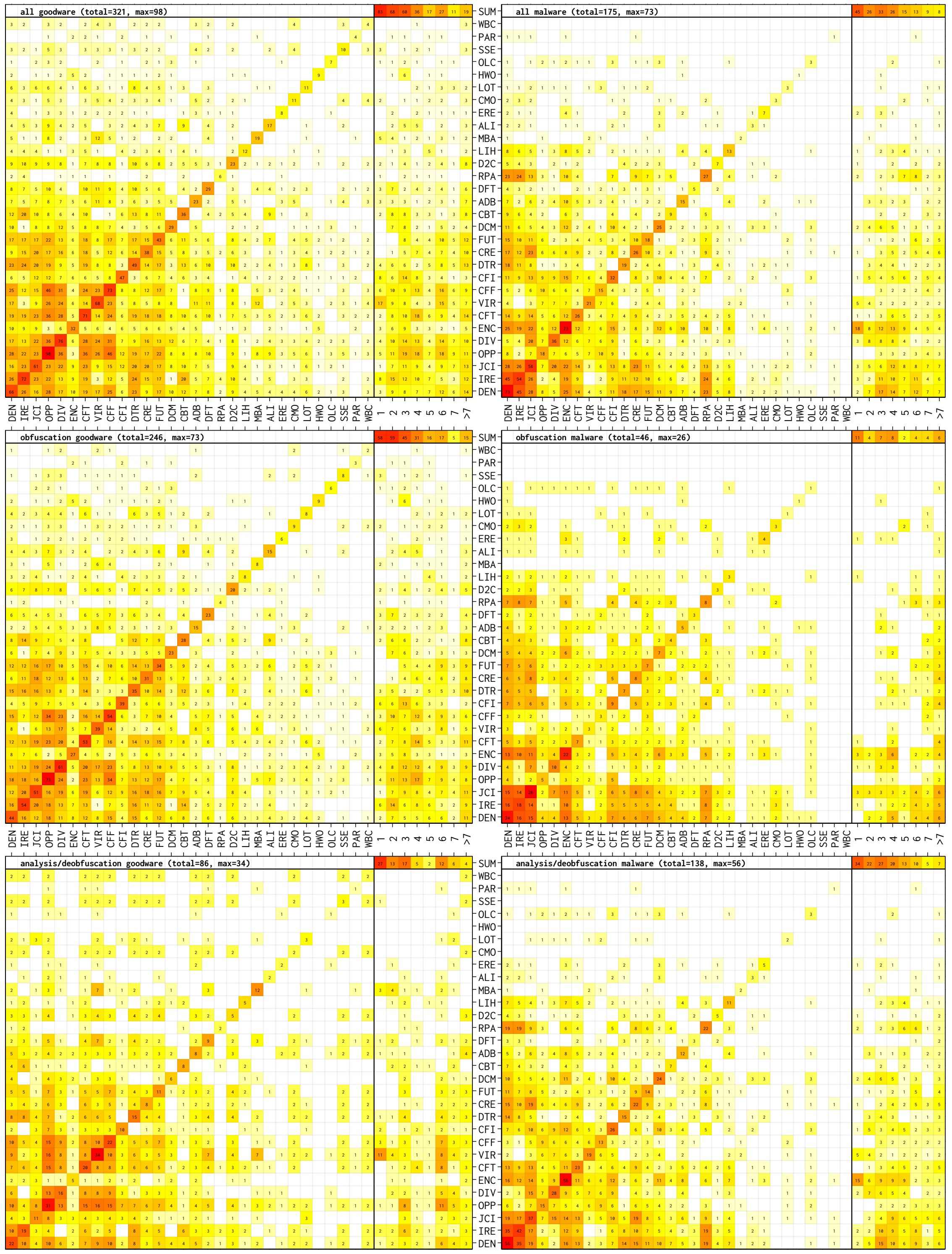


Fig. 4. Protections: Six crosstabs visualizing how publications combine or layer protections. The columns on the right indicate for each protection how many protections are in the papers that deploy that protection. The acronyms are defined in Section 3.

complex than implementing control flow flattening in script source code or in native code compiled from C source code. In particular, we think that the omnipresence of exception handling constructs in Java code could be a reason. Exception handling is implemented in Java bytecode by means of exception tables that contain an entry for each exception that is caught by each try block. Randomly mixing code fragments from a method body, as is done for control flow flattening, requires rewriting and extending the exception tables. This adds complexity to the obfuscators, but it might also reduce the strength of the protection, as the rewritten exception table allows one to identify which flattened fragments likely belonged together in the original code.

Looking specifically at natively compiled languages, we notice that data transformation is much less popular there than for other types of languages. We conjecture this is due to pointer aliasing complicating data flow analysis, a prerequisite to deploying data transformation.

Regarding dynamic code modification, we observe that it is quite popular for script and native languages but less so for managed languages. Importantly, the nature of this protection is very different in the three types of languages. In native languages, dynamic code modification denotes binary code being updated/rewritten as the program executes. In managed languages, dynamic code modification denotes custom class loading to alter which code gets loaded and executed. In script languages, it denotes dynamic generation of source code not present statically, such as with an `eval()` function that is provided a string decrypted at run time.

Anti-debugging is much less popular in managed language research than in script and natively compiled languages research, despite anti-debugging being used in the vast majority of Android apps [25]. The reason for this discrepancy is not clear.

8.2 Combinations of Protections

We also observed some interesting relations between protections. For example, almost all malware papers that deploy repackaging also deploy data encoding/encryption and identifier renaming. This is not surprising, as data encoding/encryption and identifier renaming are absolutely necessary to avoid trivial detection of repackaging. In this regard, these papers do show maturity. Likewise, the vast majority of the papers deploying data transformation or identifier renaming also deploy data encoding/encryption. This is also not surprising, as all of them are typically needed to mitigate trivial malware detection.

For goodware papers, we notice that of those deploying control flow flattening, 63% $\hat{=}$ 46/73 also deploy opaque predicates. Both are popular control flow obfuscations of which the implementation requires similar kinds of analyses and transformations, so this correlation is not surprising. Of the 11 goodware papers that deploy loop transformations, 73% $\hat{=}$ 8/11 also deploy data transformation. This is also no surprise: transforming how data is stored in arrays and transforming the loops iterating over the array's elements go hand in hand. Perhaps more surprising is that of the 19 goodware papers in which MBA is deployed, 63% $\hat{=}$ 12/19 also deploy virtualization. This is not because MBA or virtualization have any similarity or dependence in how they are deployed. Instead, they are related through the analysis with which they are attacked, such as dynamic symbolic execution or code synthesis techniques that can be used to deobfuscate, and are hence evaluated on, both MBA expressions and bytecode instruction handlers [14, 33, 66, 128, 135, 147].

9 Deployment of Analyses

Figure 8 depicts the use of individual analysis methods and combinations of them for each paper category, similar to how previous crosstab figures visualized protections. Furthermore, Figure 9 depicts which protections are being

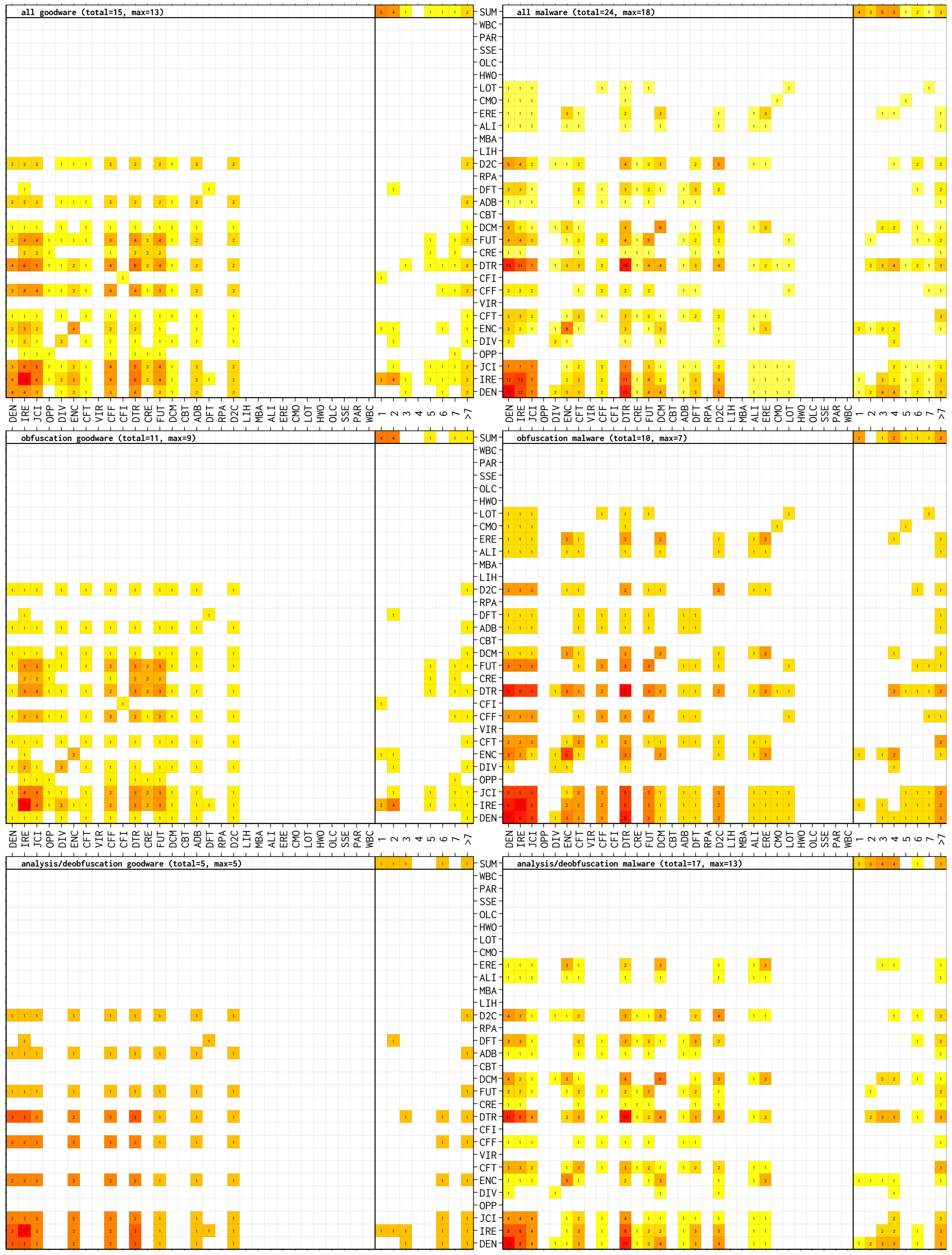


Fig. 5. Script language protections: Six crosstabs visualizing how publications targeting script languages combine or layer protections, similar to Figure 4. The acronyms are defined in Section 3.

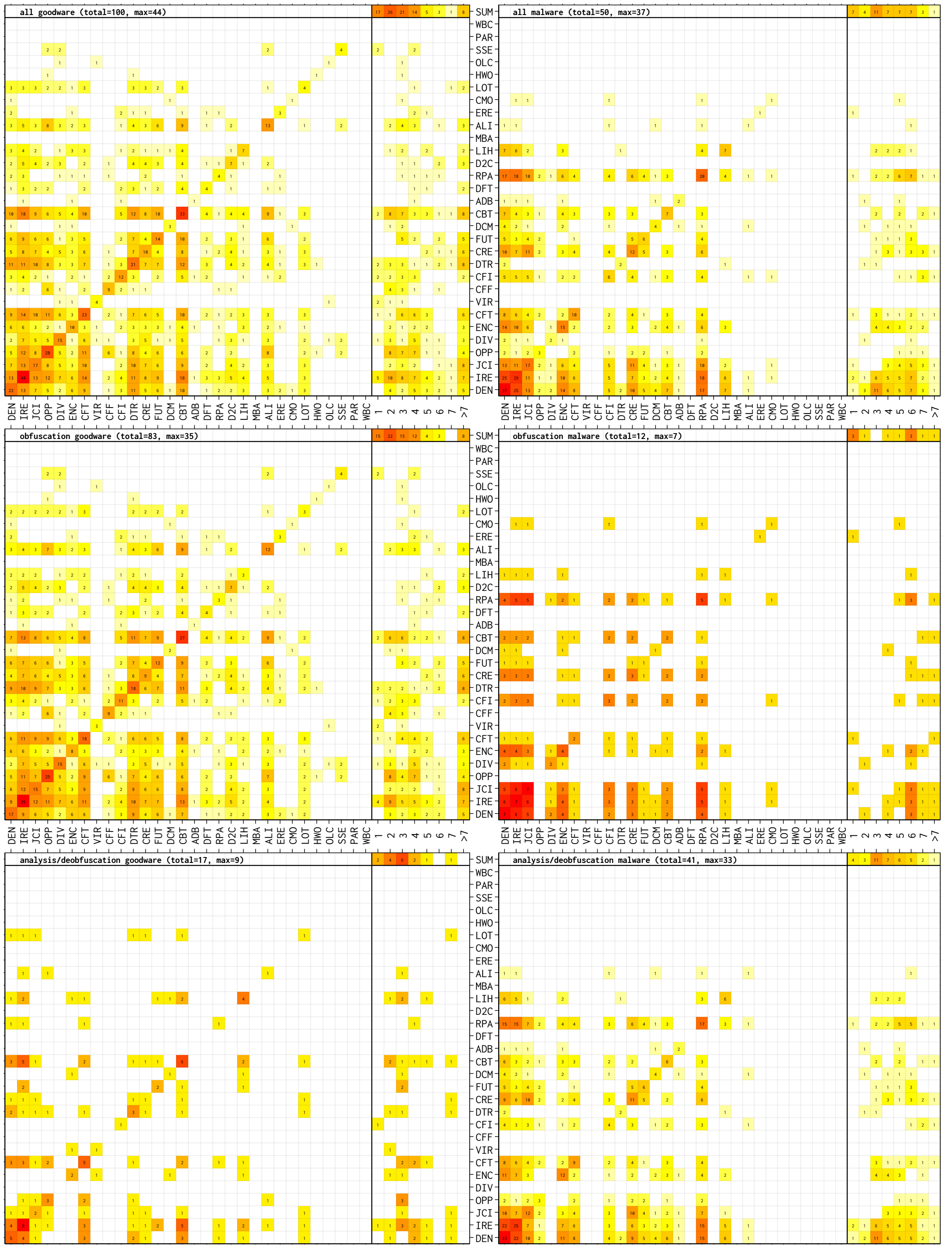


Fig. 6. Managed language protections: Six crosstabs visualizing how publications targeting managed languages combine or layer protections, similar to Figure 4. The acronyms are defined in Section 3.

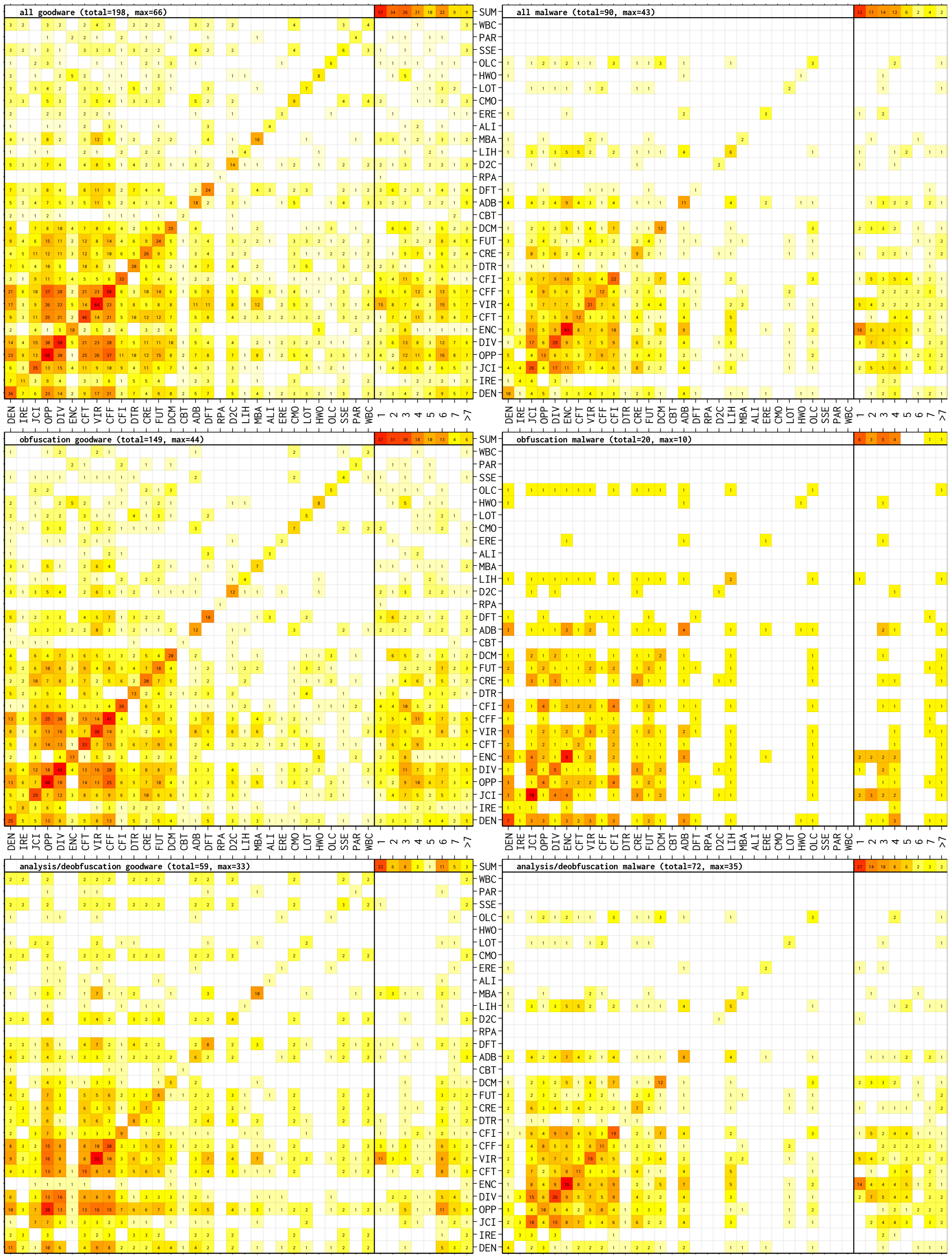


Fig. 7. Native language protections: Six crosstabs visualizing how publications targeting native languages combine or layer protections, similar to Figure 4. The acronyms are defined in Section 3.

evaluated with which analysis techniques in the surveyed papers. The analysis in Section 4.3 of the main paper is mostly based on the numbers in these crosstabs. The ones in Figure 9 can help researchers who design new obfuscations to select the most appropriate analysis to evaluate their obfuscations's strengths, assuming that popularity relates to appropriateness. Vice versa, those crosstabs can help researchers of new analysis techniques identify the obfuscations with which they can stress-test their analysis. In this regard, these crosstabs complement the survey by Schrittwieser et al. [148].

10 Software Protection Research Tools

Table 1 is an extended version of Table 4 in the main paper, now including the references to the papers that used the tools.

The tools used in at least five surveyed papers can be categorized into compilation, obfuscation, analysis, and anti-virus tools. When some tool can be used for multiple purposes, we list in the category of its most popular usage. Unless noted otherwise, tools listed here are actively being developed and maintained at the time of writing of this paper.

For many tools, we provide concrete recommendations on their usage for SP research. These recommendations are tool-specific instantiations of the general recommendations in Section 7 of the main paper. Importantly, the tool-specific recommendations need to be understood as applying to classes of tools: While we only formulate them for the more popular tools listed in Table 1 and below, they can also apply to similar, non-listed tools. For example, the recommendations provided below for IDA Pro also hold for Binary Ninja and Ghidra. Similarly, the recommendation for TXL can be extended to any source-to-source rewriter with which one could implement source-level obfuscations.

10.1 Compilation tools

Recommendation: Whatever compilers with code optimization capabilities you use to generate natively compiled samples, make sure to specify the compiler version being used and the optimization flags. Compilation of natively compiled code without any optimization (i.e., with `-O0`, which is the default option for most compilers) is absolutely not acceptable, so make sure you provide a proper optimization level flag.

LLVM[1] is an open-source collection of modular and reusable compiler and toolchain technologies. Its core libraries for optimization and code generation are built around a well-specified code representation known as the LLVM IR. This IR is also suitable for deploying obfuscations (e.g., with OLLVM as will be discussed below), for symbolic execution (with KLEE as will be discussed below), and as a target for code lifting techniques. LLVM is hence used in research as a build tool, as a protection tool, and as an analysis tool.

Clang[2] is the open-source frontend of LLVM for C-like languages, including C, C++, and Objective-C. It can also be used for source-to-source transformations and, hence, for protecting software.

GCC[3] or the open-source GNU Compiler Collection is Linux' default compilation tool flow. As GCC's design is not as modular as LLVM, it is much less popular for implementing protections, and it is used much less for analysis. As for the latter, GCC is only used to assess the resilience of obfuscations against compiler-like code analysis and optimizations.

[1]https://llvm.org/
[2]https://llvm.org/
[3]http://gcc.gnu.org/

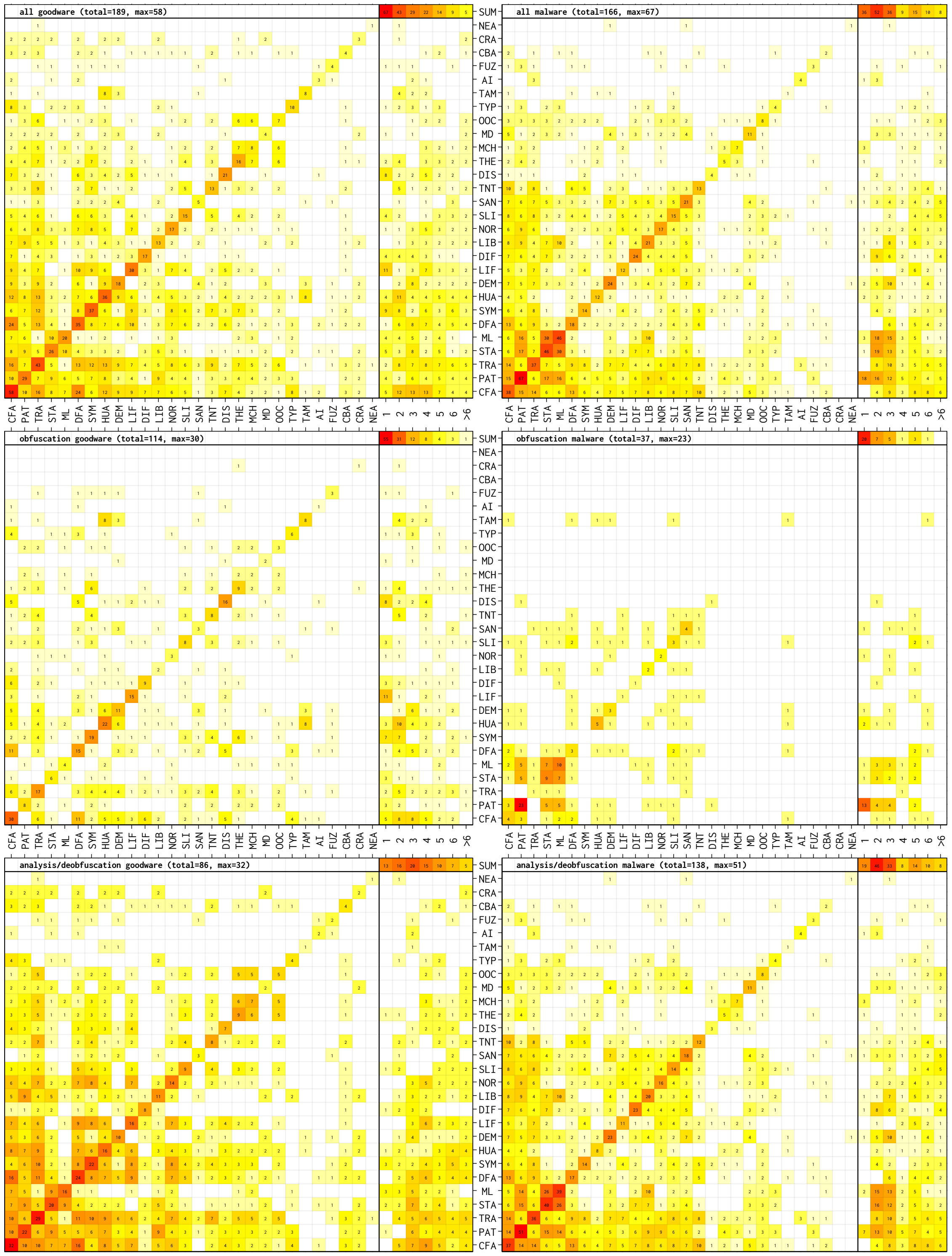


Fig. 8. Six crosstabs visualizing how publications combine analysis methods. The columns on the right indicate the total numbers of papers that combine a particular number of analysis methods (1 to 8 and more). The acronyms are defined in Section 4.

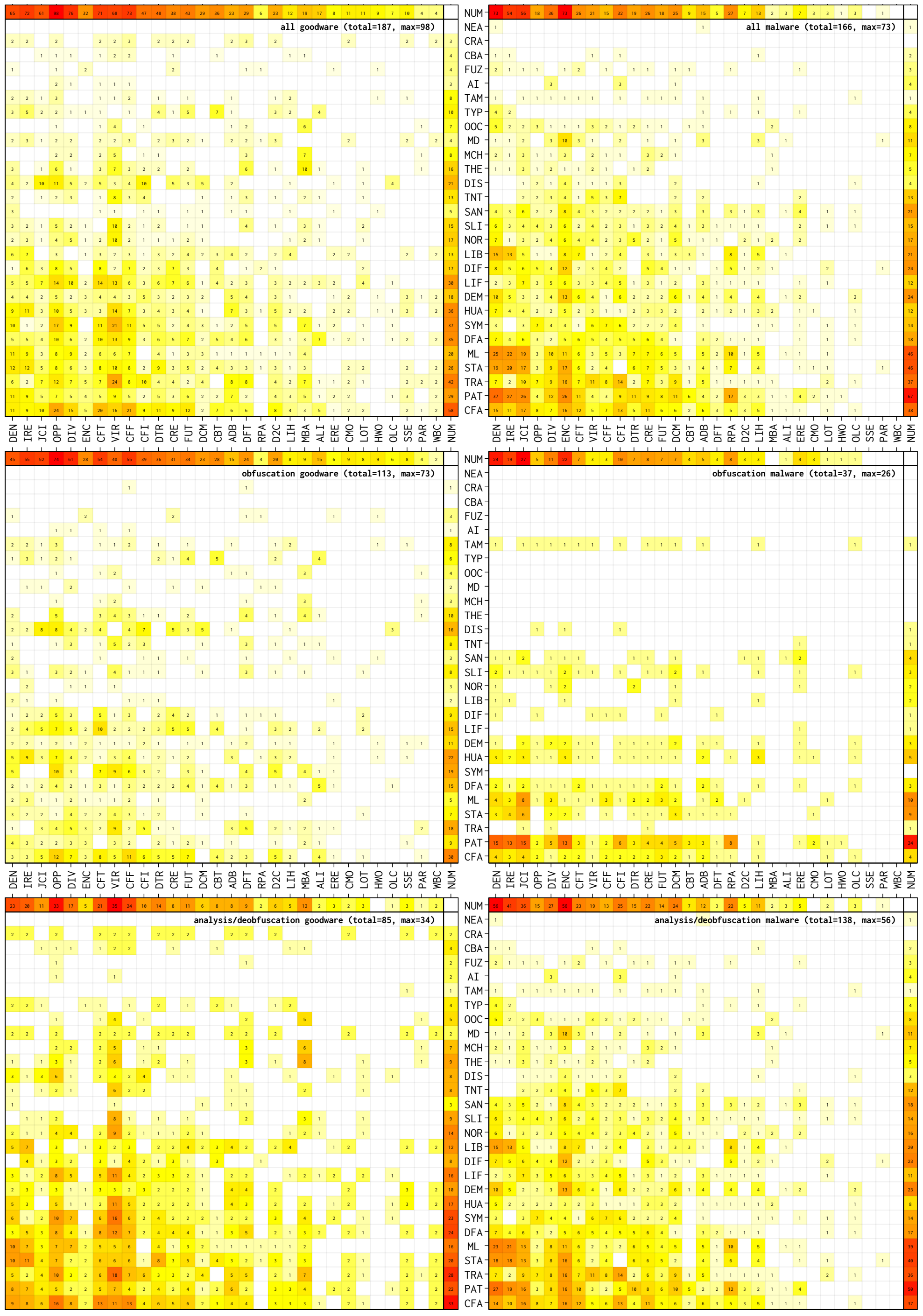
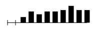
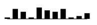
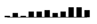
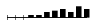
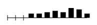
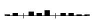
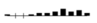
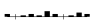
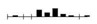
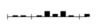
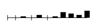
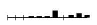
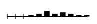
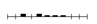
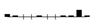
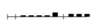
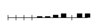
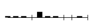
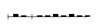
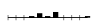
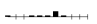
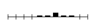
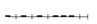
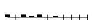
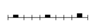
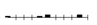
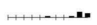
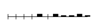
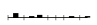
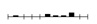
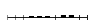
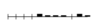
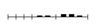
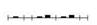
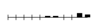
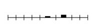
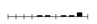
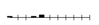
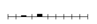


Fig. 9. Crosstab analysis vs. protections. These crosstabs show how many papers deploy the listed analysis techniques (rows) on samples protected with the listed protections (columns) for evaluating them. The acronyms are defined in Section 4 and Section 3.

Table 1. The 39 tools used by more than five surveyed papers in their experimental evaluation. The columns A = Analysis, B = Build, and P = Protection list how many papers use a tool for which purposes. 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. The symbols in column O describe the access to the source-code and mean ● = open-source, ◔ = upon request, ○ = closed-source. The F column describes how flexible/adjustable the tool is, meaning ● = adaptable/extendable, ◑ = plug-ins, ◔ = configurable, ○ = rigid. Column History = Usage per year since 2012 (left to right).

Tool	History	$	O	F	B	A	P	T
LLVM GW:[2, 3, 4, 6, 13, 14, 15, 17, 35, 69, 71, 76, 77, 85, 104, 111, 127, 130, 134, 136, 147, 152, 161, 162, 165, 167, 168, 178, 195, 199, 200, 207, 238, 241, 247, 282, 296, 297, 344, 345, 346, 350, 354, 364, 367, 368, 369, 371, 405, 410, 412, 426, 456, 468, 504, 506, 509, 530, 535, 555, 571, 572, 574, 594] — MW:[20, 100, 149, 185, 370, 388, 448, 472, 479]		●	●	●	71	22	57	73
IDA Pro GW:[3, 35, 69, 77, 103, 105, 112, 119, 138, 165, 167, 168, 171, 176, 177, 204, 221, 222, 223, 224, 277, 278, 279, 282, 292, 311, 312, 372, 379, 386, 397, 405, 415, 426, 456, 492, 505, 546, 548, 555, 564, 566, 568, 569, 570, 572] — MW:[20, 54, 100, 140, 144, 341, 363, 392, 403, 453, 458, 498, 513, 515, 544, 576, 593]		◑	○	◑	—	63	—	63
GCC GW:[2, 3, 4, 6, 29, 30, 36, 69, 75, 78, 87, 103, 112, 114, 127, 128, 130, 134, 138, 145, 159, 161, 165, 171, 175, 199, 200, 203, 221, 224, 238, 240, 247, 249, 277, 284, 320, 324, 364, 376, 378, 379, 393, 406, 415, 509, 541, 547, 596] — MW:[149, 185, 192]		●	●	●	52	3	1	52
Tigress GW:[4, 13, 14, 15, 29, 33, 36, 66, 69, 77, 85, 101, 103, 127, 128, 134, 136, 145, 147, 161, 162, 163, 165, 167, 168, 199, 200, 350, 359, 378, 503] — MW:[20, 115, 149, 237, 479, 536]		●	◔	◑	—	—	37	37
OLLVM GW:[13, 35, 69, 71, 76, 85, 111, 130, 136, 161, 162, 165, 167, 168, 195, 199, 200, 354, 364, 367, 368, 369, 371, 412, 530, 572, 574] — MW:[20, 100, 149, 185, 388, 448, 472, 479]		●	●	●	—	—	35	35
Diablo GW:[2, 3, 8, 21, 47, 48, 119, 130, 141, 171, 203, 204, 247, 249, 277, 278, 288, 324, 326, 431, 432, 461, 540, 541]		●	●	●	—	3	24	24
VirusTotal (AV) MW:[42, 49, 92, 102, 507, 208, 219, 269, 323, 413, 414, 418, 424, 438, 442, 459, 467, 482, 485, 507, 537, 544, 545, 598, 599]		●	○	○	—	24	—	24
PIN GW:[103, 147, 153, 240, 324, 359, 408, 410, 433, 487] — MW:[20, 41, 65, 83, 189, 280, 339, 387, 533, 581]		●	○	●	—	20	—	20
ProGuard GW:[166, 181, 246, 314, 396, 477, 478, 497, 512, 554, 592] — MW:[26, 70, 92, 109, 356, 450, 527, 600]		●	○	●	—	—	19	19
Soot GW:[67, 166, 179, 180, 193, 231, 300, 316, 317, 318, 337, 353, 460] — MW:[26, 49, 322, 385, 450, 591]		●	●	●	—	14	6	19
Clang GW:[2, 4, 6, 13, 77, 85, 134, 152, 161, 241, 282, 296, 412, 456, 509, 572] — MW:[149, 370]		●	●	●	17	—	1	18
VMProtect GW:[60, 103, 125, 147, 153, 359, 408, 410, 415, 492, 526, 534] — MW:[115, 420, 513, 533, 573, 581]		◑	○	○	—	—	18	18
KLEE GW:[4, 13, 14, 15, 35, 77, 104, 127, 134, 238, 350, 405, 426, 509, 535, 555]		●	●	●	—	16	—	16
Eclipse GW:[21, 44, 45, 46, 93, 141, 142, 150, 197, 198, 202, 257] — MW:[401, 402, 403]		●	●	●	2	8	6	15
Visual Studio GW:[27, 28, 103, 175, 198, 222, 433, 486, 570] — MW:[41, 51, 209, 274, 502, 537]		●	○	●	14	4	1	15
Themida GW:[125, 147, 153, 359, 408, 410, 487, 526] — MW:[51, 323, 363, 513, 573, 581]		◑	○	○	—	—	14	14
angr GW:[4, 13, 29, 36, 127, 134, 153, 159, 393, 509] — MW:[100, 149, 485]		●	●	●	—	13	—	13
OllyDbg GW:[103, 397, 408, 569, 570] — MW:[140, 144, 295, 341, 381, 462, 465, 544]		●	○	◑	—	13	—	13
APKTool GW:[164, 193, 201, 354, 396] — MW:[50, 217, 218, 298, 467, 489, 525]		●	●	●	8	4	—	12
Code Virtualizer GW:[60, 125, 130, 153, 177, 359, 415, 534] — MW:[513, 533, 573, 581]		◑	○	○	—	—	12	12
Allatori GW:[43, 160, 181, 250, 380, 497, 554] — MW:[70, 92, 450]		◑	○	○	—	—	10	10
DashO GW:[181, 380, 428, 434, 449, 554] — MW:[70, 92, 208, 450]		◑	○	○	—	—	10	10
Jad GW:[44, 93, 231, 250, 337, 342, 353, 380, 449, 460]		●	●	●	—	10	—	10
Sandmark GW:[43, 44, 46, 93, 202, 250, 347, 449, 497, 528]		●	◑	●	—	—	10	10
UPX GW:[433] — MW:[54, 140, 144, 275, 280, 363, 479, 544, 576]		●	●	●	—	—	10	10
objdump GW:[105, 112, 138, 145, 279, 350] — MW:[189, 388, 485]		●	●	●	—	9	—	9
Syntia GW:[33, 66, 114, 128, 143, 147, 153, 199] — MW:[115]		●	●	●	—	9	—	9
Triton GW:[13, 17, 66, 103, 134, 145, 147, 159, 503]		●	●	●	—	9	—	9
BinDiff GW:[35, 177, 204, 277, 278, 386, 548, 572]		●	○	○	—	8	—	8
Dex2Jar GW:[25, 164, 396, 476, 512] — MW:[281, 334, 385]		●	●	●	—	8	—	8
TXL GW:[2, 21, 47, 48, 142, 203, 247, 258]		●	○	○	—	—	8	8
Z3 GW:[77, 103, 128, 147, 238, 503] — MW:[65, 341]		●	●	●	—	8	1	8
ACTC GW:[2, 21, 47, 48, 142, 171, 203]		●	●	●	7	—	7	7
AndroGuard GW:[194, 354, 396] — MW:[265, 525, 591, 599]		●	●	●	—	7	—	7
Arybo GW:[103, 114, 143, 147, 238, 503] — MW:[115]		●	●	●	—	7	—	7
McAfee (AV) MW:[49, 54, 219, 268, 414, 453, 474]		●	○	○	—	7	—	7
scikit-learn GW:[131, 145] — MW:[50, 132, 452, 479, 527]		●	●	●	—	7	—	7
Zelix Klassmaster GW:[43, 250, 256, 380, 428, 460] — MW:[109]		◑	○	○	—	—	7	7
QEMU GW:[3] — MW:[54, 370, 395, 465, 513]		●	●	●	—	6	—	6

Visual Studio[4] is a proprietary IDE developed by Microsoft. It is used primarily for building applications but also as a debugger for dynamic analysis and, in one case, for implementing a protection scheme.

10.2 Obfuscation tools

Tigress[5] is a state-of-the-art academic SP tool for the C programming language. It is developed by C. Collberg from the University of Arizona, and by far the most popular tool for obfuscation research. Its non-commercial use is free, and

[4]https://visualstudio.microsoft.com/
[5]https://tigress.wtf/

source code is available to academics upon demand. Tigress is capable of a wide variation of transformations (such as virtualization, control flow flattening, opaque predicates, MBA expressions, self-modifying code, etc.) which it can compose in a layered fashion. While Tigress approaches ten years of age, it is still under active development.

Recommendation: Tigress is a complex tool that requires thoughtful decision-making, and hence a considerable effort, to select which protections to deploy, on which program fragments, and with which configurations. Make sure to describe your choices and provide convincing arguments for them. Importantly, over time, the default configuration options for Tigress have evolved, implying that the defaults that will be mentioned in the future on the Tigress website for its latest version might no longer reflect the default options of the version you used at the time of your research. So, make sure to mention what the default options are if you rely on them in your research.

OLLVM[6] or Obfuscator-LLVM is an open-source, academic software protection tool built upon the LLVM compilation suite [371]. It supports a number of relatively simple control flow obfuscations and diversifications applied to the compiler's IR. Unlike Tigress, OLLVM has only seen active development for a brief period, namely in the years 2015–2017, and has not been maintained since.

Recommendation: Compared to Tigress, OLLVM is a weak and mostly obsolete protection tool, so using it to generate protected samples is discouraged.

VMProtect[7] is a commercial code virtualization tool that transforms the original binary code into bytecode that is interpreted by a VM embedded in the application. The VM code is obfuscated and hardened against code analysis, and on top, this protection is layered with encryption, anti-debugging, and anti-process-virtualization techniques to mitigate reverse engineering.

Recommendation: Compared to Tigress, VMProtect is more focused on a single protection. It still requires configuration, however, to select the level of protection and the code to be protected. So as with Tigress, make sure to do a proper configuration,and report it in your paper.

Code Virtualizer[8] is another commercial code virtualization tool.

Recommendation: Similar to VMProtect, Code Virtualizer is mostly focused on a single protection. It still requires configuration, however, so spend the necessary effort to select configurations, and report them in your paper.

Themida[9] is a commercial SP tool that includes a wide range of techniques, including obfuscation, tamperproofing, and preventive techniques such as dynamic encryption, anti-debugging, metamorphic code, code virtualization, and API-wrapping.

Recommendation: Compared to VMProtect and Codevirtualizer, Themida offers more configuration options, and hence requires more configuration effort, so spend the necessary effort to select configurations, and report them in your paper.

UPX[10] is a simple, free executable packer.

Zelix Klassmaster[11] is a commercial Java obfuscator. Its features include identifier renaming, string encryption, control flow obfuscation, integer constant encryption, and type obfuscation.

[6]https://github.com/obfuscator-llvm/obfuscator
[7]https://vmpsoft.com/
[8]https://www.oreans.com/codevirtualizer.php
[9]https://www.oreans.com/themida.php
[10]https://upx.github.io/
[11]https://zelix.com/klassmaster/index.html

Recommendation: As with other configurable commercial SP tools, you need to spend the necessary effort to select configurations, and report them in your paper.

Allatori[12] is a commercial Java obfuscation tool capable of, among others, identifier renaming, control flow obfuscation, string encryption, and debug-info obfuscation, anti-decompilation techniques, and watermarking.

Recommendation: As with other configurable commercial SP tools, you need to spend the necessary effort to select configurations and report them in your paper.

DashO[13] is another commercial Java and Android software protection tool that offers, among others, identifier renaming, control flow obfuscation, string encryption, watermarking, tamperproofing, anti-debugging, anti-emulation, and anti-hook protections.

Recommendation: As with other configurable commercial SP tools, you need to spend the necessary effort to select configurations, and report them in your paper.

Sandmark[14] was an academic SP framework/tool developed at the University of Arizona to study software watermarking, tamper-proofing, and code obfuscation of Java bytecode, including decision support for deploying the supported protections [272]. It has not been maintained or developed since August 2004.

Recommendation: Being unmaintained for 20 years now, Sandmark should no longer be used for SP research.

ProGuard[15] is GuardSquare's proprietary but free tool for Android mobile app developers. ProGuard is mainly aimed at shrinking Java and Kotlin apps and obfuscates them as a side-effect. The resulting protection is much weaker than what is available in tools that specifically target software protection, such as GuardSquare's non-free DexGuard tool. While ProGuard is one of the few popular proprietary obfuscators in SP research, its representativeness of real-world obfuscation can hence be questioned.

Recommendation: Given its lack of advanced SPs, the use of ProGuard for state-of-the-art SP research is questionable.

Diablo[16] is a proof-of-concept, open-source link-time binary rewriting framework developed at Ghent University. It has been used in research to deploy protections on binary code, including as the binary-level protection tool in the already aforementioned ACTC. Furthermore, it has been used for program analysis, such as for implementing program instrumentation in the style of the PIN tool listed below. Contrary to PIN, Diablo-based instrumentation is static. In 2020, the active maintenance and development of Diablo stopped.

Recommendation: Diablo, only supported long outdated versions of Android, GCC, LLVM, GNU binutils, and the GNU C library. Furthermore, it lacks full support for today's most used instruction set architectures such as 64-bit Intel, AMD, and ARM architectures. We therefore advise against using it for SP research.

ACTC[17] or the ASPIRE Compiler Tool Chain was a proof-of-concept, open-source toolchain for protecting native ARM Android libraries and native Linux libraries/binaries. It comprised several source-to-source rewriting stages as well as several binary-rewriting stages for layering a range of SPs that implement an SP reference architecture [294], with

[12]https://allatori.com/
[13]https://www.preemptive.com/products/dasho/
[14]http://sandmark.cs.arizona.edu/
[15]https://github.com/Guardsquare/proguard
[16]https://diablo.elis.ugent.be/
[17]https://github.com/aspire-fp7/actc

GCC and LLVM compilers used in between for compiling the software. The ACTC supported a range of control flow and data flow obfuscations, remote attestation, code mobility [247], code renewability [2], as well as anti-debugging [3, 203].

Recommendation: As the ACTC builds on Diablo, we advise against using it for SP research, for the same reason.

TXL[18] is a domain-specific language and toolset for implementing source-to-source rewriters. It is leveraged in the aforementioned ACTC to implement its source-level protections.

Recommendation: Using source-to-source rewriters for deploying SPs is fine. However, in case your SPs target natively compiled languages such as C or C++, there is the caveat that evaluating the SPs' strength should not rely solely on source-level measurements. Instead, the evaluation needs to include measurements and experiments on properly built binaries (see our recommendations in Section 10.1 on compilers). Critically, check and report how the transformations applied on the source code are reflected in changed binary code. Optimizing compilers can radically transform code, including by eliminating code that is functionally redundant, so it is paramount to validate that the source-level transformations survived the optimizing compiler as intended.

The crosstab in Figure 10 shows how the deployment of protection tools and the deployment of different types of protections overlaps in the surveyed papers.

10.3 Analysis/Deobfuscation tools

IDA Pro[19] is the most widely used interactive disassembler for natively compiled code. This proprietary tool from Hex-Rays also serves as a debugger, thus integrating static and dynamic analysis. The user can override IDA Pro's decisions and provide hints to improve the disassembly results. The user can also inspect, query, and edit the disassembled code through a range of views and interfaces. IDA Pro's functionalities can be extended with plug-ins and scripts. Some popular types of plug-ins and tools that build on IDA Pro are differs (e.g., BinDiff), decompilers, and library function identification tools such as F.L.I.R.T.

Recommendation: IDA Pro is extensible and it offers an API to manipulate the data it extracted from a binary, such as the functions it reconstructed, their control flow graphs, and other data. Exploit these features in the way a real reverse engineer would do, such that your evaluation by means of IDA Pro is representative. Hex-Rays' yearly plug-in contest[20] and their plug-in repository[21] can provide inspiration for ways to exploit IDA Pro's capabilities, and so do some research papers that included plug-ins in their evaluations [20, 69, 103, 140, 167, 168, 171, 372, 492, 498, 515].

In addition, it is critical to be aware that the heuristics that IDA Pro implements to disassemble a program and to model the program in terms of its components are unique to IDA Pro. Radically different approaches exist, however, so relying on only IDA Pro cannot give a complete answer to how disassemblers are impacted by new (anti-disassembly) obfuscations. For example, IDA Pro considers each code byte to be part of, at most, one instruction and each basic block to be part of only one function. Binary Ninja differs in those two aspects, as a result of which it can deal with overlapping instructions much better and handles bogus interprocedural control flow very differently from IDA Pro [171]. In short, be careful when only relying on IDA Pro; instead consider also using Binary Ninja, Ghidra, Radare2, and other comparable tools.

[18]https://www.txl.ca/
[19]https://hex-rays.com/IDA-pro/
[20]http:hex-rays.com/contest
[21]https://plugins.hex-rays.com/

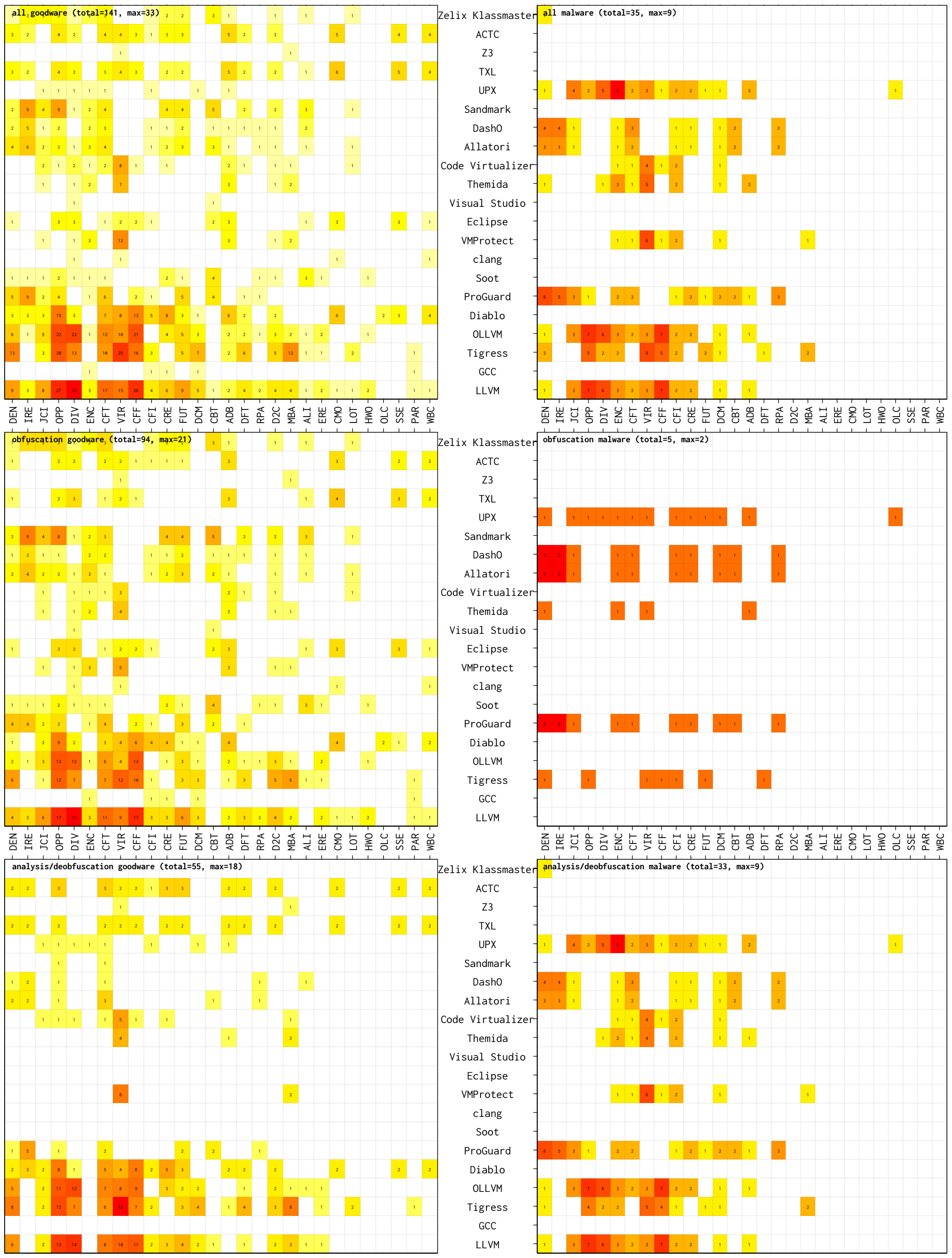


Fig. 10. Crosstab Tools and Protection Techniques. For papers that evaluate samples protected with a specific tool, this crosstab shows which protections are deployed in those papers, i.e., how many of those papers deploy the different protections.

For additional recommendations regarding the evaluation of anti-disassembly obfuscations and disassemblers, we refer to Section 4.1 of the report of the 2019 Dagstuhl Seminar on Software Protection Decision Support and Evaluation Methodologies [68].

PIN[22] by Intel is a dynamic binary instrumentation framework for the the IA-32 and x86-64 instruction-set architectures that enables the creation of dynamic program analysis tools. Its main use is the generation of execution profiles and traces. PIN has been used in hundreds of research projects in the domains of computer architecture, programming language design and implementation, and software engineering.

Recommendation: For dynamic analysis techniques, in particular trace-based techniques, it is important to evaluate (experimentally or with an informed argument) their scalability to sufficiently complex dynamic behaviors, such as traces in which the boundaries between relevant and irrelevant trace fragments have been obfuscated, or traces in which dependencies between instructions have been obfuscated with anti-taint techniques.

For additional recommendations regarding the evaluation of trace-based analysis techniques, we refer to Section 4.2 of the report of the 2019 Dagstuhl Seminar on Software Protection Decision Support and Evaluation Methodologies [68].

KLEE[23] is an open-source dynamic symbolic execution engine that is built on top of the LLVM compiler infrastructure and operates on LLVM bitcode. KLEE has been used in hundreds of research projects on software testing. It is also used for deobfuscation, e.g., to identify unreachable bogus code.

Recommendation: When using KLEE on IR code generated directly from source code with LLVM, e.g., to evaluate how an obfuscation impacts symbolic execution, either provide convincing arguments as to why such experiments are representative for how attackers could use symbolic execution starting from binaries instead of source code, or complement KLEE with native code symbolic execution engines, such as BINSEC/SE [65] or angr [154]. Alternatively, deploy KLEE on IR code obtained by decompiling or lifting binary code [77].

Eclipse[24] is an open-source IDE for Java development that is extensible with different kinds of plug-ins. Such plug-ins have been used for the analysis as well as the protection of software, and some authors use Eclipse as an interface to their build tools.

Soot[25] is an open-source static Java analysis and optimization framework. Its functionality to analyze and transform Java bytecode has been used widely in the static software analysis community, in particular to obfuscate and to reverse engineer such bytecode in SP research. Since December 2022, Soot has been officially succeeded by SootUp.

angr[26] is an open-source binary analysis framework written in Python [154]. Its analysis capabilities include static disassembly, lifting, decompilation, and control flow graph recovery, as well as dynamic symbolic execution, a.k.a. concolic execution.

OllyDbg[27] is a closed-source but free GUI-based Win32 debugger. Its latest release dates from 2013, and offers no functional 64-bit support. OllyDbg is, hence, no longer usable in research. It used to be very popular among hackers, in part because of its open architecture. Many third-party plug-ins were available, including to defeat anti-debugging

[22]https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html
[23]https://klee.github.io/
[24]https://eclipseide.org/
[25]https://www.sable.mcgill.ca/soot/
[26]https://angr.io/
[27]http://ollydbg.de/

software protections by interposing those protections' queries to the OS and environment for evidence of ongoing debugging.

Z3[28] is an open-source theorem prover from Microsoft Research. In the surveyed papers, Z3 is mainly used for solving path conditions during symbolic execution but also for checking the equivalence of obfuscated and deobfuscated code, such as MBA expressions.

objdump[29] is part of the GNU Binutils, a collection of tools to inspect and manipulate binaries. It allows for the inspection of object files and executables, their header data as well as their code and data content, and includes a simple (linear-sweep) disassembler.

Syntia[30] is an academic deobfuscation tool [33], which is reused as a baseline in several other papers that propose alternative or improved deobfuscation techniques, and as an evaluation of novel or improved obfuscation techniques.
Recommendation: While Syntia is the most commonly evaluated black-box deobfuscation approach in the surveyed papers, better-performing alternatives are now available, such as Xyntia [128], that can hence replace Syntia as a baseline for comparison.

Dex2Jar[31] is a tool to inspect Android Dalvik Executable (DEX) files and to convert them to related bytecode file formats such as Java .class files or Smali, an IR for DEX files. It is used as a basic component in a number of non-trivial analyses/deobfuscation methods approaches, but also as a standalone tool to evaluate the effectiveness of preventive obfuscations that target this type of tool.

BinDiff[32] is a binary differ that builds on IDA Pro. This used to be a proprietary closed-source tool, but as of September 2023, it has been released as open-source code. Obviously, all uses collected in our survey predate that release. BinDiff is mainly used for evaluating obfuscations, but also in the reverse engineering case study of DropBox [386].

APKTool[33] is a tool for extracting resources from Android apps in the form of APK files, for rebuilding APK files after manipulation of the resources, and for disassembling the code resources into the Smali IR.

Jad[34] is a now obsolete Java bytecode decompiler that has mainly been used to study the effect of Java obfuscation on decompilation.

AndroGuard[35] is a Python tool for analyzing, disassembling, decompiling, and debugging Android DEX files.

QEMU[36] is an open-source machine emulator and virtualizer that is popular for dynamic analysis.
Recommendation: Given QEMU's similarity to PIN, the same recommendations apply regarding scalability to complex enough programs.

scikit-learn[37] is a collection of open-source Python tools for machine learning that has some popularity in malware detection research.

[28]https://github.com/Z3Prover/z3
[29]https://www.gnu.org/software/binutils/
[30]https://github.com/rub-sysSec/syntia
[31]https://github.com/pxb1988/dex2jar
[32]https://www.zynamics.com/bindiff.html
[33]https://apktool.org/
[34]http://www.javadecompilers.com/jad
[35]https://github.com/androguard/androguard
[36]https://www.qemu.org/
[37]https://scikit-learn.org/

Arybo[38] is a tool for manipulation, canonicalization, and identification of MBA expressions. Given a complex expression, it can return the bit-level symbolic representation thereof. It has been used for reverse engineering MBA expressions and VM handlers.

Triton[39] is a Python library for building dynamic binary analysis tools. It can be used for dynamic symbolic execution, dynamic taint analysis, lifting code to LLVM IR, and more.

Recommendation: Given that Triton can be used, among others, for similar dynamic analyses as PIN and QEMU, the same recommendations apply regarding scalability to complex enough programs.

10.4 Anti-virus tools

McAfee AV[40] is a range of anti-virus and digital security solutions. In the surveyed papers, it is mainly used to evaluate how (deobfuscating) transformations applied on samples before feeding them to malware detection tools can improve those tools' outcomes.

VirusTotal[41] is a platform for analyzing files to detect malware.

The crosstabs in Figure 11 and Figure 12 show how the deployment of analysis tools, deobfuscation tools, and anti-virus tools in the surveyed papers overlaps with the types of measurements and the types of analysis results being reported in those papers.

11 Experiments with Human Subjects

Table 2 extends the information already presented in Table 6 of the main paper with details regarding the specific protections that were composed and layered in the samples handled by humans in the reported experiments. It is clear that few experiments included samples in which many protections were composed and layered.

We want to point out that the three largest combinations and layerings of protections listed for the two papers by Ceccato et al. [47, 48] were selected by industrial SP experts for protecting the assets in use cases representative for their business. As such, these combinations can serve as inspiration for researchers in search for relevant combinations of protections. Another inspiration can be found on the Tigress website, which lists so-called protection recipes[42].

12 Overview and Description of Related Work

Table 3 lists related surveys in specific topics in the SP domain in chronological order by year.

[38]https://github.com/quarkslab/arybo
[39]https://triton-library.github.io/
[40]https://www.mcafee.com/
[41]https://www.virustotal.com/
[42]https://tigress.wtf/recipes.html

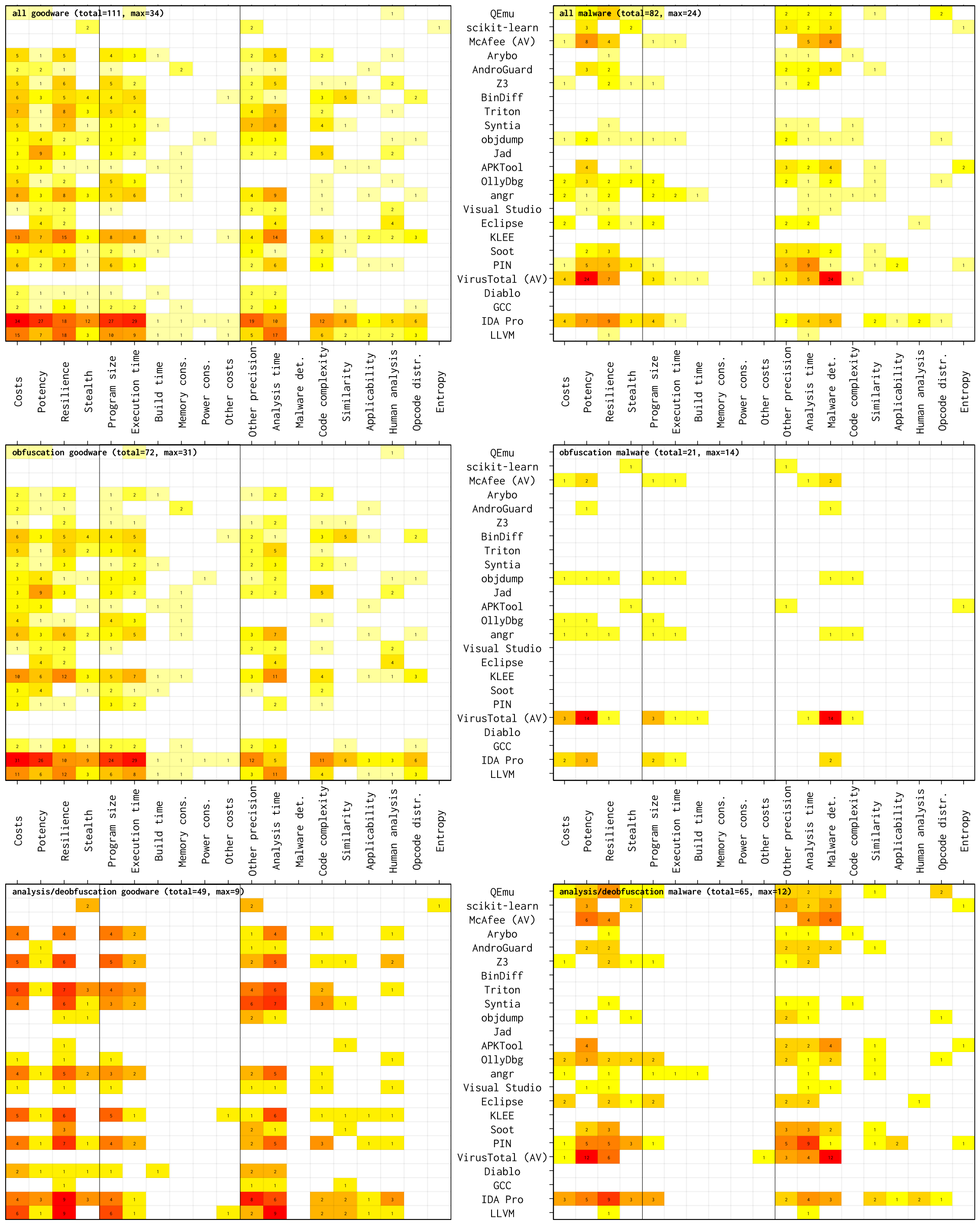


Fig. 11. Crosstab Tools and Measurements. For papers that evaluate samples by analysing them with a specific analysis tool, this crosstab shows which measurements are reported in those papers, i.e., how many of those papers report the different measurements.

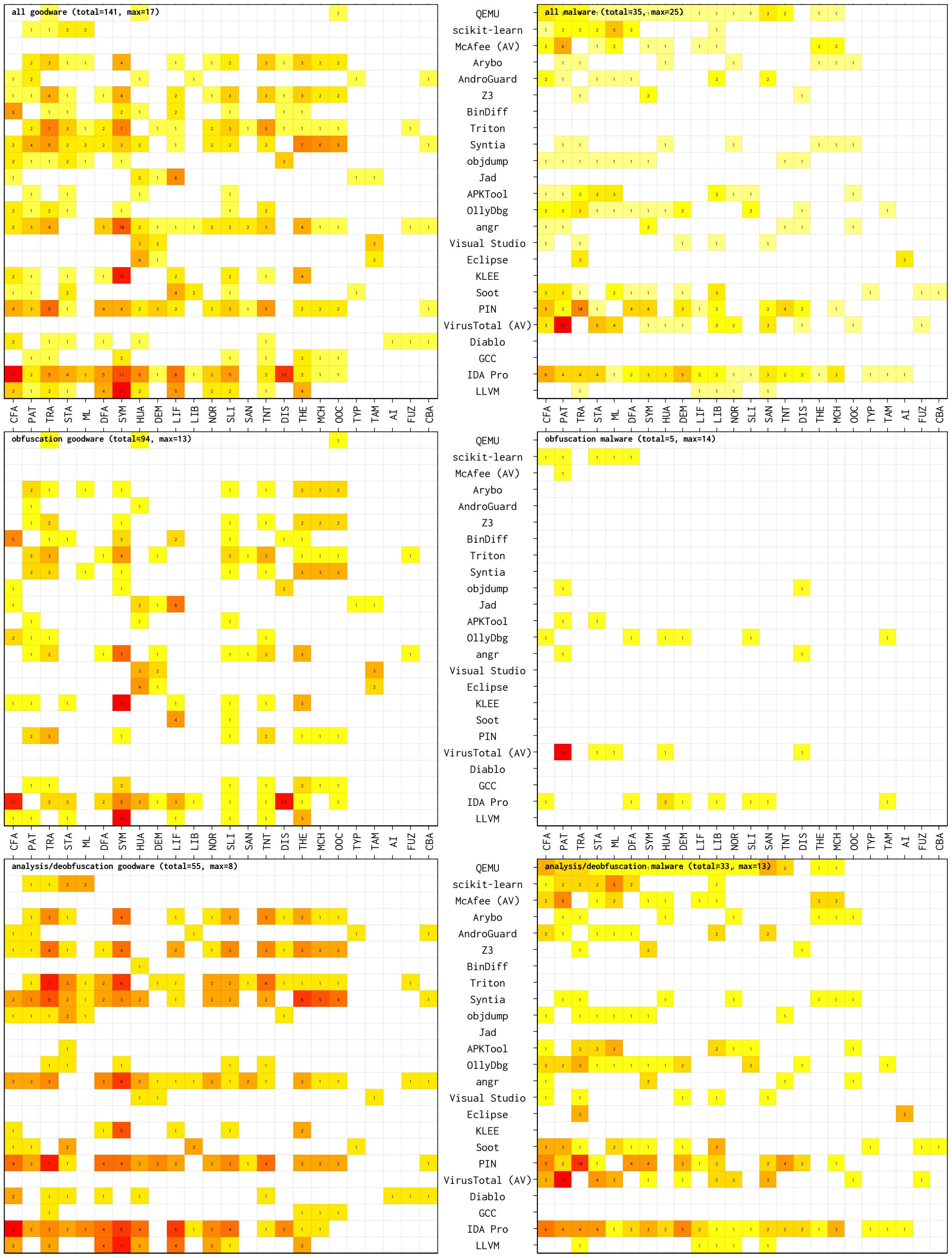


Fig. 12. Crosstab Tools and Analysis Methods. For papers that evaluate samples by analysing them with a specific analysis tool, this crosstab shows which analyses are reported in those papers, i.e., how many of those papers report results for the different types of analysis.

Table 2. Papers that report on (controlled) experiments involving human subjects performing MATE protection and attack tasks. Subjects indicates how many subjects participated of different level of expertise, ranging from ○ bachelor and master students, over ◔ PhD students 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, to ● professional security experts and pen testers. The samples column indicates what type of samples were handled by the human participants. The asterisks mark samples that are real-world programs rather than being just a "Complex" program that is somewhere between a toy program and a real-world program. The combinations column indicates which protections are deployed in the different samples, in which combinations, using abbreviations defined in Section 3. Commas separate different samples, and + indicates that protections are composed in samples. "Vanilla" denotes unprotected samples. The "Lang" column indicates what type of programming language was targeted: (N)ative, (M)anaged, or (S)cript. The format column indicates the format in which the software was reverse engineered: (S)ource, (N)ative, or (I)ntermediate. The time column indicates how long the experiments lasted. Question marks indicate the information is not available.

Paper	Year	Subjects	Samples	Combinations of protections deployed on samples	Lang	Format	Time
[199]	2021	5 ◑	Toy	MBA, DIV	N	N	12h
[27]	2021	14 ◕	Toy	Vanilla, CFF+OPP	M	I	?
[175]	2020	87 ○	Complex	Vanilla, SSE	N	S	2h
[91]	2020	22 ◑●○	Complex	Vanilla, ?	M	S	1h
[48]	2019	6 ● 1 ◑	*Mobile, Toy	D2C+WBC+VIR+ADB+OPP+CFF+SSE+CMO+DIV, D2C+WBC+VIR+ADB+OPP+CFF+CMO+DIV, D2C+ADB+OPP+CFF+SSE+CMO+DIV, OPP+CFF+ADB, DEN+ADB, DEN, WBC, VIR	N	N	30d
[28]	2019	14 ◕	Toy	Vanilla, CFF+OPP	M	I	?
[193]	2019	4 ◕	Mobile	DCM	M	I	40h
[93]	2018	2 ◔ 64 ○	Complex	Vanilla, IRE, OPP	M	I	1.5h
[106]	2018	2 ◔ 13 ○	Complex, Toy	VIR+DIV	N	N	72h
[181]	2018	63 ◕	Mobile	?	M	I	?
[116]	2017	10 ◔ 10 ○	Complex	?	S	S	?
[47]	2017	6 ●	*Mobile	D2C+WBC+VIR+ADB+OPP+CFF+SSE+CMO+DIV, D2C+WBC+VIR+ADB+OPP+CFF+CMO+DIV, D2C+ADB+OPP+CFF+SSE+CMO+DIV	N	N	30d
[117]	2016	20 ◕	Complex	?	S	S	?
[125]	2016	1 ◕	Complex, Toy	VIR+ADB+LIH, ENC+ABD, VIR+DEN	N	N	?
[174]	2016	1 ◔ 14 ○	Complex	Vanilla, DTR	N	S	3.5h
[202]	2014	12 ○	Complex	Vanilla, IRE, OPP, IRE+OPP	M	S	1h
[44]	2014	22 ◔ 52 ○	Complex	Vanilla, IRE, OPP	M	I	4h
[140]	2009	6 ◑	Malware	ENC	N	N	?
[46]	2009	22 ◔ 10 ○	Complex	Vanilla, IRE	M	I	4h
[45]	2008	8 ○	Complex	Vanilla, IRE	M	I	4h
[137]	2007	5 ●	*Complex	?+DEN+CFT+FUT+LIH+HWO	N	N	80m

Table 3. Overview of existing surveys in the SP domain. The second column indicates whether the surveys focus on malware (M) or goodware (G) software protection, as well as if it targets protections (P) and/or analyses including deobfuscation (A).

Ref. **Year**	**M/G** **P/A**	**Brief description**
[58] 2002	G P	Collberg and Thomborson gave an overview of SPs and describe obfuscation, watermarking and tamper proofing methods and group argue about their stealth, resilience, obscurity and costs. This work also defines a threat model for each protection.
[12] 2005	M P	Balakrishnan and Schulze categorized SPs into general, obstructing static analysis, targeting the disassembly phase, and obfuscation related to malware.
[124] 2006	P	Majumdar, Thomborson, et al. shortly gave an overview of the dynamic dispatcher model and opaque predicates as provable control-flow obfuscations.
[38] 2010	M P	Brand's PhD thesis and its corresponding publications revised different anti-analysis techniques. While this work heavily focuses on analysis avoidance it also includes obfuscations (control flow and data transformation as well as packing). Brand implements the different anti-analysis techniques into simple programs and tries to detect or mitigate them.
[118] 2010	G P	Long, Liu, et al. opposed refactoring methods and obfuscation methods regarding their objectives, readability, complexity, uniformity and understandability. They also create a model to link the business or functional goals of the designer and attacker with the refactoring and obfuscation methods.
[191] 2010	M P	You and Yim described different types of anti-signature techniques for malware (encryption, oligomorphism, polymorphism, and metamorphism) and obfuscation techniques used by polymorphic and metamorphic malware. The following obfuscation are discussed: dead-code insertion, register reassignment, subroutine reordering, instruction substitution, code transposition, and code integration into the target. The focus is on how the techniques look at a binary level.
[198] 2010	G P	A book chapter by Zhang, He, et al. describes the theory of obfuscation before the authors present and evaluate a call-flow obfuscation and instruction substitution method.
[126] 2011	P	Mavrogiannopoulos, Kisserli, et al. presented a taxonomy of self-modifying code. They categorize the methods in concealment, encoding, visibility, and exposure and assess several tools and methods.
[74] 2012	M A	Egele, Scholte, et al. covered different automated dynamic analysis tools and techniques for malware. The dynamic analysis techniques entail: function call monitoring, function parameter analysis and information flow tracking.
[95] 2012	G P	Hataba and El-Mahdy gauged the strength of obfuscation transformations and argued the usefulness for cloud computing applications. The publication discusses potency, resilience and costs as well as complexity metrics and map them to obfuscation techniques.
[182] 2012	A P	This special issue article by Willems and Freiling briefly summarizes static and dynamic program analysis and reverse engineering countermeasures. In regards to obfuscation the authors mention code optimization, opaque predicates, invalid branches, multi threading, import table obfuscation, runtime packers, polymorphism, and virtualization.

Continues on the next page...

Continued from the previous page...

Ref. Year	M/G P/A	Brief description
[37] 2012	M P	Branco, Barbosa, et al. gave an overview of anti-debugging and anti-disassembly as well as anti-vm used by malware. The identified techniques include the use of push and pop instructions instead of returns, converting static data to procedures, dead and fake code insertion, instruction substitutions, code reordering, register reassignment and library hiding.
[144] 2013	M A P	Roundy and Miller analyzed different malware packers. The authors took several code measurements and grouped those into categories, namely dynamic code, instruction obfuscation, code layout, anti-rewriting, anti-tracing. Measurements in the "instruction obfuscation" class are e.g., percentage of indirect calls, number of indirect jumps or non-standard uses of call and ret instructions.
[43] 2015	G P	A study by Ceccato, Capiluppi, et al. quantified 44 obfuscation methods for Java by applying them on 18 applications and measuring 10 code metrics. The metrics consider the modularity, size and complexity of the code. Their work shows, that some obfuscations substantially alter the potency of the code.
[150] 2016	P	A short review by Sebastian, Malgaonkar, et al. described the areas of application, possible obfuscation measurements, available techniques and drawbacks of code obfuscation. They classify different obfuscation techniques into layout transformations, control flow transformations, data abstraction and procedural abstractions.
[97] 2016	P	Hosseinzadeh, Rauti, et al. conducted a survey to compare the different programming languages, aims and environments of publications in the areas of diversification and obfuscation. The systematic search and selection process yielded 209 publications. One of their results is a categorization of the programming language (e.g., Java, C, Machine code, JavaScript, SQL,...) the publications apply the protection techniques to. Hosseinzadeh, Rauti, et al. mapped the languages into language classes (e.g., native code, domain specific languages, scripts,...). Their identified aims of the publications in the field of software protection cover: prevent reverse engineering, resist tampering, hide data and prevent exploitation. The targeted environments (e.g., server, cloud, desktop, mobile, any) are mostly independent according to the two biggest categories in their results being any and any native environments.
[19] 2016	P	The review article "Hopes, Fears, and Software Obfuscation" surveys research on indistinguishability obfuscation and its potential impact.
[148] 2016	A P	Schrittwieser, Katzenbeisser, et al. described the arms race between SPs and code analysis. The authors used analysis goals (locating data, locating code, extracting data, extracting code, understanding code, and simplifying code) and methods (pattern matching, automatic static, automatic dynamic, automatic symbolic, and manual analysis) to classify the field of analysis.
[63] 2016	M P	Dalla Preda and Maggi tested the effectiveness and influence of obfuscation techniques on the detection rate of anti-malware products for Android apps. Their results indicate malware detectors are not able to recognize obfuscated malware versions.

Continues on the next page...

Continued from the previous page...

Ref. Year	M/G P/A	Brief description
[55] 2016		Collberg and Proebsting published a large-scale study on repeatability in the general field of computer science. For more than 600 publications they tried to get access to the code for the experiments from the authors and to reproduce the experiments. For only 37% of the publications the code for the experiments could be obtained, and only just over half of them were executable (20% of the total number of papers). Their lack of success of obtaining the results can also be seen in our data from little sample reuse.
[16] 2016	P	In 2017 Banescu and Pretschner classified obfuscations against MATE attacks based on abstraction level (source, intermediate, binary machine code), unit (instruction, basic block, function, program, system level), dynamics (static, dynamic) and target (data, code). MATE attacks based on the attack type (code understanding, data item recovery, location recovery), dynamics (static, dynamic analysis), interpretation (syntactic, semantic attacks) and alteration (passive, active attacks).
[188] 2017	P	Xu, Zhou, et al. surveyed the field of SP and split it into code (source, binary) vs. model (circuits, Turing machines) oriented obfuscation.
[92] 2018	M A	In 2018 Hammad, Garcia, et al. compared different obfuscation tools for Android. They run the obfuscation tool on benign and malicious apps and afterwards check if apps are installable and runnable. Furthermore the obfuscated Malware is run through anti-malware products to check their detection rate. The authors group the obfuscation techniques into trivial (disassembly/reassembly, modifying the AndroidManifest, Moving the uncompressed data, repackaging) and non-trivial (junk code insertion, class renaming, member reordering, string encryption, identifier renaming, control flow manipulation, reflection).
[181] 2018	G A	Wermke, Huaman, et al. measured the usage of obfuscation used by Android apps in the Google Play store and visualized the details about obfuscation depending on last updated, number of downloads, number of apps per account. The second part of their investigation targeted the developer. Of the around 300 participants 78% had heard of obfuscation, while only 48% had used protections. 78% of a subgroup of 70 people were not able to protect an app using ProGuard.
[108] 2018	G P	Kumar and Kurian studied control flow obfuscation techniques in Java. They looked at the control flow obfuscation techniques for java used in papers and tools, tested 13 obfuscators on 16 programs and manually evaluated the outcome. They identified 36 unique techniques in literature and seven from tools. The different methods were put into classes based on the transformed component: expression, statement, basic block, method, class. Their studies indicate a gap between theory and practice as only 13 of the 36 are implemented by tools.
[113] 2018	M A	Liţă, Cosovan, et al. shared their knowledge on malware packers resulting from monitoring the development of packers for almost two years. The authors described three packers used by malware families like Upatre, Gamarue, Hedsen and explained the current trends in malware packers.
[98] 2018	P	The systematic literature review by Hosseinzadeh, Rauti, et al. from 2018 analyzed 357 publications from 1993–2017. The authors surveyed metadata, environment and aims of publications in the areas of diversification and obfuscation. Analyzed metadata includes involved countries and universities, publication type (journal article, conference paper,...) and the associated organizations' sector (academic, industrial, both, unknown).

Continues on the next page...

Continued from the previous page...

Ref. Year	M/G P/A	Brief description
[176] 2018	G A	Wang, Bao, et al. analyzed the usage of obfuscation in the iOS ecosystem. They developed a system to compute the likelihood of an app being obfuscated and evaluated it on over a million apps. The system identifies symbol renaming, string encoding, decompilation disruption and control flow flattening by means of a natural language processing model, CFG analysis and the number of IDA Pro failures. Their results suggest an upwards trend for obfuscation and a correlation between the likelihood of an app being obfuscated and the type of application (finance, privacy intellectual properties and monetization are more likely).
[29] 2018	A	The tutorial by Biondi, Given-Wilson, et al. describes static and dynamic malware detection techniques and tools such as PEiD, DIE, YARA, Cuckoo, and angr. The authors show the basic usage for each tool and describe their limitations in the presence of SPs.
[70] 2018	M G A	In 2018 a huge study by Dong, Li, et al. investigated the dispersiveness of obfuscation in over 110.000 Android apps. The authors used machine learning on apps from Google Play and six 3rd-party markets as well as VirusShare and VirusTotal to detect identifier renaming, encryption, reflection and packing. The machine learning approach uses the F-Droid Android app repository for the learning phase.
[5] 2019	P	Ahmadvand, Pretschner, et al. developed a taxonomy for the field of software protections. The taxonomy describes the view from a view of the defender, the system and the attacker and portray how these views interact with each other. The authors applied the taxonomy to around 50 publications.
[82] 2019	M G A	Graux, Lalande, et al. studied the development of obfuscation techniques over time. They used goodware (GOOD dataset) as well as malware (AndroZoo, AMD and Drebin dataset) and compared the different datasets. The observed techniques are packing (encrypting bytecode), native (usage of native code), DCL (dynamic code loading) and reflection (for hiding method and fields).
[151] 2019	A	Semenov, Davydov, et al. described available code complexity metrics for obfuscated code and proposed several improvements. Furthermore, for the different types of metrics the work outlines retrieval algorithms using graphs.
[135] 2019	P	Ollivier, Bardin, et al. explored a subset of SPs against dynamic symbolic execution. For each protection the publication outlines the strength, cost, correctness, stealth, and known mitigations as well as experimental evaluations from previous literature.
[25] 2020	G A	Berlato and Ceccato looked at the state of protections against debugging and tampering in Android apps. The work analyzed 23610 apps, out of which 75.62% employ two to four protections and only 7.49% implemented no protection. The protection present in most apps were signature checking (88.90%), installer verification (74.42%) and emulator detection (49.83%).
[139] 2020	M A	Qiu, Zhang, et al. compared the performance and resilience of Android malware detection and classification approaches against code obfuscation.
[96] 2020	G P	In their book Horváth and Buttyán surveyed and explained concepts and methods for cryptographic obfuscation, such as virtual black box obfuscation, indistinguishability obfuscation.

Continues on the next page...

Continued from the previous page...

Ref. Year	M/G P/A	Brief description
[187] 2020	P	Xu, Zhou, et al. created a taxonomy for layered obfuscation. It splits SPs into four layers: code element, software component, inter-component and application layer. Each layer contains several sub-layers with the individual targets (layout, controls, data, methods, classes, library calls, resources, DRM systems, neural networks).
[9] 2021	M A	Ardagna, Wu, et al. surveyed the features of machine learning approaches to detect malicious Android applications, as well as which malware datasets were used for evaluation and the achieved accuracy. Furthermore, the authors discussed the various obfuscation techniques present in the data sets and how they influence the features.
[94] 2021	M G A	Haq and Caballero compared different approaches for binary code similarity detection. They focused on the platform, targeted architectures, availability, datasets, methodology, and if the approach was tested against obfuscated code.
[196] 2021	A P	Zhang, Breitinger, et al. survey Android obfuscation and deobfuscation tools. They categorize, describe each tool and publication and oppose the obfuscation techniques with the deobfuscation tools and analysis methods.
[73] 2021	M G P	Ebad, Darem, et al. analyzed 67 studies in the field of software obfuscation in a systematic literature review and look at the measurements, evaluation of the obfuscation quality. They reported numbers on the venues with at least two publications (top being Computers & Security with six), the type of study (51 experiments, one case study, 16 simulations) dataset (top two are: 33 in-the-wild, 10 manually written) as well as which obfuscation quality the examined studies measured (20 potency, nine resilience, 21 cost, eight stealth, 36 similarity).
[103] 2021	P	Kochberger, Schrittwieser, et al. classified deobfuscation frameworks for virtualization-protected applications based on the degree of automation, extracted artifacts, and analysis methods. The work also presents experiments with existing tools on a uniform sample set for comparable results.

References

[1] M. AbdelKhalek and A. Shosha. 2017. JSDES: An Automated De-Obfuscation System for Malicious JavaScript. In *ARES*.

[2] B. Abrath, B. Coppens, et al. 2020. Code Renewability for Native Software Protection. *ACM Trans. Priv. Secur.*, 23, 4.

[3] B. Abrath, B. Coppens, et al. 2016. Tightly-coupled self-debugging software protection. In *ACM SSPREW*, 7:1–7:10.

[4] D. Adhikari, J. T. McDonald, et al. 2022. Argon: A Toolbase for Evaluating Software Protection Techniques Against Symbolic Execution Attacks. In *SoutheastCon*, 743–750.

[5] M. Ahmadvand, A. Pretschner, et al. 2019. A taxonomy of software integrity protection techniques. In *ADCOM*. Volume 112, 413–486.

[6] A. Altinay, J. Nash, et al. 2020. BinRec: Dynamic Binary Lifting and Recompilation. In *EuroSys*.

[7] S. Amir, B. Shakya, et al. 2018. Development and Evaluation of Hardware Obfuscation Benchmarks. *J. Hardw. Syst. Secur.*, 2, 2, 142–161.

[8] B. Anckaert, M. Madou, et al. 2007. Program Obfuscation: A Quantitative Approach. In *ACM QoP*, 15–20.

[9] C. A. Ardagna, Q. Wu, et al. 2021. A Survey of Android Malware Static Detection Technology Based on Machine Learning. *Mobile Information Systems*.

[10] E. Avidan and D. G. Feitelson. 2015. From Obfuscation to Comprehension. In *IEEE ICPC*, 178–181.

[11] M. Backes, S. Bugiel, et al. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *ACM CCS*, 356–367.

[12] A. Balakrishnan and C. Schulze. 2005. Code Obfuscation Literature Survey. CS701 Construction of Compilers. (2005).

[13] S. Banescu, C. Collberg, et al. 2016. Code Obfuscation Against Symbolic Execution Attacks. In *ACM ACSAC*, 189–200.

[14] S. Banescu, C. Collberg, et al. 2017. Predicting the Resilience of Obfuscated Code against Symbolic Execution Attacks via Machine Learning. In *USENIX Security*, 661–678.

[15] S. Banescu, M. Ochoa, et al. 2015. A Framework for Measuring Software Obfuscation Resilience against Automated Attacks. In *IEEE/ACM SPRO*, 45–51.

[16] S. Banescu and A. Pretschner. 2017. A Tutorial on Software Obfuscation. In *Advances in Computers*. Volume 108, 283–353.

[17] S. Banescu, S. Valenzuela, et al. 2021. Dynamic Taint Analysis versus Obfuscated Self-Checking. In *ACM ACSAC*, 182–193.

[18] S. Banescu. [n. d.] GitHub — A set of programs used for benchmarking the strength of obfuscation. https://github.com/tum-i4/obfuscation-benchmarks.

[19] B. Barak. 2016. Hopes, Fears, and Software Obfuscation. *Commun. ACM*, 59, 3, 88–96.

[20] S. Bardin, R. David, et al. 2017. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *IEEE S&P*, 633–651.

[21] C. Basile, D. Canavese, et al. 2019. A meta-model for software protections and reverse engineering attacks. *J. Sys. and Softw.*, 150, 3–21.

[22] R. Baumann, M. Protsenko, et al. 2017. Anti-ProGuard: Towards Automated Deobfuscation of Android Apps. In *SHCIS*, 7–12.

[23] P. Beaucamps and É. Filiol. 2007. On the possibility of practically obfuscating programs — Towards a unified perspective of code protection. *J. in Comput. Virol.*, 3, 1, 3–21.

[24] M. Bellare, I. Stepanovs, et al. 2016. New negative results on differing-inputs obfuscation. In *EUROCRYPT*, 792–821.

[25] S. Berlato and M. 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, 102463.

[26] B. Bichsel, V. Raychev, et al. 2016. Statistical Deobfuscation of Android Applications. In *ACM CCS*, 343–355.

[27] M. H. Bin Shamlan, A. S. Alaidaroos, et al. 2021. Experimental Evaluation of the Obfuscation Techniques Against Reverse Engineering. In *ICACIn*, 383–390.

[28] M. H. Bin Shamlan, M. A. Bamatraf, et al. 2019. The Impact of Control Flow Obfuscation Technique on Software Protection Against Human Attacks. In *ICOICE*, 1–5.

[29] F. Biondi, T. Given-Wilson, et al. 2018. Tutorial: An Overview of Malware Detection and Evasion Techniques. In *ISoLA*, 565–586.

[30] S. Blazy and R. Hutin. 2019. Formal Verification of a Program Obfuscation Based on Mixed Boolean-Arithmetic Expressions. In *ACM CPP*, 196–208.

[31] S. Blazy and S. 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.

[32] S. Blazy and A. Trieu. 2016. Formal verification of control-flow graph flattening. In *ACM CPP*, 176–187.

[33] T. Blazytko, M. Contag, et al. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *USENIX Security*, 643–659.

[34] J.-M. Borello and L. Mé. 2008. Code obfuscation techniques for metamorphic viruses. *J. in Comput. Virol.*, 4, 211–220.

[35] P. D. Borisov and Y. V. Kosolapov. 2020. On the Automatic Analysis of the Practical Resistance of Obfuscating Transformations. *Autom. Control Comput. Sci.*, 54, 7, 619–629.

[36] P. Borrello, E. Coppa, et al. 2021. Hiding in the Particles: When Return-Oriented Programming Meets Program Obfuscation. In *IEEE/IFIP DSN*, 555–568.

[37] R. R. Branco, G. N. Barbosa, et al. 2012. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat*.

[38] M. Brand. 2010. *Analysis avoidance techniques of malicious software*. PhD thesis. School of Computer and Security Science.

[39] Technische Universität Braunschweig. [n. d.] The Drebin Dataset. https://www.sec.tu-bs.de/~danarp/drebin/download.html.

[40] P. Brunet, B. Creusillet, et al. 2019. Epona and the Obfuscation Paradox: Transparent for Users and Developers, a Pain for Reversers. In *ACM SPRO*, 41–52.

[41] J. Calvet, J. M. Fernandez, et al. 2012. Aligot: Cryptographic Function Identification in Obfuscated Binary Programs. In *ACM CCS*, 169–182.

[42] G. Canfora, A. D. Sorbo, et al. 2015. Obfuscation Techniques against Signature-Based Detection: A Case Study. In *MST*, 21–26.

[43] M. Ceccato, A. Capiluppi, et al. 2015. A large study on the effect of code obfuscation on the quality of java code. *Empir. Softw. Eng.*, 20, 6, 1486–1524.

[44] M. Ceccato, M. Di Penta, et al. 2014. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empir. Softw. Eng.*, 19, 4, 1040–1074.

[45] M. Ceccato, M. Di Penta, et al. 2008. Towards Experimental Evaluation of Code Obfuscation Techniques. In *ACM QoP*, 39–46.

[46] M. Ceccato, M. Di Penta, et al. 2009. The effectiveness of source code obfuscation: An experimental assessment. In *IEEE ICPC*, 178–187.

[47] M. Ceccato, P. Tonella, et al. 2017. How Professional Hackers Understand Protected Code while Performing Attack Tasks. In *IEEE ICPC*, 154–164.

[48] M. Ceccato, P. Tonella, et al. 2019. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *Empir. Softw. Eng.*, 24, 1, 240–286.

[49] A. Chawdhary, R. Singh, et al. 2017. Partial evaluation of string obfuscations for Java malware detection. *Formal Aspects of Computing*, 29, 1, 33–55.

[50] Y.-C. Chen, H.-Y. Chen, et al. 2021. Impact of Code Deobfuscation and Feature Interaction in Android Malware Detection. *IEEE Access*, 9, 123208–123219.

[51] B. Cheng, J. Ming, et al. 2021. Obfuscation-Resilient Executable Payload Extraction From Packed Malware. In *USENIX Security*, 3451–3468.

[52] X. Cheng, Y. Lin, et al. 2019. DynOpVm: VM-based software obfuscation with dynamic opcode mapping. In *ACNS*, 155–174.

[53] S. Cho, H. Chang, et al. 2008. Implementation of an obfuscation tool for c/c++ source code protection on the xscale architecture. In *IFIP SEUS*, 406–416.

[54] M. Christodorescu, S. Jha, et al. 2007. Software transformations to improve malware detection. *J. in Comput. Virol.*, 3, 253–265.

[55] C. Collberg and T. A. Proebsting. 2016. Repeatability in computer systems research. *Comm. ACM*, 59, 3, 62–69.

[56] C. Collberg, C. Thomborson, et al. 1997. A Taxonomy of Obfuscating Transformations. Technical report 148. University of Auckland.

[57] C. Collberg, C. Thomborson, et al. 1998. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *ACM POPL*, 184–196.

[58] C. Collberg and C. Thomborson. 2002. Watermarking, tamper-proofing, and obfuscation — tools for software protection. *IEEE Trans. Softw. Eng.*, 28, 8, 735–746.

[59] Computing Research and Education Association of Australasia, CORE Inc. [n. d.] CORE. https://www.core.edu.au.

[60] K. Coogan and S. Debray. 2011. Equational reasoning on x86 assembly code. In *IEEE SCAM*, 75–84.

[61] M. Dalla Preda and R. Giacobazzi. 2005. Control Code Obfuscation by Abstract Interpretation. In *IEEE SEFM*, 301–310.

[62] M. Dalla Preda and R. Giacobazzi. 2005. Semantic-Based Code Obfuscation by Abstract Interpretation. In *ICALP*, 1325–1336.

[63] M. Dalla Preda and F. Maggi. 2016. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *J. Comput. Virol. and Hack. Tech.*, 13, 3, 209–232.

[64] S. Datta. 2021. DeepObfusCode: Source Code Obfuscation through Sequence-to-Sequence Networks. In *Intelligent Computing*, 637–647.

[65] R. David, S. Bardin, et al. 2016. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-level Analysis. In *IEEE SANER*.

[66] R. David, L. Coniglio, et al. 2020. QSynth - A Program Synthesis based approach for Binary Code Deobfuscation. In *BAR*.

[67] R. De Ghein, B. Abrath, et al. 2022. ApkDiff: Matching Android App Versions Based on Class Structure. In *ACM CheckMATE*, 1–12.

[68] B. De Sutter, C. Collberg, et al. 2019. Software Protection Decision Support and Evaluation Methodologies (Seminar 19331). *Dagstuhl Reports*, 9, 8, 1–25.

[69] S. H. H. Ding, B. C. M. Fung, et al. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *IEEE S&P*, 472–489.

[70] S. Dong, M. Li, et al. 2018. Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild. In *SecureComm*, 172–192.

[71] W. Dong, J. Lin, et al. 2022. CaDeCFF: Compiler-Agnostic Deobfuscator of Control Flow Flattening. In *Internetware*, 282–291.

[72] D. Dunaev and L. Lengyel. 2012. Complexity of a Special Deobfuscation Problem. In *ECBS*, 1–4.

[73] S. A. Ebad, A. A. Darem, et al. 2021. Measuring Software Obfuscation Quality–A Systematic Literature Review. *IEEE Access*, 9, 99024–99038.

[74] M. Egele, T. Scholte, et al. 2012. A Survey on Automated Dynamic Malware-Analysis Techniques and Tools. *ACM CSUR*, 44, 2.

[75] N. Eyrolles, L. Goubin, et al. 2016. Defeating MBA-Based Obfuscation. In *ACM SPRO*, 27–38.

[76] M. Fyrbiak, S. Rokicki, et al. 2018. Hybrid Obfuscation to Protect Against Disclosure Attacks on Embedded Microprocessors. *IEEE Trans. Comp.*, 67, 3, 307–321.

[77] P. Garba and M. Favaro. 2019. SATURN - Software Deobfuscation Framework Based On LLVM. In *ACM SPRO*, 27–38.

[78] J. Ge, S. Chaudhuri, et al. 2005. Control Flow Based Obfuscation. In *ACM DRM*, 83–92.

[79] F.-X. Geiger and I. Malavolta. 2018. Datasets of Android Applications: a Literature Review. *arXiv preprint arXiv:1809.10069*.

[80] L. Glanz, S. Amann, et al. 2017. CodeMatch: Obfuscation Won't Conceal Your Repackaged App. In *ESEC/FSE*, 638–648.

[81] L. Goubin, P. Paillier, et al. 2020. How to reveal the secrets of an obscure white-box implementation. *J. Crypt. Eng.*, 10, 1, 49–66.

[82] P. Graux, J.-F. Lalande, et al. 2019. Obfuscated Android Application Development. In *CCEC*.

[83] F. Gröbert, C. Willems, et al. 2011. Automated Identification of Cryptographic Primitives in Binary Programs. In *RAID*, 41–60.

[84] GuardSquare. [n. d.] Dexguard. https://www.guardsquare.com/dexguard.

[85] S. Guelton, A. Guinet, et al. 2018. Combining Obfuscation and Optimizations in the Real World. In *IEEE SCAM*, 24–33.

[86] Y. Guillot and A. Gazet. 2010. Automatic binary deobfuscation. *J. in Comput. Virol.*, 6, 3, 261–276.

[87] Y. Guillot and A. Gazet. 2009. Semi-automatic binary protection tampering. *J. in Comput. Virol.*, 5, 2, 119–149.

[88] R. Guo, Q. Liu, et al. 2022. A Survey of Obfuscation and Deobfuscation Techniques in Android Code Protection. In *IEEE DSC*, 40–47.

[89] M. R. Guthaus, J. S. Ringenberg, et al. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE WWC*. IEEE, 3–14.

[90] K. Hajarnis, J. Dalal, et al. 2021. A Comprehensive Solution for Obfuscation Detection and Removal Based on Comparative Analysis of Deobfuscation Tools. In *SMART GENCON*, 1–7.

[91] S. Hamadache and M. Elson. 2020. Creative Manual Code Obfuscation as a Countermeasure Against Software Reverse Engineering. In *AISC*, 3–8.

[92] M. Hammad, J. Garcia, et al. 2018. A Large-Scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-Malware Products. In *ICSE*, 421–431.

[93] N. Hänsch, A. Schankin, et al. 2018. Programming Experience Might Not Help in Comprehending Obfuscated Source Code Efficiently. In *SOUPS*, 341–356.

[94] I. U. Haq and J. Caballero. 2021. A Survey of Binary Code Similarity. *ACM CSUR*, 54, 3, June 2021.

[95] M. Hataba and A. El-Mahdy. 2012. Cloud Protection by Obfuscation: Techniques and Metrics. In *3PGCIC*, 369–372.

[96] M. Horváth and L. Buttyán. 2020. *Cryptographic Obfuscation: A Survey*. Springer.

[97] S. Hosseinzadeh, S. Rauti, et al. 2016. A Survey on Aims and Environments of Diversification and Obfuscation in Software Security. In *CompSysTech*, 113–120.

[98] S. Hosseinzadeh, S. Rauti, et al. 2018. Diversification and obfuscation techniques for software security: A systematic literature review. *Inf. Softw. Technol.*, 104.

[99] Irdeto. [n. d.] Cloakware by irdeto. https://irdeto.com/cloakware-by-irdeto.

[100] Z. Kan, H. Wang, et al. 2019. Deobfuscating Android Native Binary Code. In *ICSE Companion*, 322–323.

[101] S. Kang, J. Kim, et al. 2022. Program Synthesis-Based Simplification of MBA Obfuscated Malware with Restart Strategies. In *ACM CheckMATE*, 13–18.

[102] K. Kaushik, H. S. Sandhu, et al. 2022. A Systematic Approach for Evading Antiviruses Using Malware Obfuscation. In *ETBS*, 29–37.

[103] P. Kochberger, S. Schrittwieser, et al. 2021. SoK: Automatic Deobfuscation of Virtualization-Protected Applications. In *ARES*.

[104] Y. Kosolapov and P. Borisov. 2020. Similarity Features For The Evaluation Of Obfuscation Effectiveness. In *DASA*, 898–902.

[105] C. Kruegel, W. Robertson, et al. 2004. Static Disassembly of Obfuscated Binaries. In *USENIX Security*, 255–270.

[106] K. Kuang, Z. Tang, et al. 2018. Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling. *Comput. & Secur.*, 74, 202–220.

[107] A. Kumar and S. Sharma. 2019. Design and Implementation of Obfuscating Tool for Software Code Protection. In *LNME*, 665–676.

[108] R. Kumar and A. M. Kurian. 2018. A Systematic Study on Static Control Flow Obfuscation Techniques in Java. *arXiv preprint arXiv:1809.11037*.

[109] R. Kumar and A. R. E. Vaishakh. 2016. Detection of Obfuscation in Java Malware. *Procedia Computer Science*, 78, 521–529.

[110] C. Liem, Y. X. Gu, et al. 2008. A Compiler-Based Infrastructure for Software-Protection. In *ACM PLAS*, 33–44.

[111] K. Lim, J. Jeong, et al. 2017. An Anti-Reverse Engineering Technique Using Native Code and Obfuscator-LLVM for Android Applications. In *RACS*, 217–221.

[112] C. Linn and S. Debray. 2003. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM CCS*, 290–299.

[113] C. V. Liţă, D. Cosovan, et al. 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, 107–126.

[114] B. Liu, W. Feng, et al. 2021. Software Obfuscation with Non-Linear Mixed Boolean-Arithmetic Expressions. In *ICISC*, 276–292.

[115] B. Liu, J. Shen, et al. 2021. MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation. In *USENIX Security*, 1701–1718.

[116] H. Liu, C. Sun, et al. 2017. Stochastic Optimization of Program Obfuscation. In *ICSE*, 221–231.

[117] H. Liu. 2016. Towards Better Program Obfuscation: Optimization via Language Models. In *ICSE Companion*, 680–682.

[118] T. Long, L. Liu, et al. 2010. Assure High Quality Code Using Refactoring and Obfuscation Techniques. In *FCST*, 246–252.

[119] L. Luo, J. Ming, et al. 2017. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. *IEEE Trans. Softw. Eng.*, 43, 12, 1157–1177.

[120] B. Lynn, M. Prabhakaran, et al. 2004. Positive Results and Techniques for Obfuscation. In *EUROCRYPT*, 20–39.

[121] M. Madou, L. Van Put, et al. 2006. LOCO: An Interactive Code (De)Obfuscation Tool. In *ACM PEPM*, 140–144.

[122] A. Majumdar, S. Drape, et al. 2007. Metrics-based Evaluation of Slicing Obfuscations. In *IAS*, 472–477.

[123] A. Majumdar, S. Drape, et al. 2007. Slicing Obfuscations: Design, Correctness, and Evaluation. In *ACM DRM*, 70–81.

[124] A. Majumdar, C. Thomborson, et al. 2006. A Survey of Control-Flow Obfuscations. In *ICISS*, 353–356.

[125] R. Manikyam, J. T. McDonald, et al. 2016. Comparing the Effectiveness of Commercial Obfuscators against MATE Attacks. In *ACM SSPREW*.

[126] N. Mavrogiannopoulos, N. Kisserli, et al. 2011. A Taxonomy of Self-Modifying Code for Obfuscation. *Comput. Secur.*, 30, 8, 679–691.

[127] J. T. McDonald, R. Manikyam, et al. 2021. Program Protection through Software-based Hardware Abstraction. In *SECRYPT*, 247–258.

[128] G. Menguy, S. Bardin, et al. 2021. Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate. In *ACM CCS*, 2513–2525.

[129] P. Mila. [n. d.] contagio. https://contagiodump.blogspot.com.

[130] J. Ming, F. Zhang, et al. 2016. Deviation-Based Obfuscation-Resilient Program Equivalence Checking With Application to Software Plagiarism Detection. *IEEE Trans. Reliability*, 65, 4, 1647–1664.

[131] A. Mohammadinodooshan, U. Kargén, et al. 2019. Robust Detection of Obfuscated Strings in Android Apps. In *ACM AISec*, 25–35.

[132] M. Moog, M. Demmel, et al. 2021. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In *IEEE/IFIP DSN*, 569–580.

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

[134] M. Ollivier, S. Bardin, et al. 2019. How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-Oriented Protections). In *ACM ACSAC*, 177–189.

[135] M. Ollivier, S. Bardin, et al. 2019. Obfuscation: Where Are We in Anti-DSE Protections? (A First Attempt). In *ACM SSPREW*.

[136] C. B. Parker., J. T. McDonald., et al. 2021. Machine Learning Classification of Obfuscation using Image Visualization. In *SECRYPT*, 854–859.

[137] U. Piazzalunga, P. Salvaneschi, et al. 2007. Security Strength Measurement for Dongle-Protected Software. *IEEE Security & Privacy*, 5, 6, 32–40.

[138] I. V. Popov, S. K. Debray, et al. 2007. Binary Obfuscation Using Signals. In *USENIX Security*, 275–290.

[139] J. Qiu, J. Zhang, et al. 2020. A Survey of Android Malware Detection with Deep Neural Models. *ACM CSUR*, 53, 6.

[140] D. A. Quist and L. M. Liebrock. 2009. Visualizing compiled executables for malware analysis. In *IEEE VizSec*, 27–32.

[141] L. Regano, D. Canavese, et al. 2017. Towards Optimally Hiding Protected Assets in Software Applications. In *QRS*, 374–385.

[142] L. Regano, D. Canavese, et al. 2016. Towards Automatic Risk Analysis and Mitigation of Software Applications. In *WISTP*, 120–135.

[143] B. Reichenwallner and P. Meerwald-Stadler. 2022. Efficient Deobfuscation of Linear Mixed Boolean-Arithmetic Expressions. In *ACM CheckMATE*, 19–28.

[144] K. A. Roundy and B. P. Miller. 2013. Binary-Code Obfuscations in Prevalent Packer Tools. *ACM CSUR*, 46, 1.

[145] A. Salem and S. Banescu. 2016. Metadata Recovery from Obfuscated Programs Using Machine Learning. In *ACM SSPREW*.

[146] S. Sarker, J. Jueckstock, et al. 2020. Hiding in Plain Site: Detecting JavaScript Obfuscation through Concealed Browser API Usage. In *ACM IMC*, 648–661.

[147] M. Schloegel, T. Blazytko, et al. 2022. Loki: Hardening Code Obfuscation Against Automated Attacks. In *USENIX Security*, 3055–3073.

[148] S. Schrittwieser, S. Katzenbeisser, et al. 2016. Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM CSUR*, 49, 1, 4:1–4:37.

[149] S. Schrittwieser, P. Kochberger, et al. 2022. Obfuscation-Resilient Semantic Functionality Identification Through Program Simulation. In *NordSec*, 273–291.

[150] S. A. Sebastian, S. Malgaonkar, et al. 2016. A study & review on code obfuscation. In *WCFTR (Startup Conclave)*, 1–6.

[151] S. Semenov, V. Davydov, et al. 2019. Obfuscated Code Quality Measurement. In *MMA*, 1–6.

[152] Z. Sha, H. Shu, et al. 2022. Model of Execution Trace Obfuscation Between Threads. *IEEE Trans. Dep. Sec. Comp.*, 19, 6, 4156–4171.

[153] L. Shijia, J. Chunfu, et al. 2022. Chosen-Instruction Attack Against Commercial Code Virtualization Obfuscators. In *NDSS*.

[154] Y. Shoshitaishvili, R. Wang, et al. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE S&P*.

[155] P. Sivadasan and P. S. Lal. 2011. Suggesting potency measures for obfuscated arrays and usage of source code obfuscators for intellectual property protection of Java products. In *ICINT*.

[156] P. Skolka, C.-A. Staicu, et al. 2019. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *WWW*, 1735–1746.

[157] Y. Song, M. E. Locasto, et al. 2010. On the Infeasibility of Modeling Polymorphic Shellcode. *Mach. Learn.*, 81, 2, 179–205.

[158] Standard Performance Evaluation Corporation (SPEC). [n. d.] SPEC — Standard Performance Evaluation Corporation. https://spec.org.

[159] J. Stephens, B. Yadegari, et al. 2018. Probabilistic obfuscation through covert channels. In *IEEE EuroS&P*, 243–257.

[160] F.-H. Su, J. Bell, et al. 2018. Obfuscation Resilient Search through Executable Classification. In *ACM MAPL*, 20–30.
[161] A. J. Suresh and S. Sankaran. 2020. Power Profiling and Analysis of Code Obfuscation for Embedded Devices. In *IEEE INDICON*, 1–6.
[162] A. J. Suresh and S. Sankaran. 2020. A Framework for Evaluation of Software Obfuscation Tools for Embedded Devices. In *ATIS*, 1–13.
[163] M. Talukder, S. Islam, et al. 2019. Analysis of Obfuscated Code with Program Slicing. In *Cyber Security*, 1–7.
[164] X. Tang, Y. Liang, et al. 2017. On the Effectiveness of Code-Reuse-Based Android Application Obfuscation. In *ICISC*, 333–349.
[165] Z. Tian, H. Mao, et al. 2022. Fine-Grained Obfuscation Scheme Recognition on Binary Code. In *ICDF2C*, 215–228.
[166] D. Titze, M. Lux, et al. 2017. Ordol: Obfuscation-Resilient Detection of Libraries in Android Applications. In *IEEE Trustcom/BigDataSE/ICESS*, 618–625.
[167] R. Tofighi-Shirazi, I. Asăvoae, et al. 2019. Defeating Opaque Predicates Statically through Machine Learning and Binary Analysis. *arXiv preprint arXiv:1909.01640*.
[168] R. Tofighi-Shirazi, I. M. Asăvoae, et al. 2019. Fine-Grained Static Detection of Obfuscation Transforms Using Ensemble-Learning and Semantic Reasoning. In *ACM SSPREW*.
[169] D. Ugarte, D. Maiorca, et al. 2019. PowerDrive: Accurate De-obfuscation and Analysis of PowerShell Malware. In *DIMVA*, 240–259.
[170] Université du Luxembourg. [n. d.] Androzoo home. https://androzoo.uni.lu.
[171] J. Van den Broeck, B. Coppens, et al. 2021. Obfuscated integration of software protections. *Int. J. Inf. Secur.*, 20, 73–101.
[172] B. Vasilescu, C. Casalnuovo, et al. 2017. Recovering Clear, Natural Identifiers from Obfuscated JS Names. In *ESEC/FSE*, 683–693.
[173] VirusShare.com. [n. d.] VirusShare.com. https://virusshare.com.
[174] A. Viticchié, L. Regano, et al. 2016. Assessment of Source Code Obfuscation Techniques. In *IEEE SCAM*, 11–20.
[175] A. Viticchié, L. Regano, et al. 2020. Empirical assessment of the effort needed to attack programs protected with client/server code splitting. *Empir. Softw. Eng.*, 25, 1, 1–48.
[176] P. Wang, Q. Bao, et al. 2018. Software protection on the go: a large-scale empirical study on mobile app obfuscation. In *ICSE*, 26–36.
[177] P. Wang, S. Wang, et al. 2016. Translingual Obfuscation. *IEEE EuroS&P*, 128–144.
[178] P. Wang, D. Wu, et al. 2018. Protecting Million-User IOS Apps with Obfuscation: Motivations, Pitfalls, and Experience. In *ICSE-SEIP*, 235–244.
[179] Y. Wang, H. Wu, et al. 2018. ORLIS: Obfuscation-Resilient Library Detection for Android. In *MOBILESoft*, 13–23.
[180] Y. Wang, Y. Shen, et al. 2019. CFHider: Control Flow Obfuscation with Intel SGX. In *IEEE INFOCOM*, 541–549.
[181] D. Wermke, N. Huaman, et al. 2018. A Large Scale Investigation of Obfuscation Use in Google Play. In *ACM ACSAC*, 222–235.
[182] C. Willems and F. C. Freiling. 2012. Reverse Code Engineering – State of the Art and Countermeasures. *it - Information Technology*, 54, 2, 53–63.
[183] P. Wrench and B. Irwin. 2016. Detecting Derivative Malware Samples Using Deobfuscation-Assisted Similarity Analysis. *SAIEE Africa Research J.*, 107, 2, 65–77.
[184] Z. Wu, S. Gianvecchio, et al. 2010. Mimimorphism: A New Approach to Binary Code Obfuscation. In *ACM CCS*, 536–546.
[185] D. Xu, J. Ming, et al. 2017. Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping. In *IEEE S&P*, 921–937.
[186] D. Xu, B. Liu, et al. 2021. Boosting SMT solver performance on mixed-bitwise-arithmetic expressions. In *ACM PLDI*, 651–664.
[187] H. Xu, Y. Zhou, et al. 2020. Layered obfuscation: a taxonomy of software obfuscation techniques for layered security. *Cybersecurity*, 3, 1, 9.
[188] H. Xu, Y. Zhou, et al. 2017. On Secure and Usable Program Obfuscation: A Survey. *arXiv preprint arXiv:1710.01139*.
[189] B. Yadegari, J. Stephens, et al. 2017. Analysis of Exception-Based Control Transfers. In *ACM CODASPY*, 205–216.
[190] Z. Yajin and J. Xuxian. [n. d.] Android Malware Genome Project. http://www.malgenomeproject.org.
[191] I. You and K. Yim. 2010. Malware obfuscation techniques: A brief survey. In *IEEE BWCCA*, 297–300.

[192] J. Zeng, Y. Fu, et al. 2013. Obfuscation Resilient Binary Code Reuse through Trace-Oriented Programming. In *ACM CCS*, 487–498.

[193] Q. Zeng, L. Luo, et al. 2019. Resilient User-Side Android Application Repackaging and Tampering Detection Using Cryptographically Obfuscated Logic Bombs. *IEEE Trans. Dep. Sec. Comp.*, 1–1.

[194] J. Zhang, A. R. Beresford, et al. 2019. LibID: Reliable Identification of Obfuscated Third-Party Android Libraries. In *ACM ISSTA*, 55–65.

[195] X. Zhang, J. Pang, et al. 2018. Common Program Similarity Metric Method for Anti-Obfuscation. *IEEE Access*, 6, 47557–47565.

[196] X. Zhang, F. Breitinger, et al. 2021. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *FSI: Digital Investigation*, 39.

[197] X. Zhang, F. He, et al. 2008. An Inter-Classes Obfuscation Method for Java Program. In *ISA*, 360–365.

[198] X. Zhang, F. He, et al. 2010. Theory and practice of program obfuscation. *Convergence and Hybrid Information Technologies*, 426.

[199] Y. Zhao, Z. Tang, et al. 2021. Input-Output Example-Guided Data Deobfuscation on Binary. *Security and Communication Networks*, 2021.

[200] Y. Zhao, Z. Tang, et al. 2020. Semantics-aware obfuscation scheme prediction for binary. *Comput. & Secur.*, 99, 102072.

[201] H. Zhou, T. Chen, et al. 2020. UI Obfuscation and Its Effects on Automated UI Analysis for Android Apps. In *IEEE/ACM ASE*, 199–210.

[202] Y. Zhuang, M. Protsenko, et al. 2014. An(other) Exercise in Measuring the Strength of Source Code Obfuscation. In *DEXA*, 313–317.

[203] B. Abrath, B. Coppens, et al. 2020. Resilient Self-Debugging Software Protection. In *IEEE EuroS&P Workshops*, 606–615.

[204] B. Abrath, B. Coppens, et al. 2015. Obfuscating Windows DLLs. In *IEEE/ACM SPRO*, 24–30.

[205] A. Aghamohammadi and F. Faghih. 2019. Lightweight versus obfuscation-resilient malware detection in android applications. *J. Comput. Virol. and Hack. Tech.*, 16, 2, 125–139.

[206] P. Ahire and J. Abraham. 2020. Mechanisms for Source Code Obfuscation in C: Novel Techniques and Implementation. In *ECSI*, 52–59.

[207] M. Ahmadvand, D. Below, et al. 2019. VirtSC: Combining Virtualization Obfuscation with Self-Checksumming. In *ACM SPRO*, 53–63.

[208] V. Ajiri, S. Butakov, et al. 2020. Detection Efficiency of Static Analyzers against Obfuscated Android Malware. In *IEEE BigDataSecurity, HPSC, and IDS*, 231–234.

[209] A. B. Ajmal, A. Anjum, et al. 2021. Novel Approach for Concealing Penetration Testing Payloads Using Data Privacy Obfuscation Techniques. In *IEEE HONET*, 44–49.

[210] Y. Alec and J. T. Mcdonald. 2008. Tamper Resistant Software Through Intent Protection. *International Journal of Network Security*, 7, 370–382, 3.

[211] B. Anckaert, M. Jakubowski, et al. 2006. Proteus: Virtualization for Diversified Tamper-Resistance. In *ACM DRM*, 47–58.

[212] B. Anckaert, M. H. Jakubowski, et al. 2009. Runtime Protection via Dataflow Flattening. In *SECUREWARE*, 242–248.

[213] W. Andrew, M. Rachit, et al. 2006. Normalizing Metamorphic Malware Using Term Rewriting. In *IEEE SCAM*, 75–84.

[214] S. Armoogum and A. Caully. 2010. Obfuscation techniques for mobile agent code confidentiality. *Journal of E-Technology*, 1, 2, 83–94.

[215] A. Arora, T. Stallings, et al. 2017. Malware Secrets: De-Obfuscating in the Cloud. In *IEEE CLOUD*, 753–756.

[216] A. Arrott, A. Lakhotia, et al. 2018. Cluster analysis for deobfuscation of malware variants during ransomware attacks. In *Cyber SA*, 1–9.

[217] A. Bacci, A. Bartoli, et al. 2018. Detection of Obfuscation Techniques in Android Applications. In *ARES*, 1–9.

[218] A. Bacci, A. Bartoli, et al. 2018. Impact of Code Obfuscation on Android Malware Detection based on Static and Dynamic Analysis. In *SecureComm*, 379–385.

[219] K. Bakour, H. M. Ünver, et al. 2019. A Deep Camouflage: Evaluating Android's Anti-malware Systems Robustness Against Hybridization of Obfuscation Techniques with Injection Attacks. *Arabian Journal for Science and Engineering*, 44, 11, 9333–9347.

[220] V. Balachandran and S. Emmanuel. 2011. Software code obfuscation by hiding control flow information in stack. In *IEEE WIFS*, 1–6.

[221] V. Balachandran and S. Emmanuel. 2013. Potent and stealthy control flow obfuscation by stack based self-modifying code. *IEEE Trans. Inf. For. Sec.*, 8, 4, 669–681.

[222] V. Balachandran and S. Emmanuel. 2013. Software Protection with Obfuscation and Encryption. In *ISPEC*, 309–320.

[223] V. Balachandran, S. Emmanuel, et al. 2014. Obfuscation by code fragmentation to evade reverse engineering. In *IEEE SMC*, 463–469.

[224] V. Balachandran, W. K. Ng, et al. 2014. Function Level Control Flow Obfuscation for Software Security. In *CISIS*, 133–140.

[225] V. Balachandran, Sufatrio, et al. 2016. Control flow obfuscation for Android applications. *Comput. & Secur.*, 61, 72–93.

[226] S. Banescu, C. Lucaci, et al. 2016. VOT4CS: A Virtualization Obfuscation Tool for C#. In *ACM SPRO*, 39–49.

[227] S. Banescu, T. Wuchner, et al. 2015. A framework for empirical evaluation of malware detection resilience against behavior obfuscation. In *MALWARE*, 40–47.

[228] L. Barhelemy, N. Eyrolles, et al. 2016. Binary Permutation Polynomial Inversion and Application to Obfuscation Techniques. In *ACM SPRO*, 51–59.

[229] C. Barria, D. Cordero, et al. 2016. Obfuscation procedure based in dead code insertion into crypter. In *ICCCC*, 23–29.

[230] C. Basile and M. Ceccato. 2013. Towards a unified software attack model to assess software protections. In *IEEE ICPC*, 219–222.

[231] M. Batchelder and L. Hendren. 2007. Obfuscating Java: The Most Pain for the Least Gain. In *CC*, 96–110.

[232] C. K. Behera and D. L. Bhaskari. 2016. Code Obfuscation by Using Floating Points and Conditional Statements. In *AISC*, 569–578.

[233] C. K. Behera and D. L. Bhaskari. 2015. Different Obfuscation Techniques for Code Protection. *Procedia Computer Science*, 70, 757–763.

[234] C. K. Behera, G. Sanjog, et al. 2019. Control Flow Graph Matching for Detecting Obfuscated Programs. In *AISC*, 267–275.

[235] B. Bertholon, S. Varrette, et al. 2013. JShadObf: A JavaScript Obfuscator Based on Multi-Objective Optimization Algorithms. In *NSS*, 336–349.

[236] B. Bertholon, S. Varrette, et al. 2013. ShadObf: A C-Source Obfuscator Based on Multi-objective Optimisation Algorithms. In *IEEE IPDPSW*, 435–444.

[237] S. Bhansali, A. Aris, et al. 2022. A First Look at Code Obfuscation for WebAssembly. In *WiSec*, 140–145.

[238] F. Biondi, S. Josse, et al. 2017. Effectiveness of synthesis in concolic deobfuscation. *Comput. & Secur.*, 70, 500–515.

[239] N. Bitansky, R. Canetti, et al. 2011. Program Obfuscation with Leaky Hardware. In *ASIACRYPT*, 722–739.

[240] S. Blazy, S. Riaud, et al. 2015. Data tainting and obfuscation: Improving plausibility of incorrect taint. In *IEEE SCAM*, 111–120.

[241] A. Bolat, S. H. Çelik, et al. 2022. ERIC: An Efficient and Practical Software Obfuscation Framework. In *IEEE/IFIP DSN*, 466–474.

[242] P. T. Breuer, J. P. Bowen, et al. 2017. Encrypted computing: Speed, security and provable obfuscation against insiders. In *ICCST*, 1–6.

[243] R. Bruni, R. Giacobazzi, et al. 2018. Code Obfuscation Against Abstract Model Checking Attacks. In *VMCAI*, 94–115.

[244] R. Bruni, R. Giacobazzi, et al. 2018. Code obfuscation against abstraction refinement attacks. *Formal Aspects of Computing*, 30, 6, 685–711.

[245] D. Bruschi, L. Martignoni, et al. 2007. Code Normalization for Self-Mutating Malware. *IEEE Security and Privacy*, 5, 46–54, 2.

[246] C. Bunse. 2018. On the Impact of Code Obfuscation to Software Energy Consumption. In *Progress in IS*, 239–249.

[247] A. Cabutto, P. Falcarin, et al. 2015. Software Protection with Code Mobility. In *ACM MTD*, 95–103.

[248] M. Campion, M. Dalla Preda, et al. 2021. Learning metamorphic malware signatures from samples. *J. Comput. Virol. and Hack. Tech.*, 17, 1–17.

[249] D. Canavese, L. Regano, et al. 2017. Estimating Software Obfuscation Potency with Artificial Neural Networks. In *STM*, 193–202.

[250] A. Capiluppi, P. Falcarin, et al. 2012. Code Defactoring: Evaluating the Effectiveness of Java Obfuscations. In *WCRE*, 71–80.

[251] J. Cappaert, N. Kisserli, et al. 2006. Self-encrypting Code to Protect Against Analysis and Tampering. In *WISSec*.

[252] J. Cappaert and B. Preneel. 2010. A general model for hiding control flow. In *ACM DRM*, 35–42.

[253] L. Cassano, E. Lazzeri, et al. 2022. On the optimization of Software Obfuscation against Hardware Trojans in Microprocessors. In *DDECS*, 172–177.

[254] M. Ceccato, M. Dalla Preda, et al. 2007. Barrier Slicing for Remote Software Trusting. In *IEEE SCAM*, 27–36.

[255] M. Ceccato, M. Dalla Preda, et al. 2009. Trading-off Security and Performance in Barrier Slicing for Remote Software Entrusting. *Automated Software Engineering*, 16, 2, 235–261.

[256] M. Ceccato, P. Falcarin, et al. 2016. Search Based Clustering for Protecting Software with Diversified Updates. In *SSBSE*, 159–175.

[257] M. Ceccato and P. Tonella. 2011. CodeBender: Remote Software Protection Using Orthogonal Replacement. *IEEE Software*, 28, 28–34, 2.

[258] M. Ceccato, P. Tonella, et al. 2009. Remote Software Protection by Orthogonal Client Replacement. In *ACM SAC*, 448–455.

[259] H. Chai, L. Ying, et al. 2022. Invoke-Deobfuscation: AST-Based and Semantics-Preserving Deobfuscation for PowerShell Scripts. In *IEEE/IFIP DSN*, 295–306.

[260] R. S. Chakraborty, S. Narasimhan, et al. 2011. Embedded software security through key-based control flow obfuscation. In *InfoSecHiComNet*, 30–44.

[261] J.-T. Chan and W. Yang. 2004. Advanced obfuscation techniques for Java bytecode. *Journal of systems and software*, 71, 1–10, 1–2.

[262] Z. Chen, C. Jia, et al. 2017. Hidden Path: Dynamic Software Watermarking Based on Control Flow Obfuscation. In *IEEE CSE - IEEE EUC*, 443–450.

[263] H. Cho, J. H. Yi, et al. 2018. DexMonitor: Dynamically Analyzing and Monitoring Obfuscated Android Applications. *IEEE Access*, 6, 71229–71240.

[264] T. Cho, H. Kim, et al. 2017. Security Assessment of Code Obfuscation Based on Dynamic Monitoring in Android Things. *IEEE Access*, 5, 6361–6371.

[265] A. Choliy, F. Li, et al. 2017. Obfuscating function call topography to test structural malware detection against evasion attacks. In *ICNC*, 808–813.

[266] M. R. Chouchane and A. Lakhotia. 2006. Using Engine Signature to Detect Metamorphic Malware. In *ACM WORM*, 73–78.

[267] S. Chow, Y. Gu, et al. 2001. An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs. In *Information Security*, 144–155.

[268] M. Christodorescu, S. Jha, et al. 2005. Semantics-Aware Malware Detection. In *IEEE S&P*, 32–46.

[269] M. Chua and V. Balachandran. 2018. Effectiveness of Android Obfuscation on Evading Anti-Malware. In *ACM CODASPY*, 143–145.

[270] A. Cimitile, F. Martinelli, et al. 2017. Formal Methods Meet Mobile Code Obfuscation Identification of Code Reordering Technique. In *IEEE WETICE*, 263–268.

[271] F. B. Cohen. 1993. Operating System Protection through Program Evolution. *Comput. & Secur.*, 12, 6, 565–584.

[272] C. Collberg, G. Myles, et al. 2003. Sandmark — A Tool for Software Protection Research. *IEEE Security and Privacy*, 1, 4, 40–49.

[273] C. Collberg, C. Thomborson, et al. 1998. Breaking Abstractions and Unstructuring Data Structures. In *ICCL*, 28–38.

[274] P. Comparetti, G. Salvaneschi, et al. 2010. Identifying Dormant Functionality in Malware Programs. In *IEEE S&P*, 61–76.

[275] K. Coogan, S. Debray, et al. 2009. Automatic Static Unpacking of Malware Binaries. In *WCRE*, 167–176.

[276] K. Coogan, G. Lu, et al. 2011. Deobfuscation of Virtualization-Obfuscated Software: A Semantics-Based Approach. In *ACM CCS*, 275–284.

[277] B. Coppens, B. De Sutter, et al. 2013. Feedback-Driven Binary Code Diversification. *ACM Trans. Archit. Code Optim.*, 9, 4.

[278] B. Coppens, B. De Sutter, et al. 2013. Protecting your software updates. *IEEE Security Privacy*, 11, 2, 47–54.

[279] R. Costa, L. Pirmez, et al. 2012. TinyObf: Code Obfuscation Framework for Wireless Sensor Networks. In *ICWN*.

[280] V. C. Crăciun and A.-C. Mogage. 2021. Building Deobfuscated Applications From Polymorphic Binaries. In *SecITC*, 308–323.

[281] G. Crincoli, T. Marinaro, et al. 2020. Code Reordering Obfuscation Technique Detection by Means of Weak Bisimulation. In *AINA*, 1368–1382.

[282] J. Cui, Z. Song, et al. 2019. A Generalized Obfuscation Method to Protect Software of Mobile Apps. In *MSN*, 31–36.

[283] B. Cyr, J. Mahmod, et al. 2019. Low-Cost and Secure Firmware Obfuscation Method for Protecting Electronic Systems From Cloning. *IEEE Internet of Things Journal*, 6, 2, 3700–3711.

[284] A. K. Dalai, S. S. Das, et al. 2017. A code obfuscation technique to prevent reverse engineering. In *WiSPNET*, 828–832.

[285] M. Dalla Preda, W. Feng, et al. 2012. Twisting Additivity in Program Obfuscation. In *ICISTM*, 336–347.

[286] M. Dalla Preda, R. Giacobazzi, et al. 2010. Modelling Metamorphism by Abstract Interpretation. In *SAS*, 218–235.

[287] M. Dalla Preda, R. Giacobazzi, et al. 2015. Unveiling Metamorphism by Abstract Interpretation of Code Properties. *Theor. Comput. Sci.*, 577, C, 74–97.

[288] M. Dalla Preda, M. Madou, et al. 2006. Opaque Predicates Detection by Abstract Interpretation. In *AMAST*, 81–95.

[289] M. Dalla Preda and I. Mastroeni. 2018. Characterizing a property-driven obfuscation strategy. *Journal of Computer Security*, 26, 1–39.

[290] M. Dalla Preda, I. Mastroeni, et al. 2013. A Formal Framework for Property-Driven Obfuscation Strategies. In *FCT*, 133–144.

[291] Q.-V. Dang. 2022. Enhancing Obfuscated Malware Detection with Machine Learning Techniques. In *FDSE*, 731–738.

[292] S. M. Darwish, S. K. Guirguis, et al. 2010. Stealthy code obfuscation technique for software security. In *ICCES*, 93–99.

[293] B. De Sutter, B. Anckaert, et al. 2009. Instruction Set Limitation in Support of Software Diversity. In *ICISC*, 152–165.

[294] B. De Sutter, P. Falcarin, et al. 2016. A Reference Architecture for Software Protection. In *IEEE/IFIP WICSA*, 291–294.

[295] S. Debray and J. Patel. 2010. Reverse Engineering Self-Modifying Code: Unpacker Extraction. In *WCRE*, 131–140.

[296] D. Demicco, R. Erinfolami, et al. 2021. Program Obfuscation via ABI Debiasing. In *ACSAC*, 146–157.

[297] B. F. Demissie, M. Ceccato, et al. 2015. Assessment of Data Obfuscation with Residue Number Coding. In *IEEE/ACM SPRO*, 38–44.

[298] L. Deshotels, V. Notani, et al. 2014. DroidLegacy: Automated Familial Classification of Android Malware. In *ACM PPREW*.

[299] K. A. Dhanya, O. K. Dheesha, et al. 2020. Detection of Obfuscated Mobile Malware with Machine Learning and Deep Learning Models. In *SoMMA*, 221–231.

[300] B. Dharmalingam, A. Liu, et al. 2022. FineObfuscator: Defeating Reverse Engineering Attacks with Context-sensitive and Cost-efficient Obfuscation for Android Apps. In *IEEE eIT*, 368–374.

[301] W. Ding and Z. Gu. 2016. A reverse engineering approach of obfuscated array. In *ICACT*, 175–179.

[302] D. Dolz and G. Parra. 2008. Using exception handling to build opaque predicates in intermediate code obfuscation techniques. *Journal of Computer Science & Technology*, 8.

[303] S. Drape. 2006. An Obfuscation for Binary Trees. In *IEEE TENCON*, 1–4.

[304] S. Drape. 2007. Generalising the Array Split Obfuscation. *Information Sciences*, 177, 202–219, 1.

[305] S. Drape, A. Majumdar, et al. 2007. Slicing Aided Design of Obfuscating Transforms. In *IEEE/ACIS ICIS*, 1019–1024.

[306] S. Drape, C. Thomborson, et al. 2007. Specifying Imperative Data Obfuscations. In *ICIS*, 299–314.

[307] D. Dunaev and L. Lengyel. 2013. Aspects of Intermediate Level Obfuscation. In *ECBS-EERC*, 138–142.

[308] D. Dunaev and L. Lengyel. 2014. Cognitive evaluation of intermediate level obfuscator. In *CogInfoCom*, 521–525.

[309] L. Ertaul and S. Venkatesh. 2004. JHide-A tool kit for code obfuscation. In *IASTED SEA*, 133–138.

[310] L. Ertaul and S. Venkatesh. 2005. Novel Obfuscation Algorithms for Software Security. In *SERP*, 209–215.

[311] P. Falcarin, S. Di Carlo, et al. 2011. Exploiting code mobility for dynamic binary obfuscation. In *WorldCIS*, 114–120.

[312] H. Fang, Y. Wu, et al. 2011. Multi-stage binary code obfuscation using improved virtual machine. In *ISC*, 168–181.

[313] A. Fass, R. P. Krawczyk, et al. 2018. JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript. In *DIMVA*, 303–325.

[314] J. Feichtner and C. Rabensteiner. 2019. Obfuscation-Resilient Code Recognition in Android Apps. In *ARES*.

[315] R. Fellin and M. Ceccato. 2020. Experimental assessment of XOR-Masking data obfuscation based on K-Clique opaque constants. *Journal of Systems and Software*, 162, 110492.

[316] C. Foket, B. De Sutter, et al. 2014. Pushing Java Type Obfuscation to the Limit. *IEEE Trans. Dep. Sec. Comp.*, 11, 6, 553–567.

[317] C. Foket, K. De Bosschere, et al. 2020. Effective and efficient Java-type obfuscation. *Software: Practice and Experience*, 50, 2, 136–160.

[318] C. Foket, B. De Sutter, et al. 2013. A Novel Obfuscation: Class Hierarchy Flattening. In *FPS*, 194–210.

[319] A. Foroughipour, N. Stakhanova, et al. 2022. AndroClonium: Bytecode-Level Code Clone Detection for Obfuscated Android Apps. In *SEC*, 379–397.

[320] K. Fukushima, S. Kiyomoto, et al. 2006. An obfuscation scheme using affine transformation and its implementation. *IPSJ Digital Courier*, 2, 498–512.

[321] K. Fukushima, S. Kiyomoto, et al. 2009. Obfuscation mechanism in conjunction with tamper-proof module. In *CSE*, 665–670.

[322] J. Garcia, M. Hammad, et al. 2018. Lightweight, Obfuscation-Resilient Detection and Family Identification of Android Malware. *ACM Trans. Softw. Eng. Methodol.*, 26, 3, January 2018.

[323] A. Ghafarian, D. Keskin, et al. 2021. An Assessment of Obfuscated Ransomware Detection and Prevention Methods. In *FICC*, 793–811.

[324] S. Ghosh, J. Hiser, et al. 2012. Replacement Attacks against VM-Protected Applications. In *ACM VEE*, 203–214.

[325] S. Ghosh, J. Hiser, et al. 2013. Software Protection for Dynamically-Generated Code. In *ACM PPREW*.

[326] S. Ghosh, J. D. Hiser, et al. 2015. Matryoshka: strengthening software protection via nested virtual machines. In *IEEE/ACM SPRO*, 10–16.

[327] S. Ghosh, J. D. Hiser, et al. 2010. A Secure and Robust Approach to Software Tamper Resistance. In *Information Hiding*, 33–47.

[328] R. Giacobazzi. 2008. Hiding Information in Completeness Holes: New Perspectives in Code Obfuscation and Watermarking. In *IEEE SEFM*, 7–18.

[329] R. Giacobazzi. 2012. Software Security by Obscurity A Programming Language Perspective. In *ICISTM*, 427–432.

[330] R. Giacobazzi, N. Jones, et al. 2012. Obfuscation by partial evaluation of distorted interpreters. In *ACM POPL*, 63–72.

[331] R. Giacobazzi and I. Mastroeni. 2012. Making Abstract Interpretation Incomplete: Modeling the Potency of Obfuscation. In *SAS*, 129–145.

[332] R. Giacobazzi and I. Mastroeni. 2022. Property-Driven Code Obfuscations Reinterpreting Jones-Optimality in Abstract Interpretation. In *SAS*, 247–271.

[333] R. Giacobazzi, I. Mastroeni, et al. 2017. Maximal incompleteness as obfuscation potency. *Formal Aspects of Computing*, 29, 1, 3–31.

[334] L. Glanz, P. Müller, et al. 2020. Hidden in Plain Sight: Obfuscated Strings Threatening Your Privacy. In *ACM Asia CCS*, 694–707.

[335] S. Gnatyuk, V. Kinzeryavyy, et al. 2020. Code Obfuscation Technique for Enhancing Software Protection Against Reverse Engineering. In *AIMEE*, 571–580.

[336] P. Graux, J.-F. Lalande, et al. 2020. Abusing Android Runtime for Application Obfuscation. In *IEEE Euro S&P Workshops*, 616–624.

[337] S. Guang, F. Xiaoping, et al. 2019. Obfuscation-Based Watermarking for Mobile Service Application Copyright Protection in the Cloud. *IEEE Access*, 7, 38162–38167.

[338] X. Guangli and C. Zheng. 2010. The code obfuscation technology based on class combination. In *DCABES*, 479–483.

[339] W. Guizani, J.-Y. Marion, et al. 2009. Server-side dynamic code analysis. In *MALWARE*, 55–62.

[340] N. Gupta, S. Naval, et al. 2014. P-SPADE: GPU accelerated malware packer detection. In *PST*, 257–263.

[341] N. M. Hai1, M. Ogawa, et al. 2016. Obfuscation Code Localization Based on CFG Generation of Malware. In *FPS*, 229–247.

[342] A. M. H. Al-Hakimi, A. B. M. Sultan, et al. 2020. Hybrid Obfuscation Technique to Protect Source Code From Prohibited Software Reverse Engineering. *IEEE Access*, 8, 187326–187342.

[343] G. T. Harter and N. C. Rowe. 2022. Testing Detection of K-Ary Code Obfuscated by Metamorphic and Polymorphic Techniques. In *NCS*, 110–123.

[344] M. Hataba, R. Elkhouly, et al. 2015. Diversified Remote Code Execution Using Dynamic Obfuscation of Conditional Branches. In *IEEE ICDCSW*, 120–127.

[345] M. Hataba, A. El-Mahdy, et al. 2015. OJIT: A Novel Obfuscation Approach Using Standard Just-In-Time Compiler Transformations. In *International Workshop on Dynamic Compilation Everywhere*.

[346] M. Hataba, A. Sherif, et al. 2022. Enhanced Obfuscation for Software Protection in Autonomous Vehicular Cloud Computing Platforms. *IEEE Access*, 10, 33943–33953.

[347] K. Heffner and C. Collberg. 2004. The Obfuscation Executive. In *ISC*, 428–440.

[348] A. Herrera. 2020. Optimizing Away JavaScript Obfuscation. In *IEEE SCAM*, 215–220.

[349] J. Hoffmann, T. Rytilahti, et al. 2016. Evaluating Analysis Tools for Android Apps: Status Quo and Robustness Against Obfuscation. In *ACM CODASPY*, 139–141.

[350] W. Holder, J. T. McDonald, et al. 2017. Evaluating Optimal Phase Ordering in Obfuscation Executives. In *ACM SSPREW*.

[351] M. M. Hossain, S. Mohammad, et al. 2021. HEXON: Protecting Firmware Using Hardware-Assisted Execution-Level Obfuscation. In *ISVLSI*, 343–349.

[352] S. Hosseinzadeh, S. Hyrynsalmi, et al. 2015. Security and Privacy in Cloud Computing via Obfuscation and Diversification: A Survey. In *IEEE CloudCom*, 529–535.

[353] T.-W. Hou, H.-Y. Chen, et al. 2006. Three control flow obfuscation methods for Java software. *IEE Proceedings-Software*, 153, 80–86, 2.

[354] N. Hu, X. Ma, et al. 2020. DexFus: An Android Obfuscation Technique Based on Dalvik Bytecode Translation. In *FCS*, 419–431.

[355] W. Hu, J. Cheng, et al. 2022. A GAN-Based Anti-obfuscation Detection Method for Malicious Code. In *PRML*, 484–488.

[356] Z. Hu, B. V. R. E. Silva, et al. 2022. SEMEO: A Semantic Equivalence Analysis Framework for Obfuscated Android Applications. In *MobiQuitous*, 322–346.

[357] Y.-L. Huang and H.-Y. Tsai. 2012. A framework for quantitative evaluation of parallel control-flow obfuscation. *computers & security*, 31, 8, 886–896.

[358] C. B. Huidobro, D. Cordero, et al. 2018. Obfuscation procedure based on the insertion of the dead code in the crypter by binary search. In *ICCCC*, 183–192.

[359] J. Hwang and T. Han. 2018. Identifying Input-Dependent Jumps from Obfuscated Execution using Dynamic Data Flow Graphs. In *ACM SSPREW*, 3.

[360] R. N. Ismanto and M. Salman. 2017. Improving Security Level through Obfuscation Technique for Source Code Protection Using AES Algorithm. In *ICCNS*, 18–22.

[361] G. Jacob, P. M. Comparetti, et al. 2013. A Static, Packer-Agnostic Filter to Detect Similar Malware Samples. In *DIMVA*, 102–122.

[362] A. Jain, H. Gonzalez, et al. 2015. Enriching Reverse Engineering through Visual Exploration of Android Binaries. In *ACM PPREW*.

[363] D. Javaheri and M. Hosseinzadeh. 2017. A Framework for Recognition and Confronting of Obfuscated Malwares Based on Memory Dumping and Filter Drivers. *Wireless Personal Communications*, 98, 1, 119–137.

[364] J. Jiang, G. Li, et al. 2020. Similarity of Binaries Across Optimization Levels and Obfuscation. In *ESORICS*, 295–315.

[365] Y. Jiang, R. Li, et al. 2020. AOMDroid: Detecting Obfuscation Variants of Android Malware Using Transfer Learning. In *SecureComm*, 242–253.

[366] M. Jodavi, M. Abadi, et al. 2015. JSObfusDetector: A binary PSO-based one-class classifier ensemble to detect obfuscated JavaScript code. In *AISP*, 322–327.

[367] B. Johansson, P. Lantz, et al. 2017. Lightweight Dispatcher Constructions for Control Flow Flattening. In *ACM SSPREW*, 2.

[368] L. Jones, D. Christman, et al. 2018. ByteWise: A case study in neural network obfuscation identification. In *IEEE CCWC*, 155–164.

[369] H. P. Joshi, A. Dhanasekaran, et al. 2015. Impact of software obfuscation on susceptibility to Return-Oriented Programming attacks. In *IEEE Sarnoff Symposium*, 161–166.

[370] S. Josse. 2014. Malware Dynamic Recompilation. In *HICSS*, 5080–5089.

[371] P. Junod, J. Rinaldini, et al. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *IEEE/ACM SPRO*, 3–9.

[372] A. Kalysch, J. Götzfried, et al. 2017. Vmattack: Deobfuscating virtualization-based packed binaries. In *ARES*, 2.

[373] P. Kanani, K. Srivastava, et al. 2017. Obfuscation: Maze of code. In *CSCITA*, 11–16.

[374] M. G. Kang, P. Poosankam, et al. 2007. Renovo: A Hidden Code Extractor for Packed Executables. In *ACM WORM*, 46–53.

[375] S. Kang, S. Lee, et al. 2021. OBFUS: An Obfuscation Tool for Software Copyright and Vulnerability Protection. In *ACM CODASPY*, 309–311.

[376] Y. Kanzaki, A. Monden, et al. 2015. Code Artificiality: A Metric for the Code Stealth Based on an N-Gram Model. In *IEEE/ACM SPRO*, 31–37.

[377] Y. Kanzaki, A. Monden, et al. 2003. Exploiting Self-Modification Mechanism for Program Protection. In *IEEE COMPSAC*, 170.

[378] Y. Kanzaki, C. Thomborson, et al. 2015. Pinpointing and Hiding Surprising Fragments in an Obfuscated Program. In *ACM PPREW*.

[379] U. Kargén, I. Härnqvist, et al. 2022. desync-cc: An Automatic Disassembly-Desynchronization Obfuscator. In *IEEE SANER*, 464–468.

[380] M. Karnick, J. MacBride, et al. 2006. A Qualitative Analysis of Java Obfuscation. In *IASTED SEA*.

[381] A. Karnik, S. Goswami, et al. 2007. Detecting Obfuscated Viruses Using Cosine Similarity Analysis. In *AMS*, 165–170.

[382] R. Kaur, Y. Ning, et al. 2018. Unmasking Android Obfuscation Tools Using Spatial Analysis. In *PST*, 1–10.

[383] I. Khairunisa and H. Kabetta. 2021. PHP Source Code Protection Using Layout Obfuscation and AES-256 Encryption Algorithm. In *IWBIS*, 133–138.

[384] S. Khalid and F. B. Hussain. 2022. Evaluating Opcodes for Detection of Obfuscated Android Malware. In *ICAIIC*, 044–049.

[385] K. Khanmohammadi and A. Hamou-Lhadj. 2017. HyDroid: A Hybrid Approach for Generating API Call Traces from Obfuscated Android Applications for Mobile Security. In *QRS*, 168–175.

[386] D. Kholia and P. Węgrzyn. 2013. Looking Inside the (Drop) Box. In *USENIX WOOT*.

[387] D. Kim, A. Majlesi-Kupaei, et al. 2017. DynODet: Detecting Dynamic Obfuscation in Malware. In *DIMVA*, 97–118.

[388] J. Kim, S. Kang, et al. 2021. LOM: Lightweight Classifier for Obfuscation Methods. In *WISA*, 3–15.

[389] J. Kim, K. Ko, et al. 2018. Building the De-obfuscation Platform Based on LLVM. In *CSA-CUTE*, 1269–1274.

[390] S. Kim, S. Hong, et al. 2018. Obfuscated VBA Macro Detection Using Machine Learning. In *IEEE/IFIP DSN*, 490–501.

[391] J. Kinder. 2012. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *WCRE*, 61–70.

[392] J. Kinder, S. Katzenbeisser, et al. 2005. Detecting Malicious Code by Model Checking. In *DIMVA*, 174–187.

[393] J. Kirsch, C. Jonischkeit, et al. 2017. Combating Control Flow Linearization. In *SEC*, 385–398.

[394] S. Ko, J. Choi, et al. 2017. COAT: Code Obfuscation Tool to Evaluate the Performance of Code Plagiarism Detection Tools. In *ICSSA*, 32–37.

[395] C. Kolbitsch, E. Kirda, et al. 2011. The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code. In *ACM CCS*, 285–296.

[396] A. Kovacheva. 2013. Efficient Code Obfuscation for Android. In *IAIT*, 104–119.

[397] K. Kuang, Z. Tang, et al. 2016. Exploiting dynamic scheduling for VM-based code obfuscation. In *IEEE Trustcom/BigDataSE/ISPA*, 489–496.

[398] A. Kulkarni and R. Metta. 2014. A Code Obfuscation Framework Using Code Clones. In *IEEE ICPC*, 295–299.

[399] A. Kulkarni and R. Metta. 2014. A New Code Obfuscation Scheme for Software Protection. In *IEEE SOSE*, 409–414.

[400] S. Kumar, D. Mishra, et al. 2022. AndroOBFS: Time-tagged Obfuscated Android Malware Dataset with Family Information. In *IEEE/ACM MSR*, 454–458.

[401] A. Lakhotia, D. R. Boccardo, et al. 2010. Context-Sensitive Analysis of Obfuscated X86 Executables. In *ACM PEPM*, 131–140.

[402] A. Lakhotia, D. R. Boccardo, et al. 2010. Context-Sensitive Analysis without Calling-Context. *Higher Order Symbol. Comput.*, 23, 3, 275–313.

[403] A. Lakhotia, E. U. Kumar, et al. 2005. A Method for Detecting Obfuscated Calls in Malicious Binaries. *IEEE Trans. Softw. Eng.*, 31, 955–968, 11.

[404] A. Lakhotia and U. K. Kumar. 2004. Abstracting Stack to Detect Obfuscated Calls in Binaries. In *IEEE SCAM*, 17–26.

[405] P. Lan, P. Wang, et al. 2018. Lambda obfuscation. In *SecureComm*, 206–224.

[406] T. László and Á. Kiss. 2009. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica*, 30, 3–19.

[407] C. LeDoux, M. Sharkey, et al. 2012. Instruction Embedding for Improved Obfuscation. In *ACM-SE*, 130–135.

[408] J.-Y. Lee, J. H. Suk, et al. 2018. VODKA: Virtualization Obfuscation Using Dynamic Key Approach. In *ISA*, 131–145.

[409] W. Y. Lee, J. Saxe, et al. 2019. SeqDroid: Obfuscated Android Malware Detection Using Stacked Convolutional and Recurrent Neural Networks. In *Deep Learning Applications for Cyber Security*, 197–210.

[410] H. Li, Y. Zhang, et al. 2020. SymSem: Symbolic Execution with Time Stamps for Deobfuscation. In *Inscrypt*.

[411] J. Li, M. Xu, et al. 2009. Malware Obfuscation Detection via Maximal Patterns. In *IITA*, 324–328.

[412] Y. Li, Z. Sha, et al. 2021. Code Obfuscation Based on Inline Split of Control Flow Graph. In *ICAICA*, 632–638.

[413] Z. Li, Q. A. Chen, et al. 2019. Effective and Light-Weight Deobfuscation and Semantic-Aware Attack Detection for PowerShell Scripts. In *ACM CCS*, 1831–1847.

[414] Z. Li, J. Sun, et al. 2019. Obfusifier: Obfuscation-resistant Android malware detection system. In *SecureComm*, 214–234.

[415] M. Liang, Z. Li, et al. 2018. Deobfuscation of Virtualization-Obfuscated Code Through Symbolic Execution and Compilation Optimization. In *ICISC*, 313–324.

[416] Z. Liang, W. Li, et al. 2017. A parameterized flattening control flow based obfuscation algorithm with opaque predicate for reduplicate obfuscation. In *PIC*, 372–378.

[417] H.-T. Liaw and S.-C. Wei. 2014. Obfuscation for object-oriented programs: dismantling instance methods. *Software: Practice and Experience*, 44, 9, 1077–1104.

[418] C. Liu, B. Xia, et al. 2018. PSDEM: A Feasible De-Obfuscation Method for Malicious PowerShell Detection. In *IEEE ISCC*, 825–831.

[419] W. Liu and W. Li. 2016. Unifying the method descriptor in Java obfuscation. In *ICCC*, 1397–1401.

[420] Z. Liu, D. Zheng, et al. 2021. VABox: A Virtualization-Based Analysis Framework of Virtualization-Obfuscated Packed Executables. In *ICAIS*, 73–84.

[421] G. Lu and S. Debray. 2012. Automatic Simplification of Obfuscated JavaScript Code: A Semantics-Based Approach. In *IEEE SERE*, 31–40.

[422] K. Z. M. Lu. 2019. Control Flow Obfuscation via CPS Transformation. In *ACM PEPM*, 54–60.

[423] W. Lu, B. Sistany, et al. 2020. Towards Formal Verification of Program Obfuscation. In *IEEE EuroS&P Workshops*, 635–644.

[424] Ď. Lukáš and K. Dušan. 2012. C Source Code Obfuscator. In *Kybernetika* number 3. Volume 48, 494–501.

[425] M. Lupascu, D. T. Gavrilut, et al. 2018. An Overview of Obfuscation Techniques used by Malware in Visual Basic for Application Scripts. In *SYNASC*, 280–287.

[426] H. Ma, R. Li, et al. 2016. Integrated Software Fingerprinting via Neural-Network-Based Control Flow Obfuscation. *IEEE Trans. Inf. For. Sec.*, 11, 10, 2322–2337.

[427] Y. Ma, Y. Li, et al. 2021. A Classic Multi-method Collaborative Obfuscation Strategy. In *DMBD*, 90–97.

[428] J. Macbride, C. Mascioli, et al. 2005. A comparative Study of Java Obfuscators. In *IASTED SEA*.

[429] M. Madou, B. Anckaert, et al. 2005. Hybrid Static-Dynamic Attacks against Software Protection Mechanisms. In *ACM DRM*, 75–82.

[430] M. Madou, B. Anckaert, et al. 2006. On the Effectiveness of Source Code Transformations for Binary Obfuscation. In *SERP*, 527–533.

[431] M. Madou, B. Anckaert, et al. 2006. Software Protection Through Dynamic Code Mutation. In *WISA*, 194–206.

[432] M. Madou, L. Van Put, et al. 2006. Understanding Obfuscated Code. In *IEEE ICPC*, 268–274.

[433] A. Majlesi-Kupaei, D. Kim, et al. 2021. RL-BIN++: Overcoming Binary Instrumentation Challenges in the Presence of Obfuscation Techniques and Problematic Features. In *ICSCA*, 262–272.

[434] A. Majumdar, A. Monsifrot, et al. 2006. On Evaluating Obfuscatory Strength of Alias-based Transforms using Static Analysis. In *ACC*, 605–610.

[435] A. Majumdar and C. Thomborson. 2006. Manufacturing Opaque Predicates in Distributed Systems for Code Obfuscation. In *ACSC*, 187–196.

[436] A. Majumdar and C. Thomborson. 2005. Securing Mobile Agents Control Flow Using Opaque Predicates. In *KES*, 1065–1071.

[437] T. Mantoro, D. Stephen, et al. 2022. Malware Detection with Obfuscation Techniques on Android Using Dynamic Analysis. In *ICCED*, 1–6.

[438] G. Marco, M. Andrea, et al. 2016. Challenging Anti-virus Through Evolutionary Malware Obfuscation. In *EvoApplications*, 149–162.

[439] F. Martinelli, F. Mercaldo, et al. 2018. Evaluating model checking for cyber threats code obfuscation identification. *Journal of Parallel and Distributed Computing*, 119, 203–218.

[440] M. Maskur, Z. Sari, et al. 2018. Implementation of Obfuscation Technique on PHP Source Code. In *EECSI*, 738–742.

[441] J. Mason, S. Small, et al. 2009. English Shellcode. In *ACM CCS*, 524–533.

[442] 2021. *A Malware Obfuscation AI Technique to Evade Antivirus Detection in Counter Forensic Domain. Enabling AI Applications in Data Science*, 597–615.

[443] J. T. McDonald and A. Yasinsac. 2007. Applications for Provably Secure Intent Protection with Bounded Input-Size Programs. In *ARES*, 286–293.

[444] J. M. Memon, Shams-ul-Arfeen, et al. 2006. Preventing Reverse Engineering Threat in Java Using Byte Code Obfuscation Techniques. In *ICET*, 689–694.

[445] A. Mezina and R. Burget. 2022. Obfuscated malware detection using dilated convolutional network. In *ICUMT*, 110–115.

[446] S. Millar, N. McLaughlin, et al. 2020. DANdroid: A Multi-View Discriminative Adversarial Network for Obfuscated Android Malware Detection. In *ACM CODASPY*, 353–364.

[447] J. Ming, D. Xu, et al. 2017. Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. In *USENIX Security*, 253–270.

[448] J. Ming, D. Xu, et al. 2015. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *ACM CCS*, 757–768.

[449] R. Mohsen and A. M. Pinto. 2016. Evaluating Obfuscation Security: A Quantitative Approach. In *FPS*. Volume 9482, 174–192.

[450] A. K. Mondal, C. Roy, et al. 2019. Automatic Components Separation of Obfuscated Android Applications: An Empirical Study of Design Based Features. In *IEEE/ACM ASEW*, 23–28.

[451] A. Monden, A. Monsifrot, et al. 2004. A Framework for Obfuscated Interpretation. In *ACSW Frontiers*, 7–16.

[452] S. Morishige, S. Haruta, et al. 2017. Obfuscated malicious javascript detection scheme using the feature based on divided URL. In *APCC*, 1–6.

[453] A. Moser, C. Kruegel, et al. 2007. Limits of Static Analysis for Malware Detection. In *ACM ACSAC*, 421–430.

[454] D. Mu, J. Guo, et al. 2018. ROPOB: Obfuscating Binary Code via Return Oriented Programming. In *SecureComm*, 721–737.

[455] V. Mukhin, Y. Kornaga, et al. 2020. Obfuscation Code Technics Based on Neural Networks Mechanism. In *IEEE SAIC*, 1–6.

[456] 2022. *Program Code Protecting Mechanism Based on Obfuscation Tools. SAIC*, 407–419.

[457] D. C. Mumme, B. Wallace, et al. 2017. Cloud Security via Virtualized Out-of-Band Execution and Obfuscation. In *IEEE CLOUD*, 286–293.

[458] S. A. Musavi and M. Kharrazi. 2014. Back to static analysis for kernel-level rootkit detection. *IEEE Trans. Inf. For. Sec.*, 9, 9, 1465–1476.

[459] A. Naderi-Afooshteh, Y. Kwon, et al. 2019. Cubismo: Decloaking Server-Side Malware via Cubist Program Analysis. In *ACM ACSAC*, 430–443.

[460] N. A. Naeem, M. Batchelder, et al. 2007. Metrics for Measuring the Effectiveness of Decompilers and Obfuscators. In *IEEE ICPC*, 253–258.

[461] V. Nagarajan, R. Gupta, et al. 2007. Matching control flow of program versions. In *ICSM*, 84–93.

[462] S. Naval, V. Laxmi, et al. 2012. SPADE: Signature Based PAcker DEtection. In *SecurIT*, 96–101.

[463] T. Nguyen, J. Mcdonald, et al. 2020. Detecting Repackaged Android Applications Using Perceptual Hashing. In *HICSS*, 6641–6650.

[464] M. Nouman Ahmed and M. e. Hani Tayyab. 2022. Automating the Process of Developing Obfuscated Variants of PE Through ADOPE Software. In *ICCWS*, 36–41.

[465] P. O'kane, S. Sezer, et al. 2016. Detecting obfuscated malware using reduced opcode set and optimised runtime trace. *Security Informatics*, 5, 1.

[466] T. Ogiso, Y. Sakabe, et al. 2003. Software obfuscation on a theoretical basis and its implementation. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 86, 1, 176–186.

[467] S. P. Ohello and Suhardi. 2022. Android Malware Evasion Framework For Auditing Anti-Malware Resistance Against Various Obfuscation Technique And Dynamic Code Loading. In *ICITSI*, 183–188.

[468] R. Omar, A. El-Mahdy, et al. 2014. Arbitrary Control-Flow Embedding into Multiple Threads for Obfuscation: A Preliminary Complexity and Performance Analysis. In *SCC*, 51–58.

[469] N. Otsuki and H. Tamada. 2022. Overcoming the Obfuscation Method of the Dynamic Name Resolution. In *ICSIM*, 118–124.

[470] J. Palsberg, S. Krishnaswamy, et al. 2000. Experience with Software Watermarking. In *ACM ACSAC*, 308–316.

[471] D. Park, H. Khan, et al. 2019. Generation & Evaluation of Adversarial Examples for Malware Obfuscation. In *ICMLA*, 1283–1290.

[472] D. Park, H. Powers, et al. 2020. Towards Obfuscated Malware Detection for Low Powered IoT Devices. In *ICMLA*, 1073–1080.

[473] S.-M. Park, H.-C. Bae, et al. 2017. Code Modification and Obfuscation Detection Test Using Malicious Script Distributing Website Inspection Technology. In *CSA-CUTE*, 74–80.

[474] R. Paul, H. Mitch, et al. 2006. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *ACM ACSAC*, 289–300.

[475] A. Pawlowski, M. Contag, et al. 2016. Probfuscation: An Obfuscation Approach Using Probabilistic Control Flows. In *DIMVA*, 165–185.

[476] Y. Peng, Y. Chen, et al. 2019. An Adaptive Approach to Recommending Obfuscation Rules for Java Bytecode Obfuscators. In *COMPSAC*, 97–106.

[477] Y. Peng, J. Liang, et al. 2016. A control flow obfuscation method for Android applications. In *CCIS*, 94–98.

[478] Y. Peng, G. Su, et al. 2017. Control flow obfuscation based protection method for Android applications. *China Communications*, 14, 11, 247–259.

[479] D.-P. Pham, D. Marion, et al. 2021. Obfuscation Revealed: Leveraging Electromagnetic Signals for Obfuscated Malware Classification. In *ACM ACSAC*, 706–719.

[480] Y. Piao, J.-H. Jung, et al. 2016. Server-based code obfuscation scheme for APK tamper detection. *Security and Communication Networks*, 9, 6, 457–467.

[481] D. Pizzolotto and M. Ceccato. 2018. Obfuscating Java Programs by Translating Selected Portions of Bytecode to Native Libraries. In *IEEE SCAM*, 40–49.

[482] D. Pizzolotto, R. Fellin, et al. 2019. OBLIVE: Seamless Code Obfuscation for Java Programs and Android Apps. In *IEEE SANER*, 629–633.

[483] G. S. Ponomarenko and P. G. Klyucharev. 2022. On improvements of robustness of obfuscated JavaScript code detection. *J. Comput. Virol. and Hack. Tech.*

[484] S. Praveen and P. Sojan Lal. 2007. Array Data Transformation for Source Code Obfuscation. *International Journal of Computer and Information Engineering*, 1, 12.

[485] R. Qin and H. Han. 2021. BinSEAL: Linux Binary Obfuscation Against Symbolic Execution. In *SpaCCS*, 65–76.

[486] S. Qing, W. Zhi-yue, et al. 2012. Technique of source code obfuscation based on data flow and control flow tansformations. In *ICCSE*, 1093–1097.

[487] J. Qiu, B. Yadegari, et al. 2014. A Framework for Understanding Dynamic Anti-Analysis Defenses. In *ACM PPREW*.

[488] M. Rabih and P. Alexandre Miranda. 2016. Theoretical Foundation for Code Obfuscation Security: A Kolmogorov Complexity Approach. In *ICETE*, 245–269.

[489] V. Rastogi, Y. Chen, et al. 2013. DroidChameleon: Evaluating Android Anti-Malware against Transformation Attacks. In *ACM Asia CCS*, 329–334.

[490] S. Rauti and V. Leppänen. 2018. A Comparison of Online JavaScript Obfuscators. In *ICSSA*, 7–12.

[491] S. Rauti and V. Leppänen. 2014. A Proxy-Like Obfuscator for Web Application Protection. *Int. J. Inf. Tech. & Secur.*, 6, 1.

[492] R. Rolles. 2009. Unpacking Virtualization Obfuscators. In *USENIX WOOT*, 1–7.

[493] A. Romano, D. Lehmann, et al. 2022. Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. In *IEEE S&P*, 1574–1589.

[494] M. O. F. K. Russel, S. S. M. M. Rahman, et al. 2020. A large-scale investigation to identify the pattern of permissions in obfuscated Android malwares. In *ICONCS*, 85–97.

[495] M. O. F. K. Russel, S. S. M. M. Rahman, et al. 2020. A Large-Scale Investigation to Identify the Pattern of App Component in Obfuscated Android Malwares. In *MIND*, 513–526.

[496] 2021. *AndroShow: A Large Scale Investigation to Identify the Pattern of Obfuscated Android Malware. Machine Intelligence and Big Data Analytics for Cybersecurity Applications*, 191–216.

[497] H. Sagisaka and H. Tamada. 2016. Identifying the applied obfuscation method towards de-obfuscation. In *IEEE/ACIS ICIS*, 1–6.

[498] H. Saıdi, P. Porras, et al. 2010. Experiences in malware binary deobfuscation. *Virus Bulletin*.

[499] Y. Sakabe, M. Soshi, et al. 2005. Java Obfuscation Approaches to Construct Tamper-Resistant Object-Oriented Programs. *IPSJ Digital Courier*, 1, 349–361.

[500] Y. Sakabe, M. Soshi, et al. 2003. Java Obfuscation with a Theoretical Basis for Building Secure Mobile Agents. In *CMS*, 89–103.

[501] H. Salazar and C. Barria. 2021. Classification and Update Proposal for Modern Computer Worms, Based on Obfuscation. In *ITNG*, 49–57.

[502] H. Salazar and C. Barría. 2022. Construction of a Technological Component to Support ISMS for the Detection of Obfuscation in Computer Worm Samples. In *WITCOM*, 215–224.

[503] J. Salwan, S. Bardin, et al. 2018. Symbolic deobfuscation: From virtualized code back to the original. In *DIMVA*, 372–392.

[504] S. Schrittwieser, S. Katzenbeisser, et al. 2014. AES-SEC: Improving Software Obfuscation through Hardware-Assistance. In *ARES*, 184–191.

[505] S. Schrittwieser and S. Katzenbeisser. 2011. Code Obfuscation against Static and Dynamic Reverse Engineering. In *Information Hiding*, 270–284.

[506] S. Schrittwieser, S. Katzenbeisser, et al. 2014. Covert Computation - Hiding code in code through compile-time obfuscation. *Comput. & Secur.*, 42, 13–26.

[507] V. Šembera, M. Paquet-Clouston, et al. 2021. Cybercrime Specialization: An Exposé of a Malicious Android Obfuscation-as-a-Service. In *IEEE EuroS&P Workshops*, 213–226.

[508] A. Sengupta and S. Sivasankari. 2021. A Novel Approach for Analysing and Detection of Obfuscated Malware Payloads in Android Platform Using DexMonitor. In *ICACT*, 1–9.

[509] T. Seto, A. Monden, et al. 2019. On Preventing Symbolic Execution Attacks by Low Cost Obfuscation. In *IEEE/ACIS SNPD*, 495–500.

[510] M. Sewak, S. K. Sahay, et al. 2021. ADVERSARIALuscator: An Adversarial-DRL based Obfuscator and Metamorphic Malware Swarm Generator. In *IJCNN*, 1–9.

[511] M. Sewak, S. K. Sahay, et al. 2020. DOOM: A Novel Adversarial-DRL-Based Op-Code Level Metamorphic Malware Obfuscator for the Enhancement of IDS. In *ACM UbiComp-ISWC*, 131–134.

[512] Y. Shah, J. Shah, et al. 2018. Code obfuscating a Kotlin-based App with Proguard. In *ICAECC*, 1–5.

[513] M. Sharif, A. Lanzi, et al. 2009. Automatic Reverse Engineering of Malware Emulators. In *IEEE S&P*, 94–109.

[514] M. Sharif, A. Lanzi, et al. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation. In *NDSS*.

[515] M. Sharif, V. Yegneswaran, et al. 2008. Eureka: A Framework for Enabling Static Malware Analysis. In *ESORICS*, 481–500.

[516] Z. Shehu, C. Ciccotelli, et al. 2016. Towards the Usage of Invariant-Based App Behavioral Fingerprinting for the Detection of Obfuscated Versions of Known Malware. In *NGMAST*, 121–126.

[517] O. Shevtsova and D. N. Buintsev. 2009. Methods and software for the program obfuscation. In *SIBCON*, 113–115.

[518] M. Sidhardhan and K. Praveen. 2019. Data Obfuscation Using Secret Sharing. In *ICAICR*, 183–191.

[519] P. Sivadasan, P. SojanLal, et al. 2009. JDATATRANS for array obfuscation in Java source codes to defeat reverse engineering from decompiled codes. In *COMPUTE*, 1–4.

[520] M. Sosonkin, G. Naumovich, et al. 2003. Obfuscation of Design Intent in Object-Oriented Applications. In *ACM DRM*, 142–153.

[521] J. Su, D. V. Vargas, et al. 2017. Evasion Attacks Against Statistical Code Obfuscation Detectors. In *IWSEC*, 121–137.

[522] J. Su, K. Yoshioka, et al. 2016. An Efficient Method for Detecting Obfuscated Suspicious JavaScript Based on Text Pattern Analysis. In *ACM WTMC*, 3–11.

[523] J. Su, K. Yoshioka, et al. 2016. Detecting Obfuscated Suspicious JavaScript Based on Information-Theoretic Measures and Novelty Detection. In *ICISC*, 278–293.

[524] G. Suarez-Tangil, S. K. Dash, et al. 2017. DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware. In *ACM CODASPY*, 309–320.

[525] G. Suarez-Tangil, J. E. Tapiador, et al. 2016. Alterdroid: Differential Fault Analysis of Obfuscated Smartphone Malware. *IEEE Trans. Mob. Comp.*, 15, 4, 789–802.

[526] J. H. Suk and D. H. Lee. 2020. VCF: Virtual Code Folding to Enhance Virtualization Obfuscation. *IEEE Access*, 8, 139161–139175.

[527] C. Sun, H. Zhang, et al. 2020. DroidPDF: The Obfuscation Resilient Packer Detection Framework for Android Apps. *IEEE Access*, 8, 167460–167474.

[528] H. Tamada, K. Fukuda, et al. 2012. Program incomprehensibility evaluation for obfuscation methods with queue-based mental simulation model. In *ACIS SNPD*, 498–503.

[529] H. Tamada, M. Nakamura, et al. 2008. Introducing dynamic name resolution mechanism for obfuscating system-defined names in programs. In *IASTED SE* number 598-074, 125–130.

[530] K. Tang, F. Liu, et al. 2021. Anti-obfuscation Binary Code Clone Detection Based on Software Gene. In *Data Science*, 193–208.

[531] Y. Tang and S. Chen. 2007. An Automated Signature-Based Approach against Polymorphic Internet Worms. *IEEE Trans. Par. Distr. Sys.*, 18, 879–892, 7.

[532] Z. Tang, X. Chen, et al. 2009. Research on Java software protection with the obfuscation in identifier renaming. In *ICICIC*, 1067–1071.

[533] Z. Tang, K. Kuang, et al. 2017. Seead: A semantic-based approach for automatic binary code de-obfuscation. In *2017 IEEE Trustcom/BigDataSE/ICESS*, 261–268.

[534] Z. Tang, M. Li, et al. 2018. VMGuards: A novel virtual machine based code protection system with VM security as the first class design concern. *Applied Sciences*, 8, 5, 771.

[535] R. Tiella and M. Ceccato. 2017. Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation. In *IEEE SANER*, 182–192.

[536] R. Tofighi-Shirazi, M. Christofi, et al. 2018. DoSE: Deobfuscation based on Semantic Equivalence. In *SSPREW*, 1.

[537] S. Treadwell and M. Zhou. 2009. A heuristic approach for detection of obfuscated malware. In *IEEE ISI*, 291–299.

[538] H.-Y. Tsai, Y.-L. Huang, et al. 2009. A Graph Approach to Quantitative Analysis of Control-Flow Obfuscating Transformations. *IEEE Trans. Inf. For. Sec.*, 4, 2, 257–267.

[539] N. G. Tsoutsos and M. Maniatakos. 2017. Obfuscating branch decisions based on encrypted data using MISR and hash digests. In *AsianHOST*, 115–120.

[540] S. K. Udupa, S. K. Debray, et al. 2005. Deobfuscation: Reverse Engineering Obfuscated Code. In *WCRE*, 45–54.

[541] J. Van den Broeck, B. Coppens, et al. 2022. Flexible Software Protection. *Comput. & Secur.*, 116, C, 102636.

[542] M. Venable, M. R. Chouchane, et al. 2005. Analyzing Memory Accesses in Obfuscated x86 Executables. In *DIMVA*, 1–18.

[543] S.-C. Viţel, M. Lupaşcu, et al. 2022. Evolution of macro VBA obfuscation techniques. In *SIN*, 1–8.

[544] L. Vokorokos, Z. Dankovičová, et al. 2017. Using of the forensic analyzing tools, code obfuscation. In *IEEE SAMI*, 000033–000036.

[545] L. Vokorokos, M. Uchnar, et al. 2016. The obfuscation efficiency measuring schemes. In *IEEE INES*, 125–130.

[546] C. Wang, Z. Zhang, et al. 2018. Binary Obfuscation Based Reassemble. In *MALWARE*, 153–160.

[547] C. Wang, J. Davidson, et al. 2001. Protection of Software-based Survivability Mechanisms. In *DSN*, 193–202.

[548] H. Wang, S. Wang, et al. 2022. Generating Effective Software Obfuscation Sequences With Reinforcement Learning. *IEEE Trans. Dep. Sec. Comp.*, 19, 3, 1900–1917.

[549] H. Wang, D. Fang, et al. 2013. Nislvmp: Improved virtual machine-based software protection. In *CIS*, 479–483.

[550] H. Wang, D. Fang, et al. 2016. The Research and Discussion on Effectiveness Evaluation of Software Protection. In *CIS*, 628–632.

[551] L. Wang, Y. Li, et al. 2021. An Efficient Control-flow based Obfuscator for Micropython Bytecode. In *ISSSR*, 54–63.

[552] W. Wang, M. Li, et al. 2019. Invalidating Analysis Knowledge for Code Virtualization Protection Through Partition Diversity. *IEEE Access*, 7, 169160–169173.

[553] X. Wang, Y.-C. Jhi, et al. 2008. STILL: Exploit Code Detection via Static Taint and Initialization Analyses. In *ACM ACSAC*, 289–298.

[554] Y. Wang and A. Rountev. 2017. Who Changed You? Obfuscator Identification for Android. In *MOBILESoft*, 154–164.

[555] Y. Wang, S. Wang, et al. 2018. Turing Obfuscation. In *SecureComm*, 225–244.

[556] Y. Wang. 2021. The De-Obfuscation Method in the Static Detection of Malicious PDF Documents. In *ICNISC*, 44–47.

[557] Z. Wang, C. Jia, et al. 2012. Branch Obfuscation Using Code Mobility and Signal. In *IEEE COMPSACW*, 553–558.

[558] Z. Wang, Y. Shan, et al. 2020. Semantic Redirection Obfuscation: A Control flow Obfuscation Based on Android Runtime. In *TrustCom*, 1756–1763.

[559] M. Webster and G. Malcolm. 2009. Detection of metamorphic and virtualization-based malware using algebraic specification. *J. in Comput. Virol.*, 5, 221–245, 3.

[560] Y. Y. Wei and K. Ohzeki. 2010. Obfuscation methods with controlled calculation amounts and table function. In *IMCSIT*, 775–780.

[561] M. Y. Wong and D. Lie. 2018. Tackling runtime-based obfuscation in Android with TIRO. In *USENIX Security*, 1247–1262.

[562] B. Wood and A. Azim. 2021. A Novel Technique for Control Flow Obfuscation in JVM Applications Using InvokeDynamic with Native Bootstrapping. In *CASCON*, 232–236.

[563] J. Wu and A. Kanai. 2021. Utilizing obfuscation information in deep learning-based Android malware detection. In *IEEE COMPSAC*, 1321–1326.

[564] M. Wu, Y. Zhang, et al. 2016. Binary Protection Using Dynamic Fine-Grained Code Hiding and Obfuscation. In *ICINS*, 1–8.

[565] Y. Wu, V. Suhendra, et al. 2016. Obfuscating Software Puzzle for Denial-of-Service Attack Mitigation. In *IEEE iThings, GreenCom, CPSCom, and SmartData*, 115–122.

[566] F. Xiang, D. Gong, et al. 2019. Enhanced Branch Obfuscation Based on Exception Handling and Encrypted Mapping Table. In *ACM Turing Celebration Conference - China.*

[567] D. Xiao, S. Bai, et al. 2019. Obfuscation Algorithms Based on Congruence Equation and Knapsack Problem. In *EIDWT*, 128–136.

[568] X. Xie, F. Liu, et al. 2014. Mixed Obfuscation of Overlapping Instruction and Self-Modify Code Based on Hyper-Chaotic Opaque Predicates. In *CIS*, 524–528.

[569] X. Xie, F. Liu, et al. 2014. A Data Obfuscation Based on State Transition Graph of Mealy Automata. In *ICIC*, 520–531.

[570] X. Xie, B. Lu, et al. 2016. Random table and hash coding-based binary code obfuscation against stack trace analysis. *IET Inf. Sec.*, 10, 1, 18–27.

[571] W. Xiong, A. Schaller, et al. 2020. Software Protection Using Dynamic PUFs. *IEEE Trans. Inf. For. Sec.*, 15, 2053–2068.

[572] D. Xu, J. Ming, et al. 2016. Generalized Dynamic Opaque Predicates: A New Control Flow Obfuscation Method. In *ISC*, 323–342.

[573] D. Xu, J. Ming, et al. 2018. VMHunt: A Verifiable Approach to Partially-Virtualized Binary Code Simplification. In *ACM CCS*, 442–458.

[574] H. Xu, Y. Zhou, et al. 2018. Manufacturing Resilient Bi-Opaque Predicates Against Symbolic Execution. In *IEEE/IFIP DSN*, 666–677.

[575] H. Xu, Y. Zhou, et al. 2016. N-Version Obfuscation. In *ACM CPSS*, 22–33.

[576] M. Xu, L. Wu, et al. 2013. A Similarity Metric Method of Obfuscated Malware Using Function-Call Graph. *J. Comput. Virol.*, 9, 1, 35–47.

[577] X. Xu, S. Liu, et al. 2021. Research on PowerShell Obfuscation Technology Based on Abstract Syntax Tree Transformation. In *ISCTIS*, 121–125.

[578] C. Xue, Z. Tang, et al. 2018. Exploiting code diversity to enhance code virtualization protection. In *ICPADS*, 620–627.

[579] K. Yadav, R. Kamble, et al. 2022. Source Code Obfuscation: Novel Technique and Implementation. In *ICTICT4SD*, 197–204.

[580] B. Yadegari and S. Debray. 2014. Bit-Level Taint Analysis. In *IEEE SCAM*, 255–264.

[581] B. Yadegari and S. Debray. 2015. Symbolic Execution of Obfuscated Code. In *ACM CCS*, 732–744.

[582] B. Yadegari, B. Johannesmeyer, et al. 2015. A generic approach to automatic deobfuscation of executable code. In *IEEE S&P*, 674–691.

[583] H. Yamauchi, Y. Kanzaki, et al. 2006. Software obfuscation from crackers' viewpoint. In *ACST*, 286–291.

[584] L. Yang and H.-j. He. 2012. Research on Java Bytecode Parse and Obfuscate Tool. In *CSSS*, 50–53.

[585] X. Yang, L. Zhang, et al. 2019. Android Control Flow Obfuscation Based on Dynamic Entry Points Modification. In *CSCS*, 296–303.

[586] X. Yao, J. Pang, et al. 2012. A method and implementation of control flow obfuscation using SEH. In *MINES*, 336–339.

[587] X. Yao, B. Li, et al. 2018. A User-Defined Code Reinforcement Technology Based on LLVM-Obfuscator. In *CSA-CUTE*, 688–694.

[588] D. Yi. 2009. A New Obfuscation Scheme in Constructing Fuzzy Predicates. In *WCSE*, 379–382.

[589] J. Yi, L. Chen, et al. 2020. A Security Model and Implementation of Embedded Software Based on Code Obfuscation. In *TrustCom*, 1606–1613.

[590] G. You, G. Kim, et al. 2022. Deoptfuscator: Defeating Advanced Control-Flow Obfuscation Using Android Runtime (ART). *IEEE Access*, 10, 61426–61440.

[591] C. Yuan, S. Wei, et al. 2017. Scalable and Obfuscation-Resilient Android App Repackaging Detection Based on Behavior Birthmark. In *APSEC*, 476–485.

[592] L. Yuan. 2016. Detecting similar components between android applications with obfuscation. In *ICCSNT*, 186–190.

[593] K. Yuhei, S. Eitaro, et al. 2017. Stealth Loader: Trace-Free Program Loading for API Obfuscation. In *RAID*, 217–237.

[594] Z. Yujia and P. Jianmin. 2016. A New Compile-Time Obfuscation Scheme for Software Protection. In *CyberC*, 1–5.

[595] A. Zambon. 2012. Aucsmith-Like Obfuscation of Java Bytecode. In *IEEE SCAM*, 114–119.

[596] V. Željko, H. Pål, et al. 2010. Program Obfuscation by Strong Cryptography. In *ARES*, 242–247.

[597] L. Zhang, H. Meng, et al. 2018. Progressive Control Flow Obfuscation for Android Applications. In *IEEE TENCON*, 1075–1079.

[598] X. Zhang, J. Wang, et al. 2020. AndrOpGAN: An Opcode GAN for Android Malware Obfuscations. In *ML4CS*, 12–25.

[599] Y. Zhang, G. Xiao, et al. 2020. An Empirical Study of Code Deobfuscations on Detecting Obfuscated Android Piggybacked Apps. In *APSEC*, 41–50.

[600] M. Zhao. 2018. Detection of Android Malicious Obfuscation Applications Based on Multi-Class Features. In *IMCCC*, 1795–1799.

[601] Y. Zhou, A. Main, et al. 2007. Information Hiding in Software with Mixed Boolean-Arithmetic Transforms. In *WISA*, 61–75.

[602] W. Zhu, C. Thomborson, et al. 2006. Obfuscate Arrays by Homomorphic Functions. In *IEEE GRC*, 770–773.

[603] X. Zhuang, T. Zhang, et al. 2004. Hardware Assisted Control Flow Obfuscation for Embedded Processors. In *CASES*, 292–302.

[604] Y. Zhuang. 2018. The performance cost of software obfuscation for Android applications. *Comput. & Secur.*, 73, 57–72.