Defeating Program Analysis Techniques via Ambiguous Translation

Chijung Jung[*], Doowon Kim[†], Weihang Wang[‡], Yunhui Zheng[§], Kyu Hyung Lee[¶], and Yonghwi Kwon[*]
[*]*Department of Computer Science, University of Virginia*, Charlottesville, VA, USA
{cj5kd, yongkwon}@virginia.edu
[†]*Department of Electrical Engineering and Computer Science, University of Tennessee*, Knoxville, TN, USA
doowon@utk.edu
[‡]*Department of Computer Science and Engineering, University at Buffalo*, Buffalo, NY, USA
weihangw@buffalo.edu
[§]*IBM T.J. Watson Research Center*, Yorktown Heights, NY, USA
zhengyu@us.ibm.com
[¶]*Department of Computer Science, University of Georgia*, Athens, GA, USA
kyuhlee@uga.edu

***Abstract*—This research explores the possibility of a new anti-analysis technique, carefully designed to attack weaknesses of the existing program analysis approaches. It encodes a program code snippet to hide, and its decoding process is implemented by a sophisticated state machine that produces multiple outputs depending on inputs. The key idea of the proposed technique is to ambiguously decode the program code, resulting in multiple decoded code snippets that are challenging to distinguish from each other. Our approach is stealthier than previous similar approaches as its execution does not exhibit different behaviors between when it decodes correctly or incorrectly. This paper also presents analyses of weaknesses of existing techniques and discusses potential improvements. We implement and evaluate the proof of concept approach, and our preliminary results show that the proposed technique imposes various new unique challenges to the program analysis technique.**

***Index Terms*—anti-analysis, translation, program analysis**

I. Introduction

Program analysis techniques (e.g., static, dynamic, and symbolic/concolic analyses) have been a key technique to understand the behavior of a suspicious program. Given a program under analysis, program analysis techniques aim to infer behaviors of the program. A typical application in security is to determine whether the program is malicious or not. In particular, static analysis techniques [1–5] analyze target programs without running the code, by parsing the target program's code and data. Dynamic analysis [6–8] runs a target program and observes the runtime behaviors of the program. Techniques such as symbolic analysis [9–13] and forced execution techniques [14–18] aim to solve predicate conditions or forcibly execute branches to increase the coverage.

On the flip side, anti-analysis techniques aim to thwart the program analysis by attacking the analysis techniques' weaknesses. For example, obfuscation changes the static representation of the program to hide its original semantic. Packers compress or encrypt program code and unpacks at runtime. Both techniques limit static/symbolic analysis that do not run the program code. Evasive techniques introduce predicate conditions to make certain code triggered only when the conditions are satisfied, imposing challenges to dynamic and symbolic analyses that depend on the complexity of the execution paths and inputs. While forced execution handles evasive techniques by forcibly executing branches regardless of the conditions, it suffers from the inaccurate execution context caused by the ignored predicate conditions.

Proposed Research. This research explores the possibility of a more advanced anti-analysis technique called *Ambiguous Translation* that imposes significant challenges to the existing analysis techniques. The goal of this idea paper is twofold: (1) understanding existing program analysis techniques limitations and (2) demonstrating that developing an anti-analysis technique exploiting the limitations is possible. The key enabling concept is our *Ambiguous Translator*, which takes any inputs and generates many different outputs. While there can be a few intended outputs, they are challenging to be identified since all the inputs go through the same types of operations and outputs look similar, leading to the extra ambiguity in the translation.

II. Background and Related Work

There has been an arms race between program analysis and anti-analysis techniques. In this section, we describe how program analysis and anti-analysis techniques have evolved.

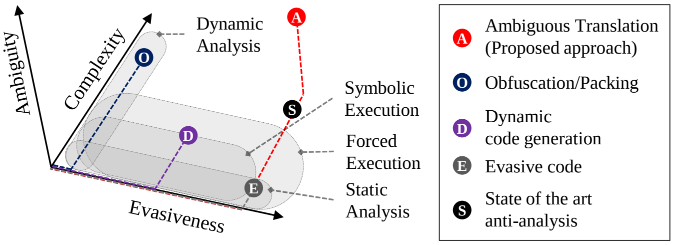


Fig. 1. Program Analysis Techniques and Anti-analysis Techniques in the Challenges for Analysis: Evasiveness, Complexity, and Ambiguity.

Anti-static Analysis Techniques. Obfuscation techniques hide malicious code leveraging various techniques including opaque predicates [19–21], code insertion/replacement [22–26], encryptions [27, 28], hardware primitives [29, 30], and sub-tree embedding [31]. However, since code snippets added for obfuscation (e.g., opaque predicates and dummy code) are non-essential in the program execution, advanced program analysis techniques can be enhanced to detect and remove them [1, 2, 32–34]. Packing is another common tactic to thwart static analysis. To analyze packed programs, it first needs to unpack the programs, which is challenging for static analysis. However, packing is trivially handled by dynamic analysis since the packed program will first unpack itself to run the original program's code. Data obfuscations (e.g., encrypting code sections and decrypting them at runtime) are easily handled by dynamic analysis [6–8].

Anti-dynamic/-symbolic Analysis Techniques. Dynamic analysis techniques run the program code with a concrete input. The analysis result of the executed path is precise while identifying various inputs to achieve high code coverage is challenging. Symbolic and concolic analyses aim to identify inputs that can cover more predicate conditions. However, they have difficulty scaling to large programs. Ollivier et al. [24] present a systematic study of multiple methods to hinder symbolic execution techniques. Most of anti-symbolic analysis techniques try to add additional control flow structures such as predicates loops to increase the number of feasible paths.

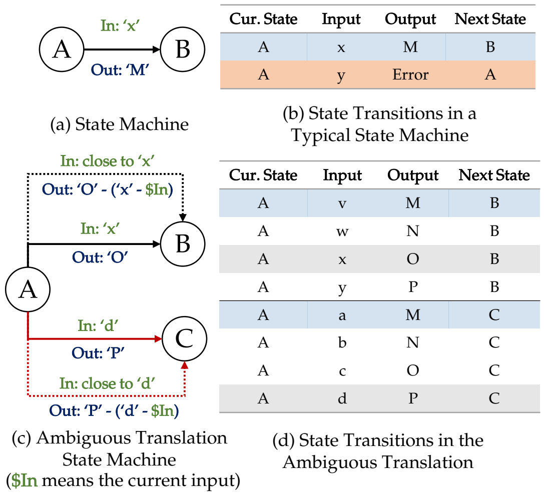
Program Analysis Techniques. Figure 1 presents both program analysis and anti-analysis techniques in three different dimensions. First, obfuscation and packing techniques (Ⓞ) implement complex computations that are difficult to be analyzed by static and symbolic analysis. However, they do not make the target program evasive, hence, dynamic analysis (e.g., running the program code) can effectively defeat them. Second, evasive techniques (Ⓔ) hide malicious code to be only triggered under a certain condition, making dynamic analysis ineffective since it requires an input to trigger the condition. Static analysis techniques are effective in analyzing them since they do not need to know the predicate conditions. Third, dynamic code generation techniques (Ⓓ) are challenging for both dynamic and static analysis because dynamic code can be generated through a particular path (with a certain input) and complex string operations. Fourth, symbolic and concolic analysis can be used to precisely analyze them, while they are computationally expensive, hence difficult to scale. Fifth, forced execution techniques can achieve high code coverage on more complex programs. However, since they forcibly execute program code regardless of the predicate conditions, their analysis results can be inaccurate, leading to false positives/negatives. Ⓢ represents the state-of-the-art anti-analysis techniques. Typically, they aim to make the target program more complex and evasive by carefully combining dynamic code generation and evasive techniques, amplifying challenges to existing analysis techniques.

Proposed Approach: *Ambiguous Translation*. The proposed approach aims to impose significant challenges to all three dimensions in Figure 1, including the new dimension *Ambiguity*. Specifically, we introduce extra *ambiguity* in analysis by allowing our translator to produce a large number of different outputs from various inputs. In particular, the input/output pairs are indistinguishable from each other, making it difficult to know which output is what is originally intended. Its translation is done by complex machinery, making it difficult to analyze for static analysis (high in the complexity axis). The hidden code is only triggered when a certain condition is satisfied (e.g., when a particular input is provided), imposing challenges to dynamic analysis (high in the evasiveness axis).

III. Ambiguous Translation

We present the design of our approach focusing on how it imposes challenges in terms of ambiguity (Section III-A), complexity (Section III-B), and evasiveness (Section III-C).

A. Ambiguity in Translation



Cur. State	Input	Output	Next State
A	x	M	B
A	y	Error	A

(a) State Machine

(b) State Transitions in a Typical State Machine

Cur. State	Input	Output	Next State
A	v	M	B
A	w	N	B
A	x	O	B
A	y	P	B
A	a	M	C
A	b	N	C
A	c	O	C
A	d	P	C

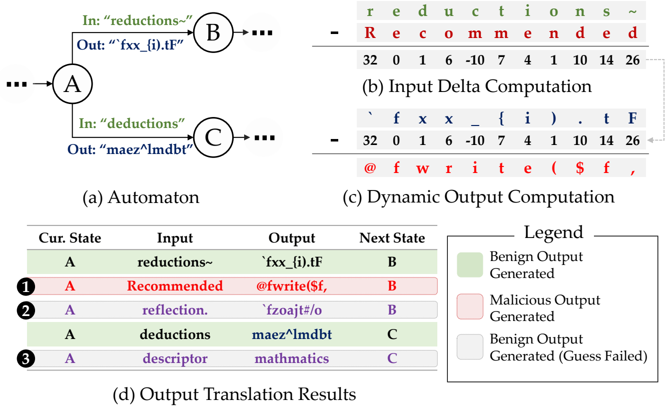
(c) Ambiguous Translation State Machine ($In means the current input)

(d) State Transitions in the Ambiguous Translation

Fig. 2. Typical state machine vs our approach in handling various inputs.

Ambiguity 1: Transition on Every Input. Figure 2-(a) and (b) show a typical state machine that fails if the state machine cannot accept the input (Orange shaded row with the input ‘y’). Such a failure reveals that the state machine only expects a particular input ‘x’, and others are invalid. Figure 2-(c) and (d) show the proposed approach that allows transition on every input. Specifically, if an input does not match with any possible transitions from the current state, it makes a transition to a state which has a similar transition condition to the provided input. For example, in Figure 2-(c), there are two transitions from A to B (if the input is ‘x’) and C (if the input is ‘d’). If the input is neither ‘x’ nor ‘d’, it calculates the distance (by computing value differences in the ASCII code) between the current input and the transition conditions, and selects a transition with the smallest distance. Observe that in Figure 2-(d), all the inputs made transitions without any errors, making it difficult to understand which input is valid or not.

Ambiguity 2: Dynamic Output Translation. Our approach produces different outputs with respect to the provided inputs according to the state transition rules. Specifically, when the input is not exactly matched with the transition's input, it applies the difference between the input and the transition's input to the output translation rule. This allows our approach



(a) Automaton

(b) Input Delta Computation

(c) Dynamic Output Computation

	Cur. State	Input	Output	Next State
	A	reductions~	`fxx_{i).tF	B
❶	A	Recommended	@fwrite($f,	B
❷	A	reflection.	`fzoajt#/o	B
	A	deductions	maez^lmdbt	C
❸	A	descriptor	mathmatics	C

(d) Output Translation Results

Fig. 3. Examples of dynamic output translation. Computations, i.e., (b) and (c), are on ASCII code values.

to generating a certain output without annotating it on the state machine. For example, assume that we want to generate 'M' as output in Figure 2-(c) and (d). Observe that the state machine does not include them in the state transitions. It only has 'O' and 'P'. However, by providing an input 'v' that is smaller (in terms of ASCII code) than the annotated input ('x'), the output is generated as smaller than the annotated output 'M'. The input 'a' also generates 'M' in the same way.

Figure 3-(a) shows a part of the example state machine. Figure 3-(b) and (c) show how the output is generated when the input 'Recommended' is provided. Specifically, for each character, it subtracts the ASCII code value of the characters and takes the value of the result. We then subtract the value to the output annotated with the transition to derive the output.

1. If the input is exactly matched with the input annotated on the automaton, it will produce the same output as annotated (the first and fourth rows).
2. If the given input is different from inputs annotated with the transitions, the state machine finds the transition with the closest delta between the given input and annotated input. For example, the input "`Recommended`" is closer to "`reductions~`" than "`deductions`" according to the delta computation. Hence, the transition to B is chosen, resulting in "`@fwrite($f,`" (❶).
3. If the input is "`reflection.`", it will make the transition from state A to state B and the output will be "`‘fzoajt#/o`" (❷). Similarly, if the input is "`descriptor`", it outputs "`mathmatics`" (❸). Observe that to know all possible outputs, one needs to search a number of inputs on each transition.

Ambiguity 3: Indiscernible Translation Execution. Deniable encryption [35–38] aims to provide similar promises to our approach: generating multiple plausible deciphered texts when incorrect encryption keys are used. Figure 4 illustrates how it works. Essentially, its ciphertext includes decoys that are indistinguishable from the secret and maps incorrect keys to the decoys. However, since having decoys leads to a larger encrypted text, some incorrect keys will be mapped to random values, which unfortunately is distinguishable from the secret.

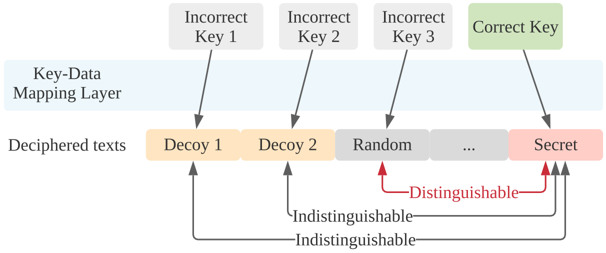


Fig. 4. Example of a deniable encryption implementation.

There are three limitations: (1) Having multiple decoys makes the encrypted text large. Also, the decoys must be carefully chosen, which requires significant manual effort. (2) There are still some keys that are mapped to random deciphered texts which are distinguishable from the secret. (3) To implement the deniable encryption, one needs to implement the key-data mapping layer that maps the keys to the data to decipher. In practice, the number of keys and deciphered texts can be revealed by analyzing the execution of the layer.

Our approach does not have the decoys, which makes every output indistinguishable from the secret. In addition, we do not have the mapping layer, making the execution of any keys indistinguishable as well.

B. Complex State Machine

Translation of inputs and outputs via a state machine is challenging to analyze for static analysis. Specifically, the code for state machine is rather a simple loop and its behavior is largely dependent on the input, which imposes significant challenges to static analysis. To increase the complexity, we add a number of dummy states and state transitions that are indistinguishable from other states and transitions. Note that the dummy states/transitions are not used to process an intended input, hence, they do not impact the operation of the state machine on the intended input. They slightly slow down the performance because those dummy states should be also considered during the state transition.

C. Evasive State Machine

Since the dummy states and transitions are indistinguishable from other states and transitions, analysis techniques need to examine every state transition. Note that this challenge depends on the *indistinguishability* of the states and state transitions. For dummy states and transitions, we generate them by slightly modifying the original states/transitions for the intended input. Specifically, the perturbed dummy states and transitions have small edit distances from the original states and transitions, making them indistinguishable. Moreover, we make the execution of the state transition indistinguishable.

IV. Preliminary Evaluation

We implemented a proof of concept translator written in PHP (2314 LOC for core translation logic).

TABLE I
DETECTION RESULTS ON MALICIOUS AND BENIGN SAMPLES.

Obfuscator	PHP Mal. Finder		Shellray		MalMax	
	Mal.	Benign	Mal.	Benign	Mal.	Benign
PHP Obfuscator [39]	399/413	161/573	479/524	0/573	573/573	0/573
YAK Pro [40]	264/413	139/573	239/524	1/573	573/573	0/573
Best PHP Obfuscator [41]	412/413	573/573	505/524	557/573	573/573	0/573
Simple PHP	413/413	573/573	524/524	573/573	573/573	0/573
Our approach	0/413	0/573	0/524	0/573	0/573	0/573

Green cells on 'Mal.' and 'Benign' columns indicate that they have no true positive and false positive, respectively (lighter green if 5~50%). Red cells represent the opposite (undesirable) results.

A. Against Static and Dynamic Analysis

We compare our approach with state-of-the-art obfuscators.

Obfuscator/malware Detector Selection. We select four obfuscators [39–42] and two malware detectors [43, 44] based on their popularity. In addition, a forced execution based PHP malware scanning tool called MalMax [45] is selected to represent the forced execution based analysis technique.

Sample Selection. We collect 573 server-side malware from known malware collection repositories [46–56]. The samples consist of eight types: webshells, backdoors, bypassers, uploaders, spammers, SQLShells, reverse shells, and flooders. For each type, we collect a similar number of samples (e.g., 61~79). From the 573 malware we collected, for each tool, we use a subset of the samples that the tool can detect them as malicious. Specifically, PHP Malware Finder and Shellray detect 413 and 524 samples, respectively.

Result for Malicious Samples. Table I shows that the tested malware detection tools fail to handle our approach (0%). Note that MalMax is a dynamic analysis technique that can detect malware without the signature. While other three tools fail to detect us as they do not know our approach, results in the MalMax show the effectiveness of our approach.

This is because MalMax focuses on executing all statements without precisely identifying attack triggering inputs. Simply executing all statements of a target is sufficient for analyzing the existing obfuscators but not sufficient for our approach.

False Positives. Some malware detectors often consider *any obfuscated programs as malicious*, causing high false positive rates. To understand false positive, 573 benign PHP program files from popular PHP programs' codebases [57–60] are collected. Initially, none of the 573 benign files are flagged as malware by the existing detectors. After they are obfuscated, we observe many of them are detected as malware (i.e., high false positive rates) by PHP Malware Finder and ShellRay.

B. Against Manual Analysis (via dynamic analysis)

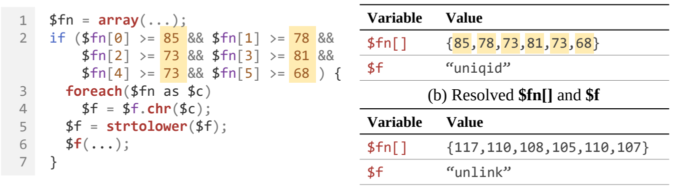
We use Xdebug [61] to monitor our approach's execution to infer the content of the intended input/output. Xdebug is a PHP debugging extension, providing various debugging primitives such as single stepping, variable dumps, and stack traces.

Analyzing Executed Statements. The analyst traces all statements that read and write inputs and values that are computed from inputs (i.e., values that are data dependent on the inputs). Unfortunately, as a state machine is implemented as a loop that makes transitions according to the current input, the resulting statement traces are identical when it delivers intended output and unindented output, meaning that the statement traces are indistinguishable.

Analyzing Values from Executed Statements. One can also dump all the values of the variables used in the executed statements. However, such analysis is only useful when one already knows the intended input. Without knowing that, analyzing values from an execution that does not deliver a malicious payload does not help. Note that forced execution techniques only forcibly run the statements without the correct input values, meaning that their analysis is also ineffective.

C. Against Symbolic Analysis

We apply symbolic analysis to our proof of concept implementation and find out a practical limitation of symbolic analysis techniques. That is, symbolic analysis in practice uses an optimization strategy that aims to find *one* input that drives the execution to the point of interest, instead of enumerating all possible inputs. However, due to the ambiguity in our approach, even one reaches a particular state, the identified input may not be the input of interest. For example, in Figure 5-(a), the array $fn represents a function name. Before it is invoked at line 6, it is constructed at lines 3–5 after satisfying multiple path conditions at line 2. Symbolic analysis encodes the path conditions and gets one solution shown in Figure 5-(b) from the underlying constraint solver. The execution successfully goes into the *true* branch and invokes the function $f. However, it invokes function uniqid instead of the originally intended function unlink shown in Figure 5-(c). Given this branch has been successfully explored, the symbolic analysis will not try other solutions, leaving other existing outputs undiscovered.



```
1  $fn = array(...);
2  if ($fn[0] >= 85 && $fn[1] >= 78 &&
       $fn[2] >= 73 && $fn[3] >= 81 &&
       $fn[4] >= 73 && $fn[5] >= 68 ) {
3    foreach($fn as $c)
4      $f = $f.chr($c);
5    $f = strtolower($f);
6    $f(...);
7  }
```

(a) Source Code

Variable	Value
$fn[]	{85,78,73,81,73,68}
$f	"uniqid"

(b) Resolved **$fn[]** and **$f**

Variable	Value
$fn[]	{117,110,108,105,110,107}
$f	"unlink"

(c) Intended **$fn[]** and **$f**

Fig. 5. Symbolic execution exploring a single input.

V. Future Directions

Understanding Generality of the Technique. We implement our prototype in PHP, the idea is general and can be implemented in other script languages that support dynamic constructs such as eval(). Immediate future work could be implementing our approach in other programming languages.

Analysis Technique against our Approach. To defeat our approach, a specialized analysis technique for our state machine is needed. Specifically, it should compute all possible transitions and potential outputs and identify all the code that can be translated. A static analysis approach can be used to identify all the states and transitions. Then, one can leverage symbolic analysis to identify inputs that can lead to valid outputs. Note that the definition of a valid output varies based on the context. If one knows the grammar of the output text, one can prune out inputs that may not fit the grammar rule. For example, if the output should be a valid function call, and output should follow the 'string()' format. Inputs that cannot generate the output can be pruned out.

Enhancing the Translation Method. The current translation in each transition of the state machine in our approach is essentially a simple substitution cipher. Understanding tradeoffs of using other ciphers such as block/stream ciphers [62–65] is a promising future direction to further enhance the proposed approach. Moreover, the proposed technique requires a large state machine to process a non-trivial size of input. Reducing the size of the state machine is also critical. A possible option can be reusing states and transitions in the state machine.

Runtime Environment against the Proposed Approach. It is possible that a cybercriminal may leverage the proposed technique to deliver a malicious payload secretly. A practical solution to prevent the proposed approach is developing a safe runtime environment that can restrict malicious behaviors executed by the proposed technique (e.g., sandboxing). For example, since the proposed approach needs to use dynamic code generation techniques, one can thwart the attack by disabling primitives for dynamic code generation.

Acknowledgment

We thank the anonymous referees for their constructive feedback. The research was supported, in part, by NSF under awards 1916499, 1908021, 1850392, 1909856, 1916500, 2047980, and a Mozilla Research Award (2019). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] J. Dahse and J. Schwenk, "Rips-a static source code analyser for vulnerabilities in php scripts," *Retrieved: February*, vol. 28, p. 2012, 2010.

[2] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2006, pp. 6–pp.

[3] D. Ryan, "Implementing basic static code analysis into integrated development environments (ides) to reduce software vulnerablitilies," *A Report submitted in partial fulfillment of the regulations governing the award of the Degree of BSc (Honours) Ethical Hacking for Computer Security at the University of Northumbria at Newcastle*, vol. 2012, 2011.

[4] D. Hauzar and J. Kofroň, "Weverca: Web applications verification for php," in *International Conference on Software Engineering and Formal Methods*. Springer, 2014, pp. 296–301.

[5] E. Kneuss, P. Suter, and V. Kuncak, "Phantm: Php analyzer for type mismatch," in *FSE'10 Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, no. CONF, 2010.

[6] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*. Springer, 2008, pp. 65–88.

[7] G. Suarez-Tangil, J. E. Tapiador, F. Lombardi, and R. Di Pietro, "Thwarting obfuscated malware via differential fault analysis," *Computer*, vol. 47, no. 6, pp. 24–31, 2014.

[8] J. Mao, J. Bian, G. Bai, R. Wang, Y. Chen, Y. Xiao, and Z. Liang, "Detecting malicious behaviors in javascript applications," *IEEE Access*, vol. 6, pp. 12 284–12 294, 2018.

[9] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[10] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A symbolic execution framework for javascript," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 513–528.

[11] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE symposium on Security and privacy*. IEEE, 2010, pp. 317–331.

[12] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 488–498.

[13] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 842–853.

[14] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-force: Forced execution on javascript," in *Proceedings of the 26th international conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017, pp. 897–906.

[15] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: force-executing binary programs for security applications," in *23rd USENIX Security Symposium*, 2014, pp. 829–844.

[16] Z. Deng, B. Saltaformaggio, X. Zhang, and D. Xu, "iris: Vetting private api abuse in ios applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 44–56.

[17] J. Wilhelm and T.-c. Chiueh, "A forced sampled execution approach to kernel rootkit identification," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 219–235.

[18] R. Johnson and A. Stavrou, "Forced-path execution for android applications on x86 platforms," in *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*. IEEE, 2013, pp. 188–197.

[19] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 2007, pp. 421–430.

[20] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1998, pp. 184–196.

[21] B. Sheridan and M. Sherr, "On manufacturing resilient opaque constructs against static analysis," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 39–58.

[22] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals." in *USENIX Security Symposium*, 2007, pp. 275–290.

[23] B. Lee, Y. Kim, and J. Kim, "binob+: a framework for potent and stealthy binary obfuscation," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 271–281.

[24] M. Ollivier, S. Bardin, R. Bonichon, and J.-Y. Marion, "How to kill symbolic deobfuscation for free (or: Unleashing the potential of path-oriented protections)," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 177–189. [Online]. Available: https://doi.org/10.1145/3359789.3359812

[25] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 189–200. [Online]. Available: https://doi.org/10.1145/2991079.2991114

[26] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *Proceedings of the 16th European Conference on Research in Computer Security*, ser. ESORICS'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 210–226.

[27] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation." in *NDSS*, 2008.

[28] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *European Symposium on Research in Computer Security*. Springer, 2011, pp. 210–226.

[29] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P.-c. Yew, "Control flow obfuscation with information flow tracking," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009, pp. 391–400.

[30] S. Schrittwieser, S. Katzenbeisser, P. Kieseberg, M. Huber, M. Leithner, M. Mulazzani, and E. Weippl, "Covert computation: Hiding code in code for obfuscation purposes," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 529–534.

[31] A. Fass, M. Backes, and B. Stock, "Hidenoseek: Camouflaging malicious javascript in benign asts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1899–1913.

[32] J. Ming, D. Xu, L. Wang, and D. Wu, "Loop: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 757–768.

[33] Q. Yang, "GitHub - quanyang/Taint-em-All: A taint analysis tool for the PHP language," 2019, https://github.com/quanyang/Taint-em-All.

[34] O. Olivo, "GitHub - olivo/TaintPHP: Static Taint Analysis for PHP web applications," 2016, https://github.com/olivo/TaintPHP.

[35] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky, "Deniable encryption," in *Annual International Cryptology Conference*. Springer, 1997, pp. 90–104.

[36] A. Sahai and B. Waters, "How to use indistinguishability obfuscation: deniable encryption, and more," in *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, 2014, pp. 475–484.

[37] M. Dürmuth and D. M. Freeman, "Deniable encryption with negligible detection probability: An interactive construction," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2011, pp. 610–626.

[38] M. Klonowski, P. Kubiak, and M. Kutyłowski, "Practical deniable encryption," in *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer, 2008, pp. 599–609.

[39] M. Fonk, "GitHub - naneau/php-obfuscator: an "obfuscator" for PSR/OOp PHP code," 2019, https://github.com/naneau/php-obfuscator.

[40] P. Kissian, "YAK Pro: Php Obfuscator," 2019, https://www.php-obfuscator.com/.

[41] "Best PHP Obfuscator," 2018. [Online]. Available: http://www.pipsomania.com/best_php_obfuscator.do

[42] R. Lie, "Simple online PHP obfuscator: encodes PHP code into random letters, numbers and/or characters," 2019, https://www.mobilefish.com/services/php_obfuscator/php_obfuscator.php.

[43] N. Systems, "GitHub - nbs-system/php-malware-finder: Detect potentially malicious PHP files," 2019, https://github.com/nbs-system/php-malware-finder/.

[44] "Shellray: A PHP webshell detector," 2019, https://shellray.com/.

[45] A. Naderi-Afooshteh, Y. Kwon, A. Nguyen-Tuong, A. Razmjoo-Qalaei, M.-R. Zamiri-Gourabi, and J. W. Davidson, "Malmax: Multi-aspect execution for automated dynamic web server malware analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1849–1866.

[46] "GitHub - tdifg/WebShell: WebShell Collect," 2016, https://github.com/tdifg/WebShell.

[47] "GitHub - BDLeet/public-shell: Some Public Shell," 2016, https://github.com/BDLeet/public-shell.

[48] "GitHub - nixawk/fuzzdb: Web Fuzzing Discovery and Attack Pattern Database," 2018, https://github.com/nixawk/fuzzdb.

[49] "GitHub - BlackArch/webshells: Various webshells," 2019, https://github.com/BlackArch/webshells.

[50] "GitHub - tanjiti/webshellSample: Webshell sample for WebShell Log Analysis," 2018, https://github.com/tanjiti/webshellSample.

[51] "GitHub - xl7dev/WebShell: Webshell & Backdoor Collection," 2017, https://github.com/xl7dev/WebShell.

[52] J. Troon, "GitHub - JohnTroony/php-webshells: Common php webshells," 2016, https://github.com/JohnTroony/php-webshells.

[53] "GitHub - bartblaze/PHP-backdoors: A collection of PHP backdoors," 2019, https://github.com/bartblaze/PHP-backdoors.

[54] "GitHub - tennc/webshell: A webshell open source project," 2019, https://github.com/tennc/webshell.

[55] "GitHub - Ridter/Pentest," 2019, https://github.com/Ridter/Pentest.

[56] "VirusShare," 2019, https://virusshare.com/.

[57] W. Foundation, "WordPress," 2019, https://wordpress.com/.

[58] "Joomla: Content Management System (CMS)," 2019, https://www.joomla.org/.

[59] M. M. Fauth, "GitHub - phpmyadmin/phpmyadmin: A web interface for MySQL and MariaDB," 2019, https://github.com/phpmyadmin/phpmyadmin.

[60] L. Masters, "CakePHP: The Rapid Development Framework for PHP," 2019, https://cakephp.org/.

[61] Derick Rethans, "Xdebug - Debugger and Profiler Tool for PHP," 2020, https://xdebug.org/.

[62] X. Lai, "On the design and security of block ciphers," Ph.D. dissertation, ETH Zurich, 1992.

[63] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "The simon and speck lightweight block ciphers," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[64] R. A. Rueppel, *Analysis and design of stream ciphers*. Springer Science & Business Media, 2012.

[65] T. W. Cusick, C. Ding, and A. R. Renvall, *Stream ciphers and number theory*. Elsevier, 2004.