Hiding Critical Program Components via Ambiguous Translation

Chijung Jung
University of Virginia
Charlottesville, VA
cj5kd@virginia.edu

Doowon Kim
University of Tennessee,
Knoxville, Knoxville, TN
doowon@utk.edu

An Chen
University of Georgia
Athens, GA
ac24057@uga.edu

Weihang Wang
University at Buffalo, SUNY
Buffalo, NY
weihangw@buffalo.edu

Yunhui Zheng
IBM Research
Yorktown Heights, NY
zhengyu@us.ibm.com

Kyu Hyung Lee
University of Georgia
Athens, GA
kyuhlee@uga.edu

Yonghwi Kwon
University of Virginia
Charlottesville, VA
yongkwon@virginia.edu

ABSTRACT

Software systems may contain critical program components such as patented program logic or sensitive data. When those components are reverse-engineered by adversaries, it can cause significantly damage (e.g., financial loss or operational failures). While protecting critical program components (e.g., code or data) in software systems is of utmost importance, existing approaches, unfortunately, have two major weaknesses: (1) they can be reverse-engineered via various program analysis techniques and (2) when an adversary obtains a legitimate-looking critical program component, he or she can be sure that it is genuine.

In this paper, we propose Ambitr, a novel technique that hides critical program components. The core of Ambitr is *Ambiguous Translator* that can generate the critical program components when the input is a correct secret key. The translator is ambiguous as it can accept any inputs and produces a number of legitimate-looking outputs, making it difficult to know whether an input is correct secret key or not. The executions of the translator when it processes the correct secret key and other inputs are also indistinguishable, making the analysis inconclusive. Our evaluation results show that static, dynamic and symbolic analysis techniques fail to identify the hidden information in Ambitr. We also demonstrate that manual analysis of Ambitr is extremely challenging.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Software reverse engineering**.

KEYWORDS

program translation, software protection, reverse engineering

1 INTRODUCTION

Software systems often contain critical program components such as classified, sensitive, or proprietary code or data, which we call *Critical Program Components* (or *CPC*). For example, patented program logic is an example of CPC. If an adversary steals or copies a competitor's software system's patented technology, it would cause significant financial loss. Similarly, in a warfare software system (e.g., software in a drone), a CPC can be a piece of code containing its operational procedures, including the targets and plans. Since an adversary can reverse-engineer a software system to reveal various critical operational secrets (e.g., targets of the military system and target operation date) which can be used against the victim, protecting CPCs is an essential requirement.

There are a few techniques that can be leveraged to hide critical program components: obfuscation [5, 12, 37, 61], packing [14, 38, 40], and encryption [66, 75]. Code obfuscation techniques *syntactically* transform the original program's code into another form of code, making it difficult to be analyzed manually. Data obfuscation techniques [3, 18, 30] change the value of data in a way that does not change the original semantic of the data while making it difficult to know the original value. However, both obfuscation techniques preserve critical semantics, meaning that they only delay the analysis but cannot protect the critical components. A packer compresses or encrypts the program code and data, and stores them in a data section of the packer's loader program. However, it is not suitable for hiding CPCs because it always decompresses (or decrypts) the original program code and data at runtime.

To understand the effectiveness of the existing techniques in hiding critical program components, we analyze approaches that can be used against the obfuscation, packing, and encryption techniques. Specifically, we observe that obfuscation techniques and packers can be easily traced and analyzed by dynamic analysis [9, 41, 68]. While encryption-based techniques are challenging to break cryptographically, the execution of the decryption function can be traced to extract the decrypted data (i.e., the genuine critical program components). To this end, we conclude that while the techniques certainly raise the bar in analysis (i.e., making the analysis challenging), it is practically feasible for a persistent and determined adversary to obtain the critical program component protected by existing techniques. More importantly, since there is no ambiguity in decoding and uncompression processes, the adversary knows that the CPC is undoubtedly correct when obtained.

In this paper, we propose a novel technique, Ambitr, that aims to hide critical program components against adversaries with access

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510139

to the target program. Specifically, we hide critical program components (e.g., program code or data) by encoding the components to a complex state-machine. Given a correct secret key, the state-machine generates the genuine critical program components (e.g., program code or data). The key difference between Ambitr and existing techniques is that Ambitr can take the incorrect secret key as input and generate legitimate-looking CPCs, making it difficult to determine whether the given input is the correct secret key or not. Unlike a typical state-machine, Ambitr's state-machine allows a transition on any inputs even if it does not match the transition's input (i.e., a typical state-machine will raise an error if it does not match). The differences between the state transition's input and the given input are then used to generate output different from the state transition's output. This significantly enlarges the input/output space of a state transition in Ambitr. With the state-machine, Ambitr introduces a unique challenge to the adversary, *Ambiguity*, meaning that even if the adversary identifies a legitimate-looking output from Ambitr, the adversary does not know whether the output is the genuine critical program component. To this end, with the sophisticated construction of our state-machine, the critical program component hidden by Ambitr is extremely challenging to be identified. Moreover, even when some possible outputs are identified, one cannot know which output is the genuine CPC.

Our contributions are summarized as follows:

- We analyze limitations of existing techniques aiming to hide program code, and investigate a possibility of adding a new challenge: ambiguity.
- We propose Ambitr, which can hide critical program components (CPCs) through a sophisticated translation technique that accepts any inputs and generates multiple plausible CPCs that are not distinguishable from the genuine CPC.
- We perform a thorough evaluation using state-of-the-art dynamic, static, and symbolic analysis tools to demonstrate Ambitr's resilience to reverse-engineering attempts.

2 POSITIONING AND BACKGROUND

2.1 Definition

Critical Program Component (CPC). We define Critical Program Component as a piece of code or data that contains critical program logic or information, which is not desirable to be known to the adversary. It is important to mention that, in our context, while the adversary knows that there is a CPC hidden in the program, he or she does not know what the CPC should be. In other words, given a set of plausible CPC examples, the adversary does not know which one is the correct CPC. In this paper, we aim to prevent the adversary from identifying and pinpointing the correct CPC.

2.2 Positioning

Typical Usage Scenario. Figure 1 illustrates how Ambitr operates under a typical usage scenario of our research. Specifically, in a target program, we use our *Ambiguous Translator* to hide a critical program component. At runtime, it receives an input from an external source such as network (❶), and feeds it to the ambiguous translator (❷) which generates outputs according to the input. If the input is the correct secret key, the genuine CPC is generated (❸). On other inputs, our ambiguous translator still produces valid outputs without failing. In particular, on certain specialized inputs, decoy CPCs that are indistinguishable from the genuine CPC are generated (❹). Finally, the outputs (i.e., CPCs) are processed or executed, if its type is an executable code (❺).

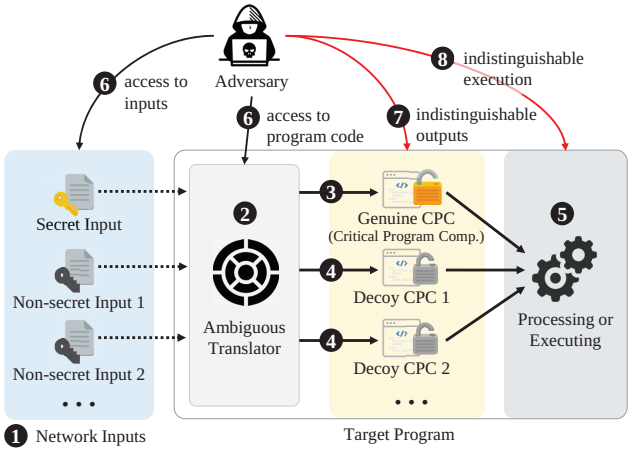


Figure 1: Assumed Scenario and Scope of the Research.

Adversary Model and Scope. In this work, we assume the adversary has access to previous inputs and the target program's code, including our ambiguous translator's logic (❻). The adversary can also run the program with any inputs including the obtained previous inputs. The goal of Ambitr is to prevent the adversary from identifying the secret input that can generate the genuine CPC without a doubt (❸). To achieve the goal, Ambitr can generate outputs including the genuine CPC and decoy CPCs that are indistinguishable from each other (❼). The execution of the ambiguous translator when it generates the original or decoy CPCs is also indistinguishable, as well as the processing or execution of the generated CPCs (❽).

We assume that the previous inputs might exist in a network log and are available to the adversary. However, the adversary does not know what is the secret input, from the obtained previous input. Some of the inputs may generate decoy CPCs. We assume that target program's behaviors and execution for processing and executing the CPCs are not distinguishable. Otherwise, the code can be traced to identify which input generates the genuine CPC. If the original target program should execute different program code, such code should be included in the executable CPC. We assume the adversary can leverage various static and dynamic analysis techniques to analyze our ambiguous translator. We consider our approach is successful if the adversary *fails to pinpoint the genuine CPC, even if many (or even all) valid CPCs are identified.*

2.3 Existing Techniques for Hiding CPC

A few techniques can be leveraged to hide a CPC in a program. Specifically, the columns in Table 1 present the techniques while each row of the table shows program analysis approaches that can be used to identify CPCs. Symbols represent the effectiveness of the program analysis approaches against each technique.

Table 1: Effectiveness of Existing Techniques and Ambitr against Program Analysis Approaches.

	Obfuscators	Packers/Crypters	Protectors	Ambitr
Static Analysis	◑[1]	◑[2]	◑[2]	●
Symbolic Analysis	◑[1]	●	●	●
Dynamic Analysis	○	○	●	●
Forced Execution	○	○	○	●

○ : Ineffective, ◑ : Less effective, ● : Effective (against analyses).
1: Static/symbolic analysis techniques have difficulty handling advanced obfuscators (with multiple layers of obfuscations) due to state explosion, while they can handle simple obfuscators.
2: Static analysis may handle known crypto algorithms while it may not generically handle them (hence half-filled circled, meaning that effective on some but not all).

2.3.1 Obfuscators. Obfuscation techniques [3, 5, 12, 16, 18, 24, 30, 37, 47, 54, 61, 64, 67, 76] aim to make the original code difficult to analyze by leveraging techniques including opaque predicates [16, 47, 67], code insertion/replacement [5, 24, 37, 54, 61, 76], and hardware primitives [12, 64].

Limitations. Obfuscation techniques that transform code into semantically equivalent forms or add non-essential code (e.g., opaque predicates and dummy code) [5, 16, 24, 37, 47, 54, 61, 67, 76] can be handled by automatically reverting or removing the modified/added code via program analysis techniques [33, 44, 49, 53, 80, 81]. Depending on the obfuscation techniques used, static and symbolic analysis may suffer from the complexity of the analysis, meaning that they might not be always effective, as described in Table 1. Typically, dynamic analysis (including forced execution [57]) techniques are highly effective in handling the obfuscation techniques. While data obfuscation techniques [3, 18, 30] change the values of data, their critical semantics are preserved and can be traced and identified by both static and dynamic analysis [46, 79].

2.3.2 Packers/Crypters. Packers [14, 38, 40] primarily aim to hinder static analysis. Specifically, they create a program containing compressed original program as data, that uncompresses and executes the original program at runtime. Crypters [2, 6, 29] are essentially advanced packers using crypto techniques to hide the program data and code. Due to the complexity of compression and encryption, static and symbolic analysis are not effective as shown in Table 1. In particular, symbolic analysis suffers from state explosion due to the complex computations of encryption schemes.

Limitations. Since a packer generated program seamlessly unpacks and executes the original code at runtime, dynamic analysis (i.e., executing the binary and extracting the uncompressed program) [13, 34, 62] can obtain the original program.

2.3.3 Protectors. Protectors [59, 69, 82, 83] are essentially advanced packers/crypters equipped with evasive anti-analysis techniques such as terminating the execution if they detect reverse-engineering attempts (e.g., running the program with a debugger). Similar to packers/crypters, since the program itself is compressed and encrypted, static and symbolic analyses are not effective, as described in Table 1. Specifically, symbolic and concolic analyses can be used to avoid the evasive techniques by extracting and solving the evasive predicate conditions. However, they are difficult to scale to the programs generated by protectors. Moreover, dynamic analysis is ineffective because of the evasive techniques.

Limitations. Forced execution techniques [19, 32, 35, 57, 78] aim to handle evasive techniques by forcibly executing branches regardless of the predicate conditions. Most protectors can be handled by the forced execution techniques. Note that since the forced execution techniques forcibly execute program code regardless of the predicate conditions, they may fail to handle an advanced protector which uses predicate conditions for both evasive techniques and decryption (i.e., decryption logic is dependent on the predicate conditions). However, by observing the predicate conditions and executions of the program, it is straightforward to tune the analysis technique to handle such advanced protectors (e.g., one can selectively solve such a critical predicate with symbolic execution to handle the limitation) [70].

2.4 Desirable Properties

We present four desirable properties of a CPC hiding techniques: *Evasiveness*, *Complexity*, *Context-Sensitivity*, and *Ambiguity*.

From Existing Literature. For the first three properties, we identify and summarize them from existing literature. Note that prior literature does not explicitly present the properties. They are only implicitly mentioned individually (e.g., evasiveness in [80], complexity in [5], context-sensitivity in [40, 44]). We systematically studied prior literature to establish the desirable properties. In particular, from program analysis papers [33, 44, 80], we mainly focus on the challenges, e.g., state-explosion caused by complexity, they pointed out. From anti-program analysis techniques [5, 24, 40, 54, 64], we pay attention to the approaches proposed by them to hinder the analysis (e.g., evasive tactics [54]). We believe the four properties thoroughly cover the core properties across the literature.

New Desired Property: Ambiguity. We introduce a new desirable characteristic: *Ambiguity* (details in Section 2.4.4).

2.4.1 Evasiveness. Programs that are highly evasive (e.g., programs with a number of evasive predicates) impose significant challenges to symbolic and dynamic analysis. For dynamic analysis, knowing a number of concrete inputs that can cover all the evasive predicates is challenging. For symbolic analysis, an excessive number of predicates and complex predicate conditions cause the scalability problem (i.e., taking too much time making the technique practically unusable).

2.4.2 Complexity. Static and symbolic analyses have difficulty analyzing programs with complex operations. Typical examples are packed/encrypted programs. Static and symbolic analyses can reverse-engineer the uncompression/decryption process. However, they fail to scale complex algorithms (e.g., a crypto algorithm).

2.4.3 Context-sensitivity. Some programs have context-sensitive code, meaning that their behaviors are dependent on a particular program execution path. Since there are a large number of program paths, it is common for static analysis to conduct context-insensitive analysis. Symbolic analysis aims to discover various execution contexts; hence often suffers from the excessive number of program execution paths, causing the path explosion. Forced execution solves the path explosion problem by *forcibly executing code guarded by branches*. However, due to the ignored branch outcomes which lead to incorrect context, the results of the execution may not be precise.

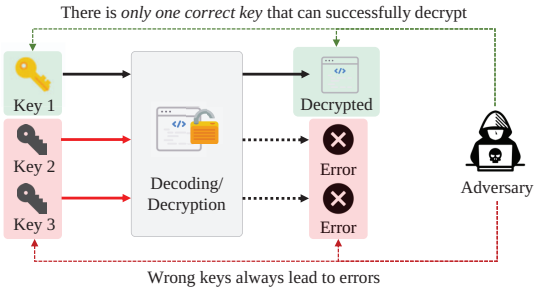


Figure 2: Existing Techniques are *NOT* Ambiguous.

2.4.4 Ambiguity. When an adversary obtains a successfully decoded/decrypted CPC, if the adversary can certainly say the CPC is genuine, we consider the technique is *not* ambiguous. In other words, if the adversary cannot determine whether the CPC is correct or not, we consider the technique has the *ambiguity* property. Specifically, obfuscators do not go through a decoding process, meaning that executing the obfuscated program would expose the critical program components. Packers/crypters/protectors typically store the compression/encryption key for the critical program component in the programs. Hence, running the program, without any particular input, would expose the CPC. Advanced crypters/protectors often store the key for CPC in a separate place, making it challenging to decrypt. Similar to Ambitr, an application may receive the key via the Internet.

Assume that an adversary obtains a few keys from the network traffic logs, and try them to the program. Figure 2 describes an example scenario with three different keys, where Key 1 is correct and Key 2 and 3 are incorrect. Unlike Ambitr, existing crypters/protectors are not ambiguous, meaning that the decoding/decryption will be only successful with Key 1 and all other keys (e.g., Key 2 and 3) will result in errors. As a result, observing any successful decryption with a key implies that the decrypted CPC are genuine.

Table 2: Properties in Existing Techniques and Ambitr.

	Obfuscators	Packers/Crypters	Protectors	Ambitr
Evasiveness	○	○	◑[1]	●
Complexity	◑[2]	◑[2]	◑[2]	●
Context-Sensitivity	○	○	○[3]	●
Ambiguity	○	○	○	●

● : High, ◑ : Medium, ○ : Low, ○ : No.

1: Protectors have medium evasiveness because while they detect the environment to avoid (e.g., VM/debugger), their detection is not sophisticated.
2: Obfuscators/Packers/Crypters/Protectors use various encoding/crypto algorithms with varying complexity, determining the complexity property. Both simple and complex algorithms are used, leading to the medium.
3: Very few protectors are context-sensitive: e.g., using a (context-sensitive) variable as a decryption key.

Summary of Desirable Properties. Table 2 shows the desirable properties in existing techniques and Ambitr. As discussed, none of existing techniques has the ambiguity property. Moreover, Ambitr is more evasive, complex, and context-sensitive than existing techniques.

3 DESIGN

3.1 Overview and Intuition

Ambitr leverages a specialized state machine to translate input to CPCs. The state machine is designed to accept any input values and generate the genuine CPC or decoy CPCs depending on the input.

The state machine achieves *Evasiveness* and *Context Sensitivity* since without knowing the particular secret key (i.e., the secret input) for the genuine CPC, executing the state machine with other inputs does not produce the genuine CPC. The state machine contains a number of states for decoy CPCs, achieving *Complexity*. Finally, the decoy CPCs and the execution of Ambitr are not distinguishable to the genuine CPC, achieving *Ambiguity*.

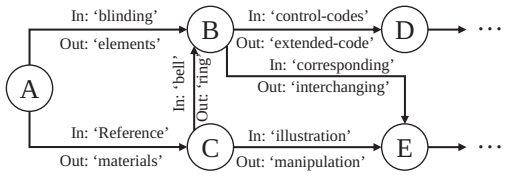
3.1.1 Ambitr versus a Typical State Machine. A typical state machine only accepts input that can make state transitions from the current state. Hence, to understand all possible inputs (and corresponding outputs), one can collect all the state transitions' inputs and come up with the permutations of them. Unlike traditional state machine that should have an accepting state, Ambitr does not have the acceptation state. It terminates when it has consumed all the inputs. Note that Ambitr's output is generated when a transition happens, not at the accepting state as a traditional state-machine does.

Figure 3-(a) shows an example state-machine. Circles and arrows represent states and state transitions including input and output of each transition ('In' and 'Out'). A traditional state machine can only accept inputs that match the state transitions' inputs. For instance, from Ⓐ, it only accepts two inputs "blinding" and "Reference" that make transitions to Ⓑ and Ⓒ, respectively. The restriction on accepted inputs essentially limits the input and output space. Figure 3-(b) shows all possible inputs and outputs of the traditional state machine from Ⓐ to Ⓓ and Ⓔ. This can be done by identifying all possible state transitions and inputs because any other inputs (e.g., the last row of Figure 3-(b)) result in errors.

Inputs for CPCs is Implicit in Ambitr. Figure 3-(c) shows inputs and outputs that can be handled by Ambitr using the state machine in Figure 3-(a). Note that it can handle all the inputs in the same way the traditional state machine handles. The first row shows an example.

Ambitr allows a CPC to be decoded by inputs that do not match the state transitions' inputs. The second row shows an example. The first input "pywudh"" does not match any transition inputs from Ⓐ: "blinding" for Ⓑ and "Reference" for Ⓒ. However, as shown in the third column, it makes a transition to Ⓑ, since the distance (in ASCII code value of each byte) between the given input and the state transition's input of Ⓑ is closer than the state transition's input of Ⓒ. When it produces an output, it also uses the measured distance between the input and the state transition's input to compute a new output value that is different from the state transition's output. By doing so, Ambitr's state machine does not have restrictions on the inputs it can take, meaning that any inputs can be accepted. Moreover, outputs that Ambitr's state machine can produce are not restricted as well.

The second, third, and fourth rows in Figure 3-(c) show examples of legitimate-looking decoy CPCs (i.e., meaningful executable code but not the genuine CPC) from inputs that do not match any state



(a) State Machine ('...' represents omitted states)

Input	Output	State Transitions
blinding control-codes	elements extended-code	A → B → D
blinding corresponding	elements interchanging	A → B → E
Reference illustration	materials manipulation	A → C → E
Reference bell control-codes	materials ring extended-code	A → C → B → D
Reference bell corresponding	materials ring interchanging	A → C → B → E
Reference Code	materials <Error>	A → C → <Error>

(b) Traditional State Machine's State Transitions

Input	Output	State Transitions
blinding control-codes	elements extended-code	A → B → D
pywudh" "VATXf.AR3R*I	system($_GET['xx']);	A → B → D
rnpjmf" "]c{i4B	unlink($file);	A → B → D
Re[l $1kb, !*0prm,$h);	mail ($to, $t,$msg,$h);	A → C → E

(c) AMBITR's State Transitions

Figure 3: Traditional State Machine vs. AMBITR.

transitions' inputs. The three inputs have different sizes, and the fourth row's input leads to different state transitions (Ⓐ to Ⓒ and Ⓔ) from the other two. Note that many more inputs can generate legitimate-looking outputs, and one can brute-force inputs (e.g., trying all possible strings for input) to enumerate them. We explain the details of the state machine in Section 3.2.1.

3.1.2 Ambiguity in AMBITR. AMBITR introduces ambiguity in two aspects: ambiguity in input/output and execution.

Ambiguous Input/Output. The input of AMBITR is ambiguous because it can take any inputs even if it does not match any state transition inputs, as shown in Figure 3-(c). When the input does not match any state transitions, AMBITR finds a transition that has the closest input to the provided input (in terms of ASCII code value of each byte of input). Observe that AMBITR's output can also differ from the state transition's output and is dependent on input, meaning that the output is also ambiguous.

The ambiguity of the outputs makes the analysis inconclusive. For example, in Figure 3-(c), the second, third, and fourth rows' outputs are all legitimate executable code. Hence, it is challenging to conclude which one is the genuine CPC.

Ambiguous Execution. One may use dynamic analysis to trace the execution of AMBITR to understand whether there are any execution differences while processing different inputs. If such a difference exists, it can be used to infer the genuine CPC. As shown in Algorithm 1 that describes the algorithm of AMBITR's state machine (will be explained in Section 3.2.1), there are no predicates and computations that behave distinctively. Hence, tracing the execution of AMBITR does not help to identify the genuine CPC.

Algorithm 1: Algorithm of Ambiguous Translator

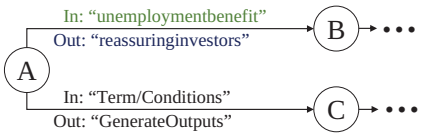
Input : InStr: Array of Tokenized Input String.
Output: OutStr: Output String.

```
1  procedure StateMachine(InStr)
       // Assign the Initial State (i.e., INIT).
2      State_cur ← INIT
3      while until it consumes all the tokens of InStr; the current token
         is InStr_cur do
           // Find the matching (or closest) transition from the current state.
4          State_next, Tran_In_Δ, Tran_Out ← FindTransition
             (State_cur, InStr_cur)
           // Change the current state
5          State_cur ← State_next
6          Out_cur ← ∅
           // Compute Output according to the distance between the input and
             transition's input
7          for each byte t_i and t_o in Tran_In_Δ and Tran_Out do
               // '·' is a string concatenation operator.
8              Out_cur ← Out_cur · Round(t_o − t_i)
9          OutStr ← OutStr · Out_cur
10     return OutStr
11 procedure FindTransition(State_cur, InStr_cur)
12     MinScore ← −1
13     for each transition tr from State_cur do
14         Score ← 0
15         tr_Δ ← ∅
16         for each byte b_t of input of transition tr, and each byte b_i
             from InStr_cur do
17             Score ← Score + | b_t − b_i |
               // char() converts a number to a string, '·' concatenates strings.
18             tr_Δ ← tr_Δ · char(b_t − b_i)
           // Finding the matching (or closest) transition.
19         if MinScore is −1 or min > score then
20             MinScore ← Score
21             Tran_In_Δ ← tr_Δ
               // tr_next represents the next state of the transition tr
22             State_next ← tr_next
               // tr_out represents the output of the transition tr
23             Tran_Out ← tr_out
24     return State_next, Tran_In_Δ, Tran_Out
```

3.2 Composing AMBITR

AMBITR consists of two components: (1) *Ambiguous Translator*, which is a piece of software that processes input according to the state machine definition to generate a CPC (Section 3.2.1) and (2) definition of the state machine that the Ambiguous Translator operates (Section 3.2.2).

3.2.1 Ambiguous Translator Runtime. The core of AMBITR is the runtime of Ambiguous Translator. It has two unique characteristics. First, regardless of the current state and input, it *always transits to another state* even the input does not match any transitions (**C1**). Note that, in a typical state machine, a state transition only happens when there is a transition that can accept the current input. Second, when AMBITR takes inputs that do not match the existing transitions, the output generated by AMBITR is also different from the transitions' outputs (**C2**). Specifically, the final output is computed



(a) Automaton

Input	Output	State Trans.
'l18mibthay.G1'#T <	**$c->predict([1,2]);**	**A → B**
sjwmq_o{""hsP)6	passthru($cmd);	A → B
x{p^bZltd&)y`r$,	urldecode($str);	A → B
swvq1G#-{ $\hn)	fwrite($f,$cnt)	A → C
phvw,V#-y]r"5	chroot($dir);	A → C
ozhm QhyyYsl"^)	bzdecompress(d)	A → C

(b) Output Translation Results

	u	n	e	m	p	l	o	y	m	e	n	t	b	e	n	e	f	i	t
-	'	l	1	8	m	i	b	t	h	a	y	.	G	1	'	#	T		<
	78	2	52	53	3	3	13	5	5	4	-11	70	27	52	71	66	18	73	56

(c) Input Delta Computation

	r	e	a	s	s	u	r	i	n	g	i	n	v	e	s	t	o	r	s
-	78	2	52	53	3	3	13	5	5	4	-11	70	27	52	71	66	18	73	56
	$	c	-	>	p	r	e	d	i	c	t	(	[	1	,	2	]	)	;

(d) Output Computation

Figure 4: Examples of Dynamic Output Translation. Computations, i.e., (b) and (c), are on ASCII Code Values.

based on a concrete input at runtime. This significantly enlarges the search space of input and output. Algorithm 1 shows its algorithm, and we will use it to explain the details of the two characteristics.

Making Transitions on Any Inputs (C1). *Ambiguous Translator* makes transitions from any states on any inputs. Specifically, for all next reachable states from the current state, it calculates the distance (by subtracting values from each byte offset and accumulating the results as shown in Figure 4) between the current input and the transitions' inputs (`FindTransition` in Algorithm 1). Lines 16-18 in Algorithm 1 essentially compute the distance (`Score`). Then, it selects a transition with the smallest distance (if there are multiple ties, we pick the first one to make it deterministic) as shown in lines 19-23 in Algorithm 1.

Dynamic Output Translation (C2). When *Ambiguous Translator* makes a transition on an input that is not exactly matched with the transition's input, it generates output that is different from the current state transition's output. Specifically, it computes the new output by applying the differences between the current input and the current state transition's input. This makes the output space significantly large as the output can vary as much as the input varies.

In Algorithm 1, one of the return values of `FindTransition` (line 4) is $Tran\_In_{\Delta}$, which represents the distance between the current input and the current state transition's input. `FindTransition` also returns the current (i.e., selected) transition's output as *Tran_Out*. Then, at lines 7-8, it computes the new output by subtracting each byte of $Tran\_In_{\Delta}$ (i.e., t_i) from the transition's output *Tran_Out* (i.e., t_o). Note that there is the `Round` function at line 8, which essentially rounds the computed value to be in the visible ASCII code value range (i.e., 32~126).

Example. Figure 4-(a) shows an automaton of a state machine where inputs and outputs of transitions are illustrated above and below the arrows. Figure 4-(c) describes an example computation of distances (i.e., delta) between the transition's input (e.g., "`unemploymentbenefit`") and the given input at runtime (e.g., "`'l18mibthay.G1'#T <`"). Specifically, for each character, it subtracts ASCII code values of the characters. The results are shown at the bottom line of Figure 4-(c). We then subtract the values to the transition's output to derive the final output (i.e., "`$c->predict([1,2]);`") as shown in Figure 4-(d).

Figure 4-(b) presents six examples of input and output pairs from Ⓐ (three for Ⓐ ↦ Ⓑ and the other three for Ⓐ ↦ Ⓒ). The first example is the one that is illustrated in Figure 4-(c) and (d). The second and third examples show inputs for generating function calls `passthru` and `urldecode`. The three examples show that the same state transition, Ⓐ ↦ Ⓑ, (with different inputs) can generate completely different outputs (i.e., CPCs), making the translation ambiguous.

The next three examples are generated via the transition Ⓐ ↦ Ⓒ. Again, depending on the given input, it generates completely different outputs, and those outputs are all legitimate executable code, making it difficult to know which one is the genuine CPC.

3.2.2 Composing Automaton. Ambitr's *Ambiguous Translator* operates on an automaton, where the definition of automaton is not particularly different from the traditional automaton. The automaton consists of states and transitions between the states, where the transitions have inputs and outputs.

States and Transitions for the Genuine CPC. We first create states and transitions that can generate the genuine CPC. Specifically, given a CPC, we tokenize the CPC to obtain a sequence of short strings (e.g., strings of 5~10 lengths). Then we add a state that can translate each token, and connect the individual states. The resulting automaton is the minimum automaton that can generate a CPC. We choose the input/output of state transitions by using a dictionary (e.g., an English dictionary). Specifically, we randomly pick two words for input (W_{in}) and output (W_{out}) of a transition. Then, to make sure that the transition can generate a desired token of CPC ($token_{cur}$), we obtain an input candidate for CPC by computing ($W_{in} - (W_{out} - token_{cur})$), which is essentially reversing the translation process.

Figure 5 shows an example. Given the same state transition used in Figure 4, we choose input and output from a dictionary. In this example, we concatenate two words, "unemployment" and "benefit" for input and "reassuring" and "investors" for output, as shown in Figure 5-(a). Then, given a token string, to translate shown in Figure 5-(b), we first compute $W_{out} - token_{cur}$ as shown in Figure 5-(c). We compute ($W_{in} - (W_{out} - token_{cur})$) as shown in (d). The outcome is the secret key that can generate the CPC token string ($token_{cur}$). Finally, we also run our ambiguous translator to check whether the secret input can generate the CPC token. Note that due to the rounding in the translation process (line 8 in Algorithm 1), some secret keys obtained by the above process cannot generate the CPC token string. If this happens, we choose another input/output pair and repeat the process until it succeeds.

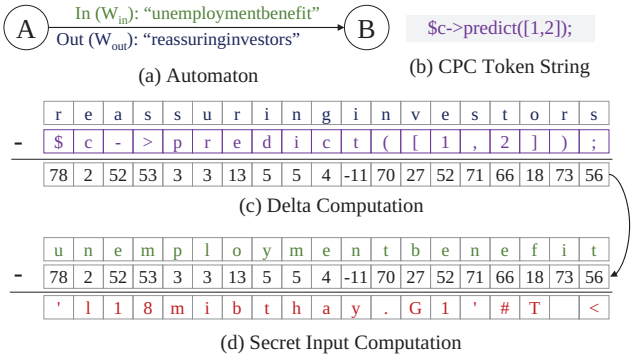


Figure 5: Identifying Secret Key for a CPC Token.

Unnecessary States and Transitions. We then add extra states and transitions between all states to hinder analysis attempts of the state machine. Dummy transitions connect *all states (not only dummy states)*, making Ambitr more difficult to analyze. Note that the dummy states and transitions are used to translate decoy (i.e., fake) CPCs. Inputs/outputs of the transitions to the dummy states are chosen in a way that the inputs of all transitions *look similar*, making it challenging to know which transitions are for the genuine CPC. Specifically, for each newly added transition, its input is derived by choosing a similar word (i.e., synonyms/antonyms in dictionaries [22, 58]) to its neighboring transition's input.

4 EVALUATION

In this section, we present various experimental results to show the effectiveness of Ambitr in comparison with existing state-of-the-art techniques and analysis tools. In particular, we evaluate Ambitr in terms of evasiveness (via dynamic analysis tools in Section 4.3.1), complexity (via static analysis tools in Section 4.3.2 and Section 4.3.3), and the context-sensitivity (Section 4.3.4).

Implementation. We implement our Ambitr creator in Python (1,322 LOC). It generates Ambitr, written in PHP (2,314 LOC excluding lines for the transition inputs and outputs).

***Ambiguous Translator* Configuration.** For the evaluation, *Ambiguous Translator* is configured to create binary samples with at least more than 300 nodes and each node has at least 5 edges.

Table 3: Ambitr Instances Statistics.

Size of CPCs (Avg.)	# of Samples	Avg. Size of Ambitr	Avg. # of States	Avg. # of Transitions
0~10 KB (2.7 KB)	345	27.36 KB	603.6	4,843.4
10~20 KB (14.5 KB)	99	77.72 KB	2,476.1	19,194.7
20~30 KB (24.0 KB)	39	130.15 KB	4,012.8	32,334.9
30~40 KB (34.7 KB)	56	175.81 KB	5,671.7	45,659.0
40~50 KB (43.7 KB)	34	209.71 KB	7,209.5	58,124.1

4.1 Applicability

To understand whether Ambitr can be created by various input/output pairs, we collect 573 code snippets and programs from popular repositories [7, 8, 23, 51, 74]. Note that for Ambitr, those input CPCs are simply strings, and values of the inputs do not affect Ambitr's performance.

We successfully generate Ambitr instances for all 573 collected samples as shown in Table 3. Given the secret input, they all successfully generate CPCs as expected. We categorize them by the samples' sizes (with an interval value of 10 KB). The sizes of Ambitr are larger than the original samples (we apply compression, e.g., gzip, to reduce the size of Ambitr). Except for the first group, the size of Ambitr is about 5 times larger than the original sample.

4.2 Automated Analysis of Ambitr

We compare Ambitr with state-of-the-art obfuscation/protector techniques to show Ambitr effectively hides CPCs. In particular, we use a forced execution technique MalMax [48] as it can effectively expose CPCs hidden by existing techniques (see Table 1).

Obfuscators/Protectors Selection. Four state-of-the-art PHP obfuscators and two crypters/protectors are chosen based on their popularity. Obfusactors include PHP Obfuscator [26], YAK Pro [36], Best PHP Obfuscator [60], and Simple Online PHP Obfuscator [39]. Crypters/protectors include Zend Guard [83] and PHP Encoder [59].

Result. As discussed in Section 1, obfuscators do not require any particular input or environment to decode and run the genuine CPC. Even without the forced execution technique (MalMax), we observe the CPC's execution by simply running them. For Zend Guard and PHP Encoder, it requires the encryption key to be accessible via network. We use MalMax to run the programs protected by Zend Guard and PHP Encoder, without encryption key access. Initially, they all fail to execute. Then, we try an incorrect key by creating another key from Zend Guard and PHP Encoder. The wrong key is essentially a key for another program. As expected, the wrong key results in failed executions for all samples because the existing techniques are not ambiguous (as shown in Figure 2).

Then, we use a correct key (obtained by tracing network communications when it runs without errors). We run MalMax again with the correct key, and all samples are successfully decrypted and expose CPCs. As discussed in Section 2.4.4, the fact that it can successfully execute indicates that the identified CPCs are genuine.

We also use MalMax to analyze Ambitr protected samples. However, MalMax fails to expose any of CPCs from the samples. This is because MalMax focuses on executing all statements without precisely identifying the key secret inputs. Simply executing all statements of a target is not sufficient for analyzing Ambitr. Moreover, while the execution of Ambitr under MalMax is incorrect, Ambitr does not cause any errors or observable behavior differences. Some generated outputs are not valid while there are still many seemingly valid outputs looks like CPCs, causing ambiguity in analysis. Even one can observe the genuine CPC (e.g., having a network trace of the input leading to the genuine CPC), knowing whether the observed CPC is the original is not verifiable.

4.3 Reverse Engineering Ambitr

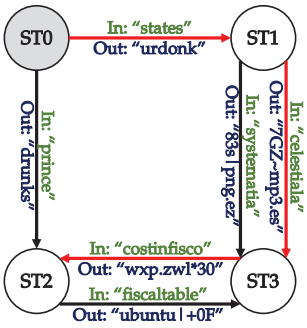
We evaluate Ambitr from a reverse-engineer's perspective in terms of how difficult to reveal the genuine CPC using various program analysis tools manually. In the following subsections, we

```php
for($i = 0; $i < ...; $i++)
    define("ST$i", $i);
define("CUR_STATE",  0);
define("INPUT",      1);
define("OUTPUT",     2);
define("NEXT_STATE", 3);

$state_tbl = array(
/* array(CURRENT_STATE,
         INPUT OF THE TRANSITION,
         OUTPUT OF THE TRANSITION,
         NEXT STATE) */
array(ST0, "states", "urdonk", ST1),
array(ST0, "prince", "drunks", ST2),
array(ST1, "celestiala", "7GZ~mp3.es", ST3),
array(ST1, "systematia", "83s|png.ez", ST3),
array(ST3, "costinfisco", "wxp.zwl*30", ST2),
array(ST2, "fiscaltable", "ubuntu|+0F", ST3),
...
);
```

```php
$cur_state = ST0;
while(1) {
  $tran = find_transition($state_tbl, $cur_state, $input);
  $cur_state = $state_tbl[$tran][NEXT_STATE];
  $result .= dynamic_translate( $state_tbl[$tran][INPUT],
                                $state_tbl[$tran][OUTPUT],
                                $input);
}
function dynamic_translate($in, $out, $cur_in) {
  for( $i = 0; $i < strlen($cur_in); $i++)
    $ret .= chr(ord($out[$i]) -
              (ord($in[$i]) - ord($cur_in[$i])));
  return $ret;
}
function find_transition($state_tbl, $cur_state, $in) {
  foreach ( $state_tbl[...] as $next_transition )
    for ( $j = 0; $j < strlen($in); $j++ )
      $d[...] += abs(ord($state_tbl[...][$j]) - ord($in[$j]));
  return index( min( $delta ) );
}
```

(a) Source Code of the Example Ambiguous Translator



(b) State Machine
(Red arrows are the transitions generate the genuine CPC. ST0 is an initial state)

Figure 6: Simplified Source Code of *Ambiguous Translator* (in PHP).

assume that the reverser obtains a sample of Ambitr without knowing the intended input that generates output.

4.3.1 Dynamic Analysis. We assume a scenario that the reverser attempts to use `Xdebug` [21] to monitor its execution. `Xdebug` is a PHP debugging extension, providing various debugging primitives such as step-debugging (i.e., single-stepping), variable dumps, and stack traces. Specifically, it traces variables that are used to compute outputs from inputs [20], similar to program slicing [1, 77].

Analyzing Executed Statements. The reverser 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 (e.g., as shown in Figure 6-(a)), the resulting traces include most of the statements regardless of whether the execution delivers an attack or not.

Analyzing Values from Executed Statements. The reverser also dumps all the values of the variables used in the executed statements. However, as the execution does not deliver the genuine CPC, analyzing the values does not help.

4.3.2 Static Analysis. Static analysis tools can be used to analyze Ambitr to identify possible output values that can be generated by *Ambiguous Translator*. Specifically, the reverser uses static taint analysis tools to find out the data flow of *Ambiguous Translator*. Further, static analysis tools that can conduct a value-set analysis (e.g., [4]) are used to infer possible values of a few key variables.

Simplified Source Code of *Ambiguous Translator*. Figure 6-(a) shows a simplified version of *Ambiguous Translator* written in PHP. Lines 1-6 define constants. Lines 8-20 build a state transition table that is essentially an array of state transition rules including current state, input/output of the transition, and next state (line 13-18). It has a loop (lines 22-28) that repeatedly finds a transition according to the input (line 23), makes the transition (line 24), and dynamically creates an output according to the input (line 25). The dynamic translation is done in a function (lines 29-34). The result is essentially a concatenated string of the dynamic outputs (line 31). Figure 6-(b) shows the ground-truth of *Ambiguous Translator* shown in Figure 6-(a). It has four states (`ST0` ~ `ST3`) and there are multiple transitions among `ST1`, `ST2` and `ST3`.

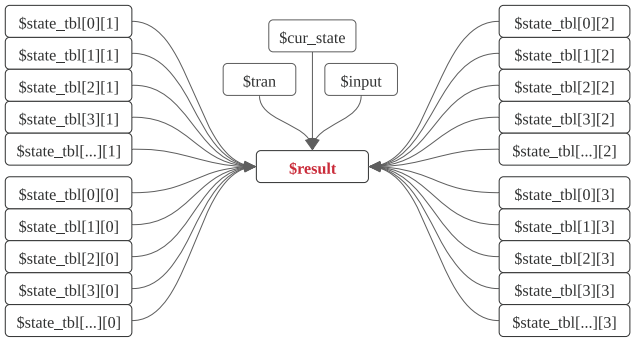


Figure 7: Data Dependency Graph by Taint Analysis.

Backward Data Slicing via Taint Analysis. There are several PHP static analysis tools that support taint analysis: Pixy [33], Eir [27], Taint'em All [81], and TaintPHP [53]. Note that most of them do not properly propagate taint tags through array and array index operations. Hence, we reimplement Figure 6 without using arrays so that they can effectively analyze Ambitr. Moreover, TaintPHP [53] does not support inter-procedure analysis; hence we inline all functions (e.g., `dynamic_translate()`) in Ambitr and feed it to TaintPHP. To this end, the reverser leverages the above four taint analysis tools to obtain a data dependency graph shown in Figure 7. It essentially shows that the value of `$result` is computed by `$out` that is again *dependent on all the variables* including `$input`, `$state_tbl` arrays, `$tran`, and `$cur_state`. While this is accurate, the result is too coarse-grained. Specifically, it shows all the `$state_tbl` arrays are contributing the value of `$result`. It does not provide a particular order of state transitions which is critical in revealing attack delivering inputs. Note that one may improve the analysis to better support arrays (i.e., array-sensitive analysis). However, while array-sensitive analysis can improve the granularity of the analysis (i.e., identifying data-dependencies at an element level), it still provides the same information and does not help identify the real CPC.

Value-set Analysis. The reverser uses three static analysis tools for PHP: PHPStan [45], Psalm [73], and WeVerca [28]. The tools

Table 4: Value-set Analysis Result for Key Variables.

Variable	Value-Set
`$tran`	`{0, 1, 2, 3, ... }`
`$cur_state`	`{ST0, ST1, ST2, ST3, ... }`
`$ret`	`{"urtonk", "drunks", "7GZ~mp3.gs", "83s|png.gz" "wxp.zy|*3F", "ubuntu|+0F", ... }`
`$result`	Combinations of values of `$ret`

implement a data flow analysis technique that can be used to build value-set analysis, which identifies a set of possible values a variable can have during the execution [4]. The reverser leverages them to infer potential values that each variable can hold in the Ambitr instance shown in Figure 6-(a). Table 4 shows the result of the value-set analysis on each key variable in Figure 6-(a). In short, the result is not an effective way to expose the genuine CPC due to two reasons. First, while the analysis reveals *all possible inputs* for `$ret` and `$result`, it simply dumps all the outputs of the transitions in *Ambiguous Translator*. To analyze *Ambiguous Translator*, one has to understand the order of outputs generated by transitions rather than a set of outputs. Second, even for the revealed outputs stated in *Ambiguous Translator* as shown in Table 4, they are misleading. Those outputs are not the ones that will be generated when an attack delivering input is provided. For instance, the Ambitr instance in Figure 6-(a) can deliver a code snippet `unlink('/tmp/.found.txt');` when a sequence of inputs `spines`, `TEA[steam]`, and `aegtconfine` are provided. The inputs dynamically transform the outputs annotated on the transitions (i.e., `urdonk`, `7GZ~mp3.es`, and `wxp.zwl*30`) into the code snippets (i.e., `unlink`, `('/tmp/.fo`, and `und.txt');` respectively).

4.3.3 Symbolic Execution Tools. In this section, the reverser uses symbolic execution tools to reverse-engineer the genuine CPC translation logic of Ambitr. Specifically, four symbolic execution tools, THAPS [31], PHPScan [72], KPHP [25], and Symex [50] are used. The tools aim to identify all possible inputs that can lead to new program execution paths or states. Note that a non-array version of Ambitr is used, as the symbolic executions fail to support array properly.

State Explosion. None of the symbolic execution tools we used finishes the analysis in a week due to state explosion [10, 15, 71]. Specifically, for each state, *Ambiguous Translator* has multiple transitions to the next states. Hence, the number of possible transition paths grows exponentially. For instance, suppose the input has x words requiring x state transitions, there will be 5^x possible transition paths, leading to state explosion. KPHP [25] crashed after running 7 hours 17 minutes due to insufficient memory. Further, we create a simplified version of Ambitr that has a single transition with a 4-byte input for each transition. The four symbolic execution tools failed to finish the analysis within a week as well.

Experiments with Enhanced PHPScan. Since the vanilla versions of symbolic execution tools failed to analyze a very small instance of Ambitr (with a single transition), we manually optimize PHPScan [72] and use it to analyze Ambitr. Specifically, we modify PHPScan so that it can (1) *cache and reuse solved constraints*, and (2) *merge and reduce multiple constraints into fewer constraints*. We use a machine with Intel i7-8550U 4.0 GHz and 16 GB RAM to run this experiment. We conduct two experiments. We prepare Ambitr instances with (1) different numbers of transitions where each transition will take 3-byte input and (2) a single transition but with different input lengths.

1) Different numbers of transitions: As the number of transitions increases, the number of states to explore is increased exponentially. For instance, with a single additional transition, the number of states becomes 10 times larger. We prepare simplified versions of Ambitr that have 4, 5, and 6 transitions where each transition takes 3 characters long input. We use PHPScan to analyze them. It takes about 3 hours, 2 days, and 4 weeks to finish the analysis of *Ambiguous Translator* with 4, 5, and 6 transitions, respectively. Note that the input length (currently 3) is a root cause of state explosion. In this example, we set it 3 for each transition.

2) Different input lengths: Dynamic output translation also causes the state explosion. To understand its impact on the number of states during the symbolic execution, we create a simplified version of Ambitr with input lengths of 6, 7 and 8. Analyzing a single transition for the input length 6 (i.e., 6 characters input) takes about 15 hours 30 minutes. Input lengths 7 and 8, which are typical lengths of inputs in our samples, take more than 2.9 days and 13 days to finish the analysis, respectively. This shows that analyzing even *a single transition* is time-consuming.

Optimization Causing Under-approximation. Symbolic analysis, in practice, uses an optimization strategy that aims to find *one* input that drives the execution to a particular point instead of enumerating all possible inputs. As a result, even the reverser reaches a particular state, the identified input is unlikely an attack delivering input. For example, in Figure 8-(a), the array `$fn` represents a function name. Before it's invoked at line 6, it is constructed at lines 3–5 after satisfying multiple path conditions at line 2. Symbolic analysis encodes the path conditions and gets one solution shown in Figure 8-(b) from the underlying constraint solver. The execution successfully goes into the true branch and invokes the function `$f`. However, it invokes function `uniqid` instead of function `unlink` that constitutes the genuine CPC as shown in Figure 8-(c). Given this branch has been successfully explored, the symbolic analysis will not try other solutions satisfying the path condition and thus cannot discover the genuine CPC.

```
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**

Figure 8: Symbolic Execution Exploring a Single Input.

Describing Constraints. Although the reverser can use symbolic analysis to model the path predicates as constraints and drive the execution to a particular program location, it is challenging to *explicitly encode the criteria of the genuine CPC as constraints* (e.g., constraints that describe the CPC). In other words, he may not even know what exactly he is looking for and how to describe the logic in a way the underlying constraint solver can understand. For

example, it is challenging because any valid statements and function names can be potentially CPC. As a result, the satisfiable solutions to the incomplete constraints may lead to a place of interest but will not reveal the genuine CPC.

Empirical Experiment on the State Explosion. To understand how difficult to analyze Ambitr with symbolic execution techniques in detail, we run experiments with PHPScan [72] which uses the z3 solver [43] for constraint solving. Note that the original version of PHPScan was too inefficient. It failed to finish the analysis on a very small Ambitr sample (e.g., a single transition of 4 characters long input/output) in 24 hours. Hence, we manually improve the PHPScan's performance by modifying it to (1) cache and reuse already solved constraints, and (2) merge and reduce multiple identical constraints into fewer constraints. We run the enhanced version of PHPScan on a machine with Intel i7-8550U 4.0 GHz and 16 GB RAM.

Table 5 shows the experiment results. As shown in the "Automaton Size" columns, we created 16 different sizes *Ambiguous Translator*. The size is defined as a pair of the length of input characters and the number of the transition. For instance, "6 chars., 1 trans." means a *Ambiguous Translator* that has a single transition between two states, and the transition input/output is 6 characters long. An example can be a sub state-machine of Figure 9 between `ST_0` and `ST_1` (Input: "states", Output: "urdonk"). The "# Const." columns present the number of constraints that should be explored by PHPScan. The "Time" columns show the required time for the analysis. Note that as the *Ambiguous Translator* gets bigger, the number of constraints increases exponentially. In many cases (i.e., the gray cells), the experiments did not finish even after 10 days. For those cases, we estimate the required time based on the number of processed constraints and remaining (also estimated) constraints. Observe that the enhanced version of PHPScan takes more than 10 days to analyze *Ambiguous Translator* instances with more than 3 transitions of 5 characters input/output (which is much smaller than typical *Ambiguous Translator* we generated and used).

Figure 9 consists of 17 transitions and its average input/output size is 9.64, which is much larger than the largest *Ambiguous Translator* presented in Table 5 (4 transitions of 7 characters long input/output). Note that even if the analysis successfully finishes, the analysis results (e.g., inputs to make all possible transitions) do not expose the genuine CPC.

Table 5: PHPScan on Different Sizes of Ambitr

Automaton Size (Input, Trans.)	# of Const.	Time	Automaton Size (Input, Trans.)	# of Const.	Time
4 chars., 1 trans.	35 K	31.5 m	6 chars., 1 trans.	10 M	14.8 h
4 chars., 2 trans.	386 K	4.3 h	6 chars., 2 trans.	112 M	4.7 d
4 chars., 3 trans.	6 M	8.1 d	6 chars., 3 trans.	1.2 B	149.3 d
4 chars., 4 trans.	61 M	121 d	6 chars., 4 trans.	12 B	5.7 y
5 chars., 1 trans.	181 K	2.9 h	7 chars., 1 trans.	4 M	3.5 d
5 chars., 2 trans.	1.9 M	19.8 h	7 chars., 2 trans.	4.5 B	26.3 d
5 chars., 3 trans.	31 M	37.8 d	7 chars., 3 trans.	49 B	1.6 y
5 chars., 4 trans.	315 M	1.4 y	7 chars., 4 trans.	498 B	20 y

Gray cells indicate that the experiments did not finish in 10 days. The times presented for them are estimated based on the performance measured in the first ten days of execution.

4.3.4 Source Code and Input Analysis. We aim to show how the reverser would make manual reverse-engineering attempts to find

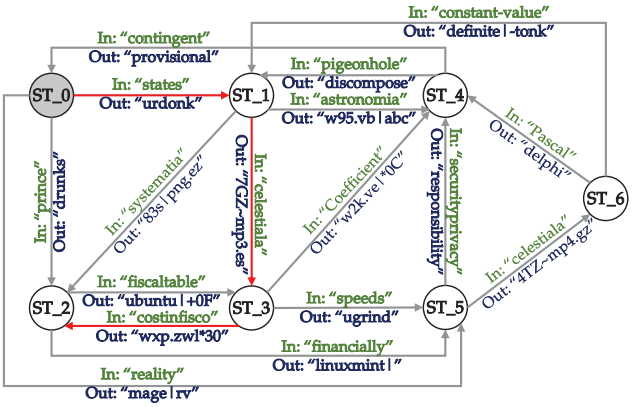


Figure 9: Simplified *Ambiguous Translator* used in the Input Analysis.

	Input	Output	Transitions
Initial Input	states celestials cost-effective fiscal-year	urdonk 7GZ~mp3.e wxp.>nl'%0ive ubuntu5C3;	ST_0→ST_1→ST_3 →ST_2→ST_3
First Trial	realm; celestials cost-effective fiscal-year	sWgmmP 4TZ~mp4.g dekj'mlWl ok +#sklm33azr	ST_0→ST_5→ST_6 →ST_1→ST_2
Second Trial	province celestials cost-effective fiscal-year	dr{vq| iely}~int boo*jeVWZw\{ Zaak5ylar	ST_0→ST_2→ST_5 →ST_4→ST_1
Third Trial	states planetary cost-effective fiscal-year	urdonk 5&avpug,u ios7ofsmsive ei^^q,iWXu	ST_0→ST_1→ST_2 →ST_5→ST_4
Fourth Trial	states astronomical cost-effective fiscal-year	urdonk w95.vb|abeal prtv-jhpe Pive Zcizear	ST_0→ST_1→ST_4 →ST_0→ST_2

Figure 10: Inputs used during the Input Analysis.

out the secret input leading to the genuine CPC in Ambitr by manually inspecting source code and guessing inputs. We assume the reverser obtained a sample of Ambitr and reverse-engineered *Ambiguous Translator* as shown in Figure 9. Then, the reverser executes the sample and identifies input that the sample retrieves. The input are shown in the first row of Figure 10 (Initial Input). As expected, the input does not lead to the genuine CPC. To this end, the reverser tries to guess inputs leveraging knowledge gained from manual source code inspection.

The reverser modifies the first input by guessing a possible alternative word. Specifically, `realm;` is chosen as it is a synonym for `states`, the original input. Note that all other inputs remain unchanged. However, since the first input leads to a different transition (`ST_0→ST_5`), *all the subsequent transitions* (shown in the last column) *are different from the transitions for the initial input*, resulting in a completely different output. In the second trial, the reverser changes the first input to `province`, which is another synonym for `states`. Again, the output and the transitions are changed significantly, leaving no particular hints for the next trial. From the third trial, the reverser starts to guess the second input. Specifically, `planetary` is used. Observe that the first output word remains the same, while all the subsequent outputs and transitions are changed. While this shows that the first input is related to the first output, it is not useful in reverse-engineering the attack delivering the input. The fourth trial is similar. Changing a single word in the input leads to all subsequent output words, and transitions being changed.

5 DISCUSSION

Generality. While we implement our prototype in PHP, the idea is general and can be implemented in other programming languages. To support executable CPCs, one needs to implement dynamic code generation and execution primitives such as `eval()`. Script languages such as JavaScript and Python support them by default. In other programming languages such as C/C++, one may leverage JIT compilation techniques [11].

Handling Non-ASCII Inputs and Outputs. For better readability, we only discuss example cases when inputs and outputs are ASCII characters. However, Ambitr seamlessly supports non-ASCII inputs and outputs. Specifically, if the input is out of range of ASCII characters, Ambitr calculates the distance of provided input and the state transition's input without converting them to ASCII code value. Similarly, Ambitr computes the output directly from the distance values and state transition output without considering their ASCII values.

Threats to Validity. The experiments in Section 4.2 are conducted by two individuals who have sufficient background in computer science using state-of-the-art open-source tools. Specifically, the experiment presented in Section 4.3.3 is conducted by a computer science Ph.D. student with sufficient program analysis and security background. The work in Section 4.3.4 is done by an expert in software engineering and security (holding a Ph.D. in Computer Science). In addition, two undergraduate students majoring in Computer Science (focusing on computer security) have repeated the experiments and reached the same conclusions. Note that all participants did not know the proposed approach prior to the experiment. The analysis results may differ depending on the tools' capability and the analysts' expertise.

6 RELATED WORK

Hiding Program Code. There exists a line of work in obfuscation to hide program code leveraging opaque predicates [16, 47, 67], code insertion/replacement [5, 37, 54, 61, 76], encryptions [66, 75], hardware primitives [12, 64], and sub-tree embedding [24]. However, opaque predicates can be detected and removed via advanced program analysis techniques [44]. Dummy code snippets inserted into an existing program can be identified and removed via dependency analysis such as taint analysis [17, 27, 33, 42, 52, 53, 55, 56, 63, 65, 81].

Anti-analysis Techniques. Recently, [54] presents a systematic study of multiple methods to hinder symbolic execution techniques. Specifically, it inserts additional code to increase the number of feasible paths. Ambitr's *Ambiguous Translator* not only increases the number of feasible paths but also provides many more additional challenges such as ambiguity via dynamic output translation. [24] transforms program code snippets into a sub abstract syntax tree (AST), and injects the tree into the AST of a program. However, dynamic analysis and symbolic analysis tools can detect such injected code. Data obfuscations (e.g., encrypting code sections and decryption them at runtime) are easily handled by dynamic analysis [9, 41, 68]. Approaches that require particular hardware support are difficult to be used in real-world program, as many systems may not satisfy the hardware requirement. Unlike them, Ambitr is challenging to be analyzed by static, symbolic, and dynamic analysis tools as shown in Section 4. It does not require any particular hardware or software.

7 CONCLUSION

Protecting critical program components (e.g., patented program logic or sensitive data) is an important requirement in software systems. In this paper, we present Ambitr, a novel technique that hides critical program components via a sophisticated state machine based translator called *Ambiguous Translator*. It imposes fundamental challenges to state-of-the-art program analysis techniques by adding a new dimension of the challenge: ambiguity. Our evaluation of the comparison with a diverse set of state-of-the-art analysis techniques, including dynamic, static, and symbolic execution, shows that Ambitr is effective in hiding critical program components.

ACKNOWLEDGMENTS

We thank the anonymous referees for their constructive feedback. The authors gratefully acknowledge the support of NSF 1916499, 1908021, 2047980, 1850392, 1853374, 1924777, 2145616, and 2047980. This research was also partially supported by a Mozilla Research Award, a Facebook Research Award, and a gift from Cisco Systems. 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] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. *SIGPLAN Not.* 25, 6 (June 1990), 246–256. https://doi.org/10.1145/93548.93576
[2] Christian Ammann. 2012. Hyperion: Implementation of a PE-Crypter.
[3] David E Bakken, R Rarameswaran, Douglas M Blough, Andy A Franz, and Ty J Palmer. 2004. Data obfuscation: Anonymity and desensitization of usable data sets. *IEEE Security & Privacy* 2, 6 (2004), 34–41.
[4] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 5–23.
[5] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code Obfuscation against Symbolic Execution Attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/2991079.2991114
[6] Cristian Barría, David Cordero, Claudio Cubillos, and Robinson Osses. 2016. Obfuscation procedure based in dead code insertion into crypter. In *2016 6th International Conference on Computers Communications and Control (ICCCC)*. IEEE, 23–29.
[7] BDLeet. 2016. GitHub - BDLeet/public-shell: Some Public Shell. https://github.com/BDLeet/public-shell.
[8] Bart Blaze. 2019. GitHub - bartblaze/PHP-backdoors: A collection of PHP backdoors. https://github.com/bartblaze/PHP-backdoors.
[9] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. 2008. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. Springer, 65–88.
[10] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. 1992. Symbolic model checking: 1020 states and beyond. *Information and computation* 98, 2 (1992), 142–170.
[11] Juan Manuel Martinez Caamaño and Serge Guelton. 2018. Easy::Jit: Compiler Assisted Library to Enable Just-in-Time Compilation in C++ Codes. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming* (Nice, France) *(Programming'18 Companion)*. Association for Computing Machinery, New York, NY, USA, 49–50. https://doi.org/10.1145/3191697.3191725
[12] Haibo Chen, Liwei Yuan, Xi Wu, Binyu Zang, Bo Huang, and Pen-chung Yew. 2009. Control flow obfuscation with information flow tracking. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 391–400.
[13] Binlin Cheng, Jiang Ming, Jianmin Fu, Guojun Peng, Ting Chen, Xiaosong Zhang, and Jean-Yves Marion. 2018. Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 395–411. https://doi.org/10.1145/3243734.3243771

[14] Binlin Cheng, Jiang Ming, Erika A Leal, Haotian Zhang, Jianming Fu, Guojun Peng, and Jean-Yves Marion. 2021. Obfuscation-Resilient Executable Payload Extraction From Packed Malware. In *30th USENIX Security Symposium (USENIX Security 21)*.

[15] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. 2011. Model checking and the state explosion problem. In *LASER Summer School on Software Engineering*. Springer, 1–30.

[16] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 184–196.

[17] Johannes Dahse and Jörg Schwenk. 2010. RIPS-A static source code analyser for vulnerabilities in PHP scripts. *Retrieved: February* 28 (2010), 2012.

[18] Biniam Fisseha Demissie, Mariano Ceccato, and Roberto Tiella. 2015. Assessment of Data Obfuscation with Residue Number Coding. In *Proceedings of the 1st International Workshop on Software Protection* (Florence, Italy) *(SPRO '15)*. IEEE Press, 38–44.

[19] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2015. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 44–56.

[20] Derick Rethans. 2009. Variable tracing with Xdebug — Derick Rethans. https://derickrethans.nl/variable-tracing-with-xdebug.html.

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

[22] dwyl. 2019. A text file containing 479k English words. https://github.com/dwyl/english-words.

[23] Evi1cg. 2019. GitHub - Ridter/Pentest. https://github.com/Ridter/Pentest.

[24] Aurore Fass, Michael Backes, and Ben Stock. 2019. Hidenoseek: Camouflaging malicious javascript in benign asts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1899–1913.

[25] Daniele Filaretti and Sergio Maffeis. 2014. An executable formal semantics of PHP. In *European Conference on Object-Oriented Programming*. Springer.

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

[27] Heilan Yvette Grimes. 2015. Eir - Static Vulnerability Detection in PHP Applications. (2015).

[28] David Hauzar and Jan Kofroň. 2014. WeVerca: Web Applications Verification for PHP. In *International Conference on Software Engineering and Formal Methods*. Springer, 296–301.

[29] Cristian Barría Huidobro, David Cordero, Claudio Cubillos, Héctor Allende Cid, and Claudio Casado Barragán. 2018. Obfuscation procedure based on the insertion of the dead code in the crypter by binary search. In *2018 7th International Conference on Computers Communications and Control (ICCCC)*. IEEE, 183–192.

[30] Imperva. 2021. Data Obfuscation. https://www.imperva.com/learn/data-security/data-obfuscation/.

[31] Torben Jensen, Heine Pedersen, Mads Chr Olesen, and René Rydhof Hansen. 2012. Thaps: automated vulnerability scanning of php applications. In *Nordic conference on secure IT systems*. Springer, 31–46.

[32] Ryan Johnson and Angelos Stavrou. 2013. Forced-path execution for android applications on x86 platforms. In *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*. IEEE, 188–197.

[33] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 6–pp.

[34] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A Hidden Code Extractor for Packed Executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode* (Alexandria, Virginia, USA) *(WORM '07)*. Association for Computing Machinery, New York, NY, USA, 46–53. https://doi.org/10.1145/1314389.1314399

[35] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-force: Forced execution on javascript. In *Proceedings of the 26th international conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 897–906.

[36] Pascal Kissian. 2019. YAK Pro: Php Obfuscator. https://www.php-obfuscator.com/.

[37] Byoungyoung Lee, Yuna Kim, and Jong Kim. 2010. binOb+: a framework for potent and stealthy binary obfuscation. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 271–281.

[38] Young Bi Lee, Jae Hyuk Suk, and Dong Hoon Lee. 2021. Bypassing Anti-Analysis of Commercial Protector Methods Using DBI Tools. *IEEE Access* 9 (2021), 7655–7673.

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

[40] Alessandro Mantovani, Simone Aonzo, Xabier Ugarte-Pedrero, Alessio Merlo, and Davide Balzarotti. 2020. Prevalence and Impact of Low-Entropy Packing Schemes in the Malware Ecosystem. In *Network and Distributed System Security (NDSS) Symposium, NDSS*, Vol. 20.

[41] Jian Mao, Jingdong Bian, Guangdong Bai, Ruilong Wang, Yue Chen, Yinhao Xiao, and Zhenkai Liang. 2018. Detecting malicious behaviors in javascript applications. *IEEE Access* 6 (2018), 12284–12294.

[42] Ibéria Medeiros, Nuno F Neves, and Miguel Correia. 2014. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 23rd international conference on World wide web*. ACM, 63–74.

[43] Microsoft. 2020. Z3Prover/z3: The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[44] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. Loop: Logic-oriented opaque predicate detection in obfuscated binary code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 757–768.

[45] Ondřej Mirtes. 2019. GitHub - phpstan/phpstan: PHP Static Analysis Tool. https://github.com/phpstan/phpstan.

[46] Shoya Morishige, Shuichiro Haruta, Hiromu Asahina, and Iwao Sasase. 2017. Obfuscated malicious javascript detection scheme using the feature based on divided url. In *2017 23rd Asia-Pacific Conference on Communications (APCC)*. IEEE, 1–6.

[47] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE, 421–430.

[48] Abbas Naderi-Afooshteh, Yonghwi Kwon, Anh Nguyen-Tuong, Ali Razmjoo-Qalaei, Mohammad-Reza Zamiri-Gourabi, and Jack W Davidson. 2019. MalMax: Multi-Aspect Execution for Automated Dynamic Web Server Malware Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1849–1866.

[49] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software.. In *NDSS*, Vol. 5. Citeseer, 3–4.

[50] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. 2011. Auto-locating and fix-propagating for HTML validation errors to PHP server-side code. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 13–22.

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

[52] Paulo Jorge Costa Nunes, José Fonseca, and Marco Vieira. 2015. phpSAFE: A security analysis tool for OOP web application plugins. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

[53] Oswaldo Olivo. 2016. GitHub - olivo/TaintPHP: Static Taint Analysis for PHP web applications. https://github.com/olivo/TaintPHP.

[54] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. 2019. How to Kill Symbolic Deobfuscation for Free (or: Unleashing the Potential of Path-Oriented Protections). In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 177–189. https://doi.org/10.1145/3359789.3359812

[55] OneSourceCat. 2015. GitHub - OneSourceCat/phpvulhunter: A tool that can scan php vulnerabilities automatically using static analysis methods. https://github.com/OneSourceCat/phpvulhunter.

[56] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. 2011. PHP Aspis: using partial taint tracking to protect against injection attacks. In *2nd USENIX Conference on Web Application Development*, Vol. 13.

[57] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-force: force-executing binary programs for security applications. In *23rd USENIX Security Symposium*. 829–844.

[58] PHP. 2019. PHP: Pspell Functions. https://www.php.net/manual/en/ref.pspell.php.

[59] phpencoder 2021. PHP Encoder, protect PHP scripts with SourceGuardian and bytecode. https://www.sourceguardian.com/.

[60] Pipsomania. 2018. Best PHP Obfuscator. http://www.pipsomania.com/best_php_obfuscator.do

[61] Igor V Popov, Saumya K Debray, and Gregory R Andrews. 2007. Binary Obfuscation Using Signals. In *USENIX Security Symposium*. 275–290.

[62] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. 2006. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. 289–300. https://doi.org/10.1109/ACSAC.2006.38

[63] Dewhurst Ryan. 2011. 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* 2012 (2011).

[64] Sebastian Schrittwieser, Stefan Katzenbeisser, Peter Kieseberg, Markus Huber, Manuel Leithner, Martin Mulazzani, and Edgar Weippl. 2013. Covert computation: Hiding code in code for obfuscation purposes. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 529–534.

[65] Design Security. 2016. GitHub - designsecurity/progpilot: A static analysis tool for security. https://github.com/designsecurity/progpilot.
[66] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation.. In *NDSS*.
[67] Brendan Sheridan and Micah Sherr. 2016. On Manufacturing Resilient Opaque Constructs Against Static Analysis. In *European Symposium on Research in Computer Security*. Springer, 39–58.
[68] Guillermo Suarez-Tangil, Juan E Tapiador, Flavio Lombardi, and Roberto Di Pietro. 2014. Thwarting obfuscated malware via differential fault analysis. *Computer* 47, 6 (2014), 24–31.
[69] themida 2021. Oreans Technologies. https://www.oreans.com/Themida.php.
[70] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo Bringas. 2016. RAMBO: Run-Time Packer Analysis with Multiple Branch Observation. 186–206. https://doi.org/10.1007/978-3-319-40667-1_10
[71] Antti Valmari. 1998. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*. Springer-Verlag, London, UK, UK, 429–528. http://dl.acm.org/citation.cfm?id=647444.727054
[72] Bart van Arnhem. 2017. GitHub - bartvanarnhem/phpscan: Symbolic execution inspired PHP application scanner for code-path discovery. https://github.com/bartvanarnhem/phpscan.
[73] Vimeo. 2019. GitHub - vimeo/psalm: A static analysis tool for finding errors in PHP applications. https://github.com/vimeo/psalm.
[74] VirusShare. 2019. VirusShare.com. https://virusshare.com/.
[75] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. 2011. Linear obfuscation to combat symbolic execution. In *European Symposium on Research in Computer Security*. Springer, 210–226.
[76] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. 2011. Linear Obfuscation to Combat Symbolic Execution. In *Proceedings of the 16th European Conference on Research in Computer Security* (Leuven, Belgium) *(ESORICS'11)*. Springer-Verlag, Berlin, Heidelberg, 210–226.
[77] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering* (San Diego, California, USA) *(ICSE '81)*. IEEE Press, 439–449.
[78] Jeffrey Wilhelm and Tzi-cker Chiueh. 2007. A forced sampled execution approach to kernel rootkit identification. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 219–235.
[79] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 921–937.
[80] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 732–744. https://doi.org/10.1145/2810103.2813663
[81] Quan Yang. 2019. GitHub - quanyang/Taint-em-All: A taint analysis tool for the PHP language. https://github.com/quanyang/Taint-em-All.
[82] yodap 2021. Yoda's Protector. https://sourceforge.net/projects/yodap/.
[83] zendguard 2021. Protect PHP Code With Zend Guard. https://www.zend.com/products/zend-guard.