Security Analysis of PHP Bytecode Protection Mechanisms

Dario Weißer, Johannes Dahse ✉, and Thorsten Holz

Horst Görtz Institute for IT-Security (HGI),
Ruhr-University Bochum, Germany
{firstname.lastname}@rub.de

Abstract. PHP is the most popular scripting language for web applications. Because no native solution to compile or protect PHP scripts exists, PHP applications are usually shipped as plain source code which is easily understood or copied by an adversary. In order to prevent such attacks, commercial products such as *ionCube*, *Zend Guard*, and *Source Guardian* promise a source code protection.
In this paper, we analyze the inner working and security of these tools and propose a method to recover the source code by leveraging static and dynamic analysis techniques. We introduce a generic approach for decompilation of obfuscated bytecode and show that it is possible to automatically recover the original source code of protected software. As a result, we discovered previously unknown vulnerabilities and backdoors in 1 million lines of recovered source code of 10 protected applications.

Keywords: Security, Reverse Engineering, Obfuscation, PHP, Bytecode

1 Introduction

Protecting intellectual property (IP) in software systems, such as algorithms, cryptographic keys, serial numbers, or copyright banners, is a challenging problem: an adversary can study the program with static or dynamic analysis methods [7, 13, 19] and attempt to deduce the sensitive information. To impede such an analysis, many different types of obfuscation techniques for binary executables were developed (e.g., [3, 11, 15, 21]). Although the semantics of the program can be reconstructed with different (automated) reverse engineering methods [4,16,20,29], obfuscation provides at least some protection of the source code and hampers an adversary to a certain extent.

IP protection is more challenging in the web context: PHP, the most popular server-side scripting language on the web, is an interpreted language. This implies that an interpreter transforms the PHP source code on demand into bytecode that is then executed. As such, an adversary who can obtain access to the source code (e.g., via software bugs or a legitimate trial version) can directly study or modify the code and reveal sensitive information. To remedy such attacks, different tools are available that offer code protection: commercial products like *ionCube*, *Zend Guard*, and *Source Guardian* promise to "prevent unlicensed use and reverse engineering and to safeguard intellectual property" [30].

All these tools follow the same methodology: They pre-compile PHP source code into obfuscated bytecode that can then be shipped without the original source code. On the server side, these tools require a PHP extension that allows to run the bytecode. Popular PHP software such as *NagiosFusion*, *WHMCS*, and *xt:Commerce* ship certain files protected with such tools to safeguard their IP. As a result, an adversary can at most access the pre-compiled bytecode and cannot directly access the source code. Unfortunately, it is not documented how these products work internally and what security guarantees they provide.

In this paper, we address this gap. We study the three most popular commercial PHP code protection products in detail and analyze their security properties. We find that all share the same limitation that enables an adversary to reconstruct the semantics of the original code. More specifically, we introduce methods to recover the code by statically and dynamically analyzing the interpretation of the bytecode. Since the interpreter needs to transform the encrypted/obfuscated bytecode back to machine code, we can recover the semantic information during this phase. We found that all tools can be circumvented by an adversary and we are able to successfully reconstruct the PHP source code. In this paper, we first present our findings from manually reverse engineering the different PHP code protection tools. Based on these findings, we introduce our method to break the encryption and obfuscation layers using dynamic analysis techniques, and show how to build a generic decompiler. Note that our techniques can be used against all PHP bytecode protectors that rely on bytecode interpretation and our method is not limited to the three analyzed products.

To evaluate our decompiler, we studied several popular protected software programs. We uncovered critical vulnerabilities and backdoors in some of these protected programs that would have remained invisible without decompilation since the identified flaws were hidden in obfuscated/encrypted bytecode. Furthermore, we detected critical security vulnerabilities in the products themselves that weaken the encrypted application's server security. In conclusion, our results indicate that PHP source code protection tools are not as strong as claimed by the vendors and such tools might even lead to an increased attack surface.

In summary, we make the following contributions in this paper:

- We analyze and document in detail the inner working of the three most popular PHP bytecode protectors.
- We propose a method to generically circumvent such protectors based on the insight that we can recover the semantics of the original code during the interpretation phase. We present an automated approach to reconstruct protected PHP source code and implemented a prototype of a decompiler.
- We evaluate our prototype with 10 protected, real-world applications and show that it is possible to reconstruct the original source code from the protected bytecode.

Last but not least, we would like to raise awareness about the usage of PHP bytecode protectors and their effectiveness on protecting sensitive data. We hope that our research can guide future work on protecting interpreted languages and that it offers new insights into the limitations of obfuscation techniques.

2 Background

In order to analyze PHP source code protectors, we first take a look at several PHP internals. We provide a brief introduction to PHP's interpreter, virtual machine, and instructions. Then, we outline the general concept of PHP source code protectors and introduce the three most popular tools on the market.

2.1 PHP Interpreter

PHP is a platform independent scripting language that is parsed by the PHP interpreter. The PHP interpreter is written in C and can be compiled cross-platform. Unlike low-level languages such as C, no manual compilation into an executable file is performed for PHP code. Instead, an application's code is compiled to *PHP bytecode* on every execution by the Zend engine. The Zend Engine [25] is a core part of PHP and is responsible for the code interpretation.

During the compilation process, a PHP file's code is split into tokens by a tokenizer. The process is initiated by PHP's core function `zend_compile_file()`. After tokenizing the code, the compiler uses the tokens to compile them into bytecode. Similarly, the core function `zend_compile_string()` compiles a string and is used, for example, to run code within `eval()`. As we will see in Section 4, PHP core functions play an important role for the dynamic analysis of bytecode protectors. An overview of the PHP interpreter's structure is given in Figure 1.

After the engine parsed and compiled the PHP code into bytecode, its instructions (*opcodes*) are executed by PHP's virtual machine (VM) that comes with the Zend Engine. It has a *virtual CPU* and its own set of instructions. These instructions are more high level than regular machine code and are not executed by the CPU directly. Instead, the virtual machine provides a handler for each instruction that parses the VM command and runs native CPU code.

The execution process is initiated by passing the *opcode array* to the function `zend_execute()`. It iterates over the opcodes and executes one after another. Calls to user-defined functions are handled recursively and return to the call site's opcode. The function terminates when a *return* opcode in the *main* opcode array is found. In the next section, we look at opcodes in detail.

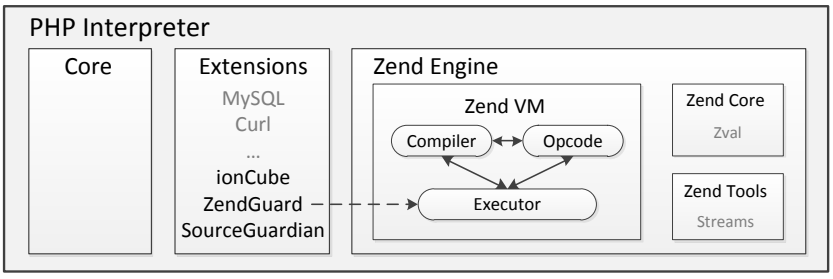

Fig. 1. The PHP interpreter with its core, extensions, and the Zend Engine.

2.2 PHP Bytecode

The register-based bytecode of PHP consists of opcodes, constants, variables, and meta information. PHP has around 150 different opcodes that cover all existing language constructs [26]. Basically, each opcode has an *opcode number* that is used to find the corresponding opcode handler in a lookup table, two *parameter* operands, and a *result* operand to store return values. The parameter operands of an opcode store the values that are processed in the operation. These operands can have five different types and there is a variety of use cases. For example, an operand can be a constant or a variable. Temporary variables are used for auxiliary calculations or results that are not assigned to a variable.

Since there are different operand types, there are multiple instances of opcode handlers as there are 25 possible operand combinations for each instruction. For example, the handler function for adding two variables is different to the one for adding two constants. The overall number of handler functions is less than $150*25$ because some combinations are redundant or invalid. The index to retrieve the handler address from the handler table is calculated using the following formula:

$$index = opcode\_number * 25 + op1\_type * 5 + op2\_type \tag{1}$$

Every operand combination for each opcode is stored within this table and links to the appropriate handler that performs the operation. Invalid combinations terminate the PHP process with a corresponding error message.

Next to the opcodes, the bytecode contains structures. These hold constant values, such as numbers or strings, and variable names which are referenced in operands with a key. Furthermore, meta information, such as line numbers and doc comments, is available as well as a *reserved* variable that allows extensions to store additional information. The bytecode of user-defined functions and methods is stored similarly in opcode arrays. Here, the name and argument information is stored additionally. A global function table links to the corresponding opcode array by function name. Classes have their own method table that links to the methods. When a method or function call is initiated, PHP uses these tables to find the appropriate opcode array and executes it.

In the following, we take a look at a code sample and its bytecode after compilation. The following three lines of PHP code perform a mathematical operation, concatenate the result with a static string, and print the result *RAID2015*.

```
$year = 2000 + 15;
echo "RAID" . $year;
```

The disassembly of the compiled code is shown in Table 1. We have already mapped the opcode numbers to the corresponding handler names as well as variable names to operands. The compilation process converted the script into four operations. First, the `ADD` opcode handler adds the two constants `2000` and `15` and stores the result in the temporary variable `TMP:1`. Second, the `ASSIGN` opcode handler assigns the temporary variable `TMP:1` to the variable `$year`. Third, the `CONCAT` opcode handler concatenates the string `'RAID'` with the variable `$year` and stores the result in the temporary variable `TMP:2`. Fourth, the `ECHO` opcode handler prints the value of the temporary variable `TMP:2`.

Table 1. Exemplary bytecode.

#	Opcode	Operand 1	Operand 2	Result
1	ADD	`2000`	`15`	TMP:1
2	ASSIGN	`$year`	TMP:1	
3	CONCAT	`'RAID'`	`$year`	TMP:2
4	ECHO	TMP:2		

2.3 PHP Bytecode Encoder

The general idea to create a *closed-source* PHP application is to compile a PHP script once and to dump all opcode arrays. This data can then be directly deployed to PHP's executor without another compilation of the source code. Because PHP has no native solution for this, a custom PHP extension can be implemented that dumps the bytecode into a file before it is executed (*encoder).* A second extension (*loader*) then parses the dumpfile and deploys the bytecode to the PHP engine. The process is depicted in Figure 1 with a dashed arrow. As a drawback, PHP version specific extensions have to be provided if the bytecode format changes with different PHP releases.

However, as we have seen in Section 2.2, PHP bytecode is still readable, thus, additional protection mechanisms are reasonable. For example, it is possible to add several encryption layers around the bytecode. Furthermore, the execution of the encrypted bytecode can be limited to a specific user license or hardware environment by the loader extension. While such mechanisms can increase the security (or obscurity), the performance of an application might suffer. In the following, we introduce the three most popular commercial PHP bytecode protection tools. For all three, the loader extension is available for free, while the encoder extension is commercial. All three products promise bytecode protection by offering encryption, environment restriction, prevention of file tampering, as well as symbol name obfuscation (except for SourceGuardian).

ionCube is probably the most popular and most used software that obfuscates PHP scripts since 2003. The vendor describes its product as "the ideal and only serious no-compromise solution for protecting PHP" [9]. A single-user license for the latest version 8.3 costs $ 199.

Zend Guard has been developed by Zend Technologies in order to protect scripts from software pirates. The currently available version 7.0 costs $ 600 annually and is the most expensive solution. The vendor's online shop claims "to prevent unlicensed use and reverse engineering and to safeguard your intellectual property through encryption and obfuscation" [30]. However, during our analysis, no encryption process was identified.

SourceGuardian exists since 2002 [14] and was merged with phpShield in 2006. Both products are similar with the difference that SourceGuardian includes environment restriction features. Encoded files can be compatible with different versions of PHP at once. The product is advertised as "the most advanced PHP Encoder on the market" [22]. The latest version 10.1.3 is available for $ 159.

3 Static Analysis of Loader Extensions

In order to reveal the inner working of the introduced tools, we reverse engineered the corresponding loader extensions. In approximately four weeks, we analyzed the encoders for PHP version 5.4 which was the only common supported PHP version for the encoders at the time of our analysis. As a result, we were able to identify protection mechanisms, algorithms, and security vulnerabilities. Although new versions of the encoders were released in the meantime, no significant changes in the inner working were introduced to the best of our knowledge. In this section, we first provide a brief overview of the encoder similarities and then go into product-specific details. Due to space limitation and ethical considerations (see Section 8), we focus on our key findings.

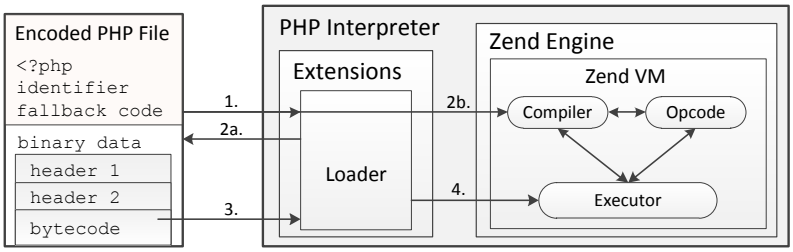

Fig. 2. Workflow of a loader extension: it parses the binary data of a protected file and extracts the PHP bytecode that is then executed.

3.1 Overview

Although all analyzed encoders use different methods for data encoding and encryption, the overall structure of a protected file and its binary data is similar. We depicted the general workflow in Figure 2. First, each protected file identifies itself as encoded (1). Then, native PHP code provides a fallback routine in case no loader extension was found (2b). It informs the user about the missing extension and terminates the execution. If the loader is available, the binary data is parsed (2a) that hides the PHP bytecode in a proprietary binary format. A first unencrypted header is extracted that specifies the PHP and encoder version. The following second header is encoded or encrypted and stores more detailed information regarding license information, expiry time and environment restrictions. Some of these information can be outsourced to an external license file. In this case, the second header also stores information regarding this file's protection. Finally, the PHP bytecode follows in a proprietary format (3). If the license is not expired and the runtime environment matches the restriction rules, the PHP bytecode is executed by passing it to the PHP VM (4). In the following, we introduce product specific details. We reverse engineered each loader extension's process of extracting the bytecode from the binary data step by step. Surprisingly, very little obfuscation is used within the extensions itself. This would at least hinder the reverse engineering process and require more effort, but not prevent it. An overview of the identified core features is given in Table 2.

Table 2. Overview of the loader extensions' internals.

file format	ionCube	Zend Guard	SourceGuardian
	`<?php`	`<?php`	`<?php`
identifier		`@Zend;`	`@"SourceGuardian";`
fallback length	`//004ff` (hex)	`4147;` (octal)	
fallback code	try to load extension		try to load extension
	print message	print message	print message
data	`?>`	`?>`	`sg_load('`binary data`')`
	binary data	binary data	`?>`
fallback check	Adler32	n/a	32bit BSD
data encoding	custom Base64/raw	n/a	Base64
header 1	ionCube	Zend Guard	SourceGuardian
header info	version, PRNG seed	version	version, protection flags
header 2	ionCube	Zend Guard	SourceGuardian
encryption	XOR with PRNG	n/a	Blowfish CBC mode
encoding		custom GZIP	Lempel Ziv (lzo1x)
checksum	MD4, custom Adler32	n/a	32bit BSD
header info	license info	license info	license info
	restriction rules	restriction rules	restriction rules
	fallback checksum	license checksum	fallback checksum
bytecode	ionCube	Zend Guard	SourceGuardian
encryption	XOR with PRNG	n/a	Blowfish CBC mode
encoding	GZIP	custom GZIP	Lempel Ziv (lzo1x)
checksum	custom Adler32	n/a	32bit BSD
obfuscation	forged opcode nr	forged opcode nr	n/a
license file	XOR, Blowfish	DSA signature	Blowfish encryption

3.2 IonCube

The ionCube loader is compiled with multiple optimizations and without debug symbols. All internal function names are obfuscated. Internal strings, such as error messages, are XOR'ed with a static 16 bytes key, while a prefixed character in the string denotes the offset for the XOR operation to start. Other protection mechanisms against reverse engineering are not present.

The loader hooks `zend_compile_file()` and tests for the string `<?php //` at the beginning of an executed PHP file. The then following hexadecimal number denotes the size of the native PHP fallback code. If the ionCube extension is loaded, the fallback code is skipped and the loader begins to parse the binary data at the end of the file.

IonCube ships the binary data either raw or encoded with a custom base64 encoding (default). The base64 character set is slightly modified so that numbers occur first in the set. Once the data is decoded, a binary stream is used to parse it. The first four byte specify the ionCube version used for encoding. Because

ionCube's format changed over time, different parsing routines are used. For example, `0x4FF571B7` is the identifier for version 8.3. The version identifier is followed by three integers: the file size, header size, and a header key. This key is used to decrypt the header size and file size, using the following calculation:

```
header_size = header_key ^ ((header_size ^ 0x184FF593) - 0xC21672E)
file_size = (file_size ^ header_key ^ 0x23958cde) - 12321
```

While the `file_size` is used for debugging, the `header_size` indicates the length of the second header. The second header is XOR'ed with a pseudo random byte stream. For this purpose, a *SuperKISS*-like PRNG is seeded with the `header_key` from the first header. The last 16 bytes of the encrypted second header represent the MD4 checksum of the decrypted second header's data. An Adler32 checksum follows that is used to verify the integrity of the first and second header, as well as its MD4 checksum. Within the modified Adler32 algorithm, the sum of all bytes is initialized to `17` instead of `1`. The MD4 checksum is unscrambled first using the following pseudo algorithm.

```
md4_sum = substr(header, -16);
for(i=0; i<16; i++) {
    md4_sum[i] = ((md4_sum[i] >> 5) | (md4_sum[i] << 3));
}
```

Here, every byte of the checksum is rotated by 5 bits. Then, the PRNG is initialized and pseudo-random bytes and parts of the MD4 checksum are used as decryption key. The following pseudo algorithm is used to decrypt the second header byte-wise.

```
prng = new ion_prng(header_key);
for(i=0; i < header_size - 16; i++) {
    raw_header[i] ^= md4_sum[i & 15] ^ (prng->next() & 0xFF);
}
```

At the end, the integrity of the header's data is verified by calculating and comparing its MD4 checksum. The second header contains the configuration values of the ionCube protection. For example, a version number allows the loader to determine if the file's PHP version matches the system's PHP version and to find the corresponding decoding routine. A checksum of the native fallback code allows to validate its integrity. Furthermore, licensing information and environment restriction rules are found. For optional name obfuscation, a salt is found that is used to hash names of variables, functions, and classes with MD4. If the restriction rules are outsourced to a license file, the file path and decryption key is provided.

After the second header and its checksum, two integers and the encrypted PHP data follow. The first integer seeds a PRNG in order to generate a byte sequence that is the XOR key for the encrypted PHP data. After successful decryption, the data is decompressed using GZIP. At this point, the opcode

numbers within the bytecode are still encrypted. A new PRNG is seeded with the second integer. In combination with a static value from the second header, a new byte sequence is generated in order to decrypt the opcode numbers and to perform a runtime obfuscation. We explain this process in Section 4.2 in detail.

In contrast to Zend Guard and SourceGuardian, ionCube does not send the bytecode to PHP's native execution function in order to run the code. Instead, slightly modified parts of the native PHP interpreter are compiled into the loader extension and are used as a proprietary VM with further obfuscation.

3.3 Zend Guard

Zend Guard's loader extension is not protected against reverse engineering, except for the obfuscation of its verbose error messages. These are XOR'ed using the four bytes `\xF8\x43\x69\x2E`. Moreover, Zend Guard leaks its compile information which helps to exclude library code from reverse engineering, while the latest loader version even includes debug symbols.

In order to detected encoded files, Zend Guard replaces PHP VM's function `zend_compile_file()`. If the string `<?php @Zend;` is not found at the beginning of the file, it will be passed back to the original compile function. Otherwise, the file is processed by Zend Guard. It reads the octal number in the second line and skips the amount of bytes of the fallback routine in order to reach the raw binary data at the end of the file.

To retrieve the data, a parser iterates over the byte stream. The data blocks are stored using a simple binary format which basically consists of four different data types: bytes, bools, numbers, and strings. When a single byte or bool is requested from the stream, the parser reads one character. Numeric values consist of a single byte which defines the length of the number followed by the actual integer (e. g., `[\x05][2015\x00]`). Strings extend the integer type with a byte sequence and its length (e. g., `[\x02][5\x00][RAID\x00]`). Both, numbers and strings, are terminated with a null byte.

The binary data consists of two parts: a minimalistic header followed by compressed data. There are four values stored within the first header. The first value indicates the version of Zend Guard and the second value identifies the PHP version the script was compiled for. Then, two numeric values specify the size of the compressed and the uncompressed data. The then following data is compressed using *GZIP* and a custom compression dictionary. This dictionary lists words that occur frequently in PHP code in order to improve the compression rate. It is required to decompress the data and stored within the Zend Guard loader. The decompressed data contains a second header and the PHP bytecode.

In the second header, license information, such as the license owner's name, and configuration flags, such as if the license is attached as a file, is stored. It also provides an expiration timestamp. If the current time is ahead of the timestamp, or if the license is invalid, Zend Guard denies executing the file. The license is validated by calculating an Adler32 checksum of the owner's name and comparing it to a checksum in the header. For this purpose, a slightly modified Adler32 algorithm is used that lacks the modulo operation.

Once the second header is completely decoded and verified, the compiled PHP data is parsed. For this purpose, Zend Guard uses the previously introduced stream format to parse the data. Opcodes, literals, and variables are the main part of this structure but also meta information, such as line numbers or comments, is available if it was not explicitly disabled during encoding. Moreover, Zend Guard scrambles the opcode numbers. For recovery, a substitution table is created using constant values from two arrays and the license owner checksum from the second header. With the help of the substitution table and the current opcode index, the original opcode number can be calculated.

Furthermore, Zend Guard is able to obfuscate function and variable names. Here, the lowercased name is XOR'ed with the MD5 hash of the name, such that the original name is lost. However, the name can be brute-forced with a dictionary. Also, the key space is reduced because the original name is lowercased first and the obfuscated name has the same length as the original one.

3.4 SourceGuardian

The process of reverse engineering SourceGuardian is comforted by available debug symbols within its loader extension. Almost all function and symbol names are intact and ease the understanding of internal processes. A PHP file's protection is indicated in the first line with the identifier `@"SourceGuardian"` (or `@"phpSHIELD"` for older versions). Instead of hooking the `zend_compile_file()` function, SourceGuardian adds a new native function `sg_load()` to the PHP core which parses a proprietary binary format.

```
<?php @"SourceGuardian";
if(!function_exists('sg_load')){ // fallback code
} return sg_load('12345678CHECKSUM/BASE64/BASE64/BASE64/BASE64=');
```

The argument of `sg_load()` is a concatenation of 16 hexadecimal characters and *base64* encoded binary data. First, a checksum is calculated over the characters ranging from the opening PHP tag `<?php` to the 8th character of the `sg_load()` argument. This checksum is then compared to the next eight characters of the argument in order to verify the fallback code's integrity. The checksum algorithm appears to be a 32bit version of the BSD checksum.

The *base64* encoded binary data is decoded and reveals a binary format with four data types. The type `char` is used to represent single characters, small numbers, and boolean values. Integers are stored using four bytes in little endian format (type `int`) and strings can either be zero terminated (type `zstr`) or have a prefixed length (type `lstr`).

At the beginning, a first header is parsed that contains a version number and protection settings, in case the file is locked to a specific IP address or hostname. The first byte of a data block in the first header decides upon the purpose of the upcoming bytes, until a `0xFF` byte is found. For example, the value *0x2* indicates that execution is restricted to a hostname and the value *0x4* indicates that the length of the second header follows.

Once the offset of the encrypted second header is obtained from the first header, it is decrypted using the block cipher *Blowfish* in CBC mode. For this purpose, SourceGuardian's loader comes with multiple static keys that belong to three different groups. Depending on the file version, it iterates over the appropriate group's keys until the decryption succeeds (checksum matches). Multiple keys exist because of backwards compatibility and the *phpShield* aggregation:

```
NTdkNGQ1ZGQxNWY0ZjZhMjc5OGVmNjhiOGMzMjQ5YWY= // public key
MmU1NDRkMGYyNDc1Y2Y0MjU5OTlmZDExNDYwMzcwZDk= // public key
NzkxNThhZDhkOThjYTk3ZDE5NzY4OTRkYzZkYzM3MzU= // license file key
ODI0YzI2YmMyODQ2MWE4MDY3YjgzODQ2YjNjZWJiMzY= // phpShield pub key
YTJmNjc2MDQ3MWU5YzAxMjkxNTkxZGEzMzk2ZWI1ZTE= // phpShield pub key
```

In case the execution of the protected file is restricted to a server's IP address or hostname, this value is appended to the decryption key of the second header and body. Hence, the loader will not be able to decrypt the binary block in other environments and the execution fails. By default, an attacker can perform decryption by using the static keys. We believe that the additional key data (IP or hostname) does not add any further security because the origin of a stolen source code file is most likely known to the attacker or can be brute-forced.

Each successfully decrypted block contains three integers and the actual data. The first integer is a checksum calculated over the plain data. The second integer contains the length of the unencrypted data and the third integer is the size of the data after decompression. The checksums are calculated with the previously mentioned 32bit BSD checksum. If the first integer matches the calculated checksum, the decryption was successful.

At this point the data is decrypted, but still compressed with the Lempel Ziv algorithm. SourceGuardian uses the *lzo1x* implementation and *lzss* for files encoded with an older version of SourceGuardian. The second header and the PHP data blocks are compressed and encrypted using this technique.

Similar to the first header, a parser iterates over the data and retrieves the values of the second header. It contains information about the environment restrictions, such as the license owner, license number, file creation, and file expiration date. After the second header, the PHP data follows. SourceGuardian is able to store multiple versions of it for compatibility with different PHP versions. One data block is used for each version. Each block consists of two integers that note the compatible PHP version and the size of the encrypted data, as well as the actual PHP data. If a compatible data block for the currently running PHP version is found, the block is decrypted. No further obfuscation, such as of variable names, is performed and the deobfuscated opcode array is passed to `zend_execute()`.

3.5 Security Vulnerabilities in Loader Extensions

Protecting a PHP application can prevent intellectual property theft and modification when shipped to a customer. At the same time, however, it prohibits

that the customer can review the code before it is deployed and run on his server. In order to mitigate risks, PHP mechanisms such as `safe_mode` and `disable_functions` can be activated that can forbid OS interaction, such as executing system commands, when running unknown protected PHP code.

During the process of reverse engineering, we detected memory corruption vulnerabilities in each of the loader extension. By crafting a malicious PHP file, it is possible to corrupt the loader's parser and to inject shellcode. While these vulnerabilities are not remotely exploitable, they allow a protected application to bypass PHP's security mechanisms and to execute arbitrary system commands with user privileges of the web server. We informed ionCube, Zend Guard, and SourceGuardian about these issues.

Furthermore, we detected an undocumented feature in SourceGuardian which allows to leak the license information. By sending the HTTP GET parameter `__sginfo__` to a protected application, it responds with the encoder version, registration date, license owner, and date of encoding.

4 Generic Deobfuscation via Dynamic Analysis

We now introduce two *dynamic* approaches to analyze protected PHP applications. Our goal is to retrieve information about the original code by circumventing deployed encryption or obfuscation layers at runtime. We tested both approaches against ionCube, Zend Guard, and SourceGuardian and found all tools to be vulnerable against both attacks.

4.1 Debugging

A straight-forward approach to analyze protected PHP code is to include it into the context of own code that uses PHP's built-in debug functions to leak information about the current runtime environment. For example, the functions `get_defined_vars()`, `get_defined_functions()`, `get_declared_classes()`, and `get_class_methods()`, as well as PHP's built-in `ReflectionClass` allow to retrieve a list of all variables, user-defined functions, classes, and methods. Once obtained, variables can be dumped and functions can be called as a blackbox with different input in order to obtain further information. All three tested tools have an option to prevent the inclusion of compiled code within an untrusted context to prevent this analysis, but this is disabled by default.

4.2 Hooking

A more sophisticated approach is to hook [10] internal PHP functions in order to retrieve the complete bytecode before it is executed. As explained in Section 3, Zend Guard and ionCube replaces the `zend_compile_file()` function. It returns the decoded and decrypted bytecode of a given file. We can use these functions as a black box in order to retrieve the deobfuscated opcode arrays without knowledge of the loaders' inner working. Bytecode from SourceGuardian files cannot be obtained this way because it does not replace `zend_compile_file()`.

However, every product passes the deobfuscated opcode arrays as an argument to `zend_execute()` (see Figure 1, dashed arrow). By hooking this function, we can interrupt the execution and obtain the main opcode array. Opcode arrays of methods and functions can be located with the help of PHP's internal `compiler_globals` structure. This way, the raw PHP bytecode of applications protected with SourceGuardian can be retrieved directly. For ionCube and Zend Guard, further obfuscation has to be removed (see Section 3.2 and 3.3).

ionCube To avoid opcode dumping, ionCube implements a runtime obfuscation that XOR's single opcodes before execution and XOR's them again afterwards. This ensures that only one opcode is deobfuscated at a time. Furthermore, ionCube contains a copy of the native PHP engine and bytecode is processed within the loader instead of the PHP VM. Consequently, the last step in Figure 2 is omitted and ionCube's internal `zend_execute()` function needs to be hooked for dynamic analysis.

The executed instructions are obfuscated with two techniques. First, the opcode number, the handler address, and the operands of all opcodes are encrypted with XOR. Second, numeric operands of assignments are obfuscated by mathematical operations with constants. The *reserved* variable of the opcode array references to an ionCube structure which contains the keys for opcode decryption (see Section 3.2) and assignment deobfuscation. Each opcode is XOR'ed with a different key which is referenced by the opcode index. Then, the opcode is executed and obfuscated again using the same XOR operation. By retrieving all keys from the ionCube structure, we are able to deobfuscate all opcodes.

Zend Guard When a PHP file is parsed, the opcode number is used by the PHP interpreter to resolve the handler's address. As noted in Section 3.3, Zend Guard removes the opcode number before passing the bytecode to `zend_execute()`. In order to recover the opcode number again, we can search the present handler address in the opcode lookup table. This is achieved by calculating the index of all existent opcodes (see Formula 1 in Section 2.2) and comparing it to the index of the current address.

5 Decompiler for Obfuscated PHP Bytecode

Using the insights introduced in Section 3 and the techniques presented in Section 4.2, we implemented a decompiler. For this purpose, we set up a PHP environment with the three loader extensions as well as a custom PHP extension. The decompilation is performed in three steps. First, we hook PHP's executor in order to access the bytecode. Second, we remove all remaining obfuscation and dump the bytecode to a file. Third, the dumped bytecode is decompiled into PHP syntax. It is also possible to statically recover the PHP bytecode from the protected PHP file without execution by using the insights presented in Section 3. However, the implementation of a version-specific parser for each loader extension is required, while the dynamic approach can be applied generically.

5.1 Hooking

Our PHP extension hooks the `zend_execute()` function by replacing it with our own implementation. Then, we execute each file of a protected PHP application in our PHP environment. As explained in Section 3, the corresponding loader extension now hooks `zend_compile_file()` and extracts the PHP bytecode from the proprietary binary format. When the bytecode is passed to `zend_execute()`, our extension terminates the execution. Because SourceGuardian does not hook `zend_compile_file()` and implements the native PHP function `sg_load()`, we here intercept only the second invocation of `zend_execute()`. This way, we allow the initial execution of `sg_load()` that performs the initial decryption and deobfuscation of the bytecode, before it is passed to `zend_execute()` again.

5.2 Dumping

For ionCube and Zend Guard, we perform further bytecode deobfuscation as described in Section 4.2. Then, the bytecode is free of any encoder-specific modifications. Each opcode array contains several data structures which are referred by the operands (see also Section 2.2). Operands of type `VAR` and `CV` refer to variables with a name which is stored within the `vars` structure. Constants are used by operands of type `CONST` and can be found within the `literals` structure. Opcodes themselves are stored in the `opcodes` structure. If the opcode array represents a function, the parameters are available in the structure `arg_info`. We begin dumping the main opcode array and continue with user defined functions. Classes are stored in an own structure that basically contain information about member variables and the method table. After dumping the bytecode into a file, it can be deployed to our decompiler for further processing.

5.3 Decompilation

Next, each instruction is inspected and transformed into the corresponding source code representation. The opcodes can be grouped into one of three different types of instructions:

1. **Expressions** are instructions which produce temporary values that are used as operands by other instructions, for example, a mathematical operation.
2. **Statements** are instructions that cannot be used as an expression and do not have a return value, for example an `echo` or `break` statement.
3. **Jumps** are special cases of statements. They defer the execution by a jump and represent a loop or conditional code.

In general, the best way of decompiling bytecode back into source code is to create a graph by connecting separated basic blocks such that each part of the code can be converted separately [1, 2, 12]. However, this approach is out of scope for this paper. For our proof of concept, we follow a simpler approach: our decompiler is based on a pattern recognition approach that finds jump and loop structures. Empirically we found that this approach is already sufficient to recover most PHP source code.

Table 3. Exemplary bytecode with decompiled syntax.

#	Opcode	Operand1	Operand2	Result	Code
1	ADD	222	333	TMP:1	
2	ASSIGN	$test	TMP:1		$test = 222 + 333;
3	IS_GREATER	$test	500	TMP:2	
4	JMPZ	TMP:2	JMP:7		if ($test>500) {
5	ECHO	$test			echo $test;
6	JMP	JMP:7			}
7	RETURN	1			return 1;

Our approach consists of two steps. First, we iterate over all opcodes in order to reconstruct expressions and statements. During this process, ternary operators and arrays are rebuilt and coherent conditions are merged. Afterwards, we remain with PHP source code and jump instructions. Finally, we try to find patterns of commonly used jump and loop structures in order to reassemble the control flow.

The code in Table 3 provides an example of PHP bytecode. Here, we first buffer the PHP syntax of the `ADD` expression stored in `TMP:1` (`op1+op2`). Next, the first line of code is recovered by resolving the operand `TMP:1` in the assignment of variable `$test`. Further, we construct the *greater-than* constraint created from the variable `$test` and the constant value `500` (`op1>op2`). Then, the operand `TMP:2` can be resolved in line 4. In the next line, we create the `echo` statement. We ignore the `JMP` for now and finish with the transformation of the return statement. When all expressions and statements are processed, we begin with finding patterns by processing the jump operands. In our example, we recognize the `JMPZ` in line 4 that jumps to the same location as the following `JMP` in line 6 as an `if` construct.

Similarly, we can recognize more complex `if`/`else` constructs. As shown previously, a single `if` block without an `else` branch is identified by a conditional jump instruction that skips upcoming statements in case the condition fails. Unoptimized bytecode has a `JMP` instruction inside the `if` block that jumps to the next instruction after the `if` block. In this particular case, the second jump is unnecessary for execution but helps to recognize the pattern. If this `JMP` instruction would skip upcoming statements instead, these statements would be assigned to an `elseif`/`else` block.

In PHP bytecode, `for` loops have an unique pattern. The overall layout comprises a loop constraint, a `JMPZNZ`, an iteration expression, a `JMP`, followed by the loop body and a final `JMP`. The `JMPZNZ` operation has two jump locations stored in its operands. The first jump is taken in case of a zero value, and the second one otherwise. The second location points behind the loop body. The interpreter jumps to this location when the condition of the `JMPZNZ` instruction does not match. The bytecode at the first location represents the start of the loop body. The `JMP` instruction at the body's end jumps back to the loop's constraint.

Similarly, `while` loops can be detected. Here, a constraint is followed by a `JMPZ` instruction that points behind the loop's body. Then, the loop's body follows which ends with a `JMP` instruction that points back to the loop's constraint.

More convinient is the recognition of `foreach` loops. Here, the rare opcode `FE_RESET` is used to reset an array's pointer and then a `FE_FETCH` opcode follows to fetch the current array's element. Then, the loop body follows that ends with a `JMP` instruction. The initial `FE_` opcodes both have a jump location stored in their second operand. This location points behind the last `JMP` instruction in the loop's body and it is accessed when the loop is finished. The `JMP` instruction itself points back to the `FE_FETCH` opcode.

In order to resolve nested constructs, our algorithm uses an inside out approach in several iterations. We mark sustained patterns as resolved and repeat our pattern matching algorithm until no new patterns are detected. This way, in a nested construct, the most inner pattern is resolved first, followed by the identification of the outer pattern in the next iteration.

Our pattern matching approach works very well on unoptimized bytecode since PHP adds redundant opcodes that ease the recognition process. Unfortunately, these patterns can change when bytecode optimization is enabled. Here, redundant operations are removed, structures are compressed, and targets of jump operations are pre-resolved. This makes it significantly harder to find and decompile structures. To overcome such limitations, a more elaborated decompiler design could be implemented in the future [2].

Parts of our approach for reconstructing expressions and statements into source code could be adopted for other register-based virtual machines. While simple opcodes, such as for addition or concatenation, can be compared to other languages, complex opcodes, such as for the access of arrays, are very PHP specific. For stack-based bytecode, as used in Java, Python, or Perl, the operands have to be resolved from the stack first. Our pattern matching approach for the evaluation of code structures bases on artifacts found in PHP bytecode and thus is not directly applicable to other languages.

6 Evaluation

We evaluate our decompiler in two steps. First, we try to quantify our decompilation results by encoding a set of known source code and comparing the decompiled code to the original version. Then, we test our decompiler against 10 protected real-world applications and try to recover unknown source code.

6.1 Source Code Reconstruction

Measuring the quality of decompiled PHP code is hard and, to the best of our knowledge, no code similarity algorithm for PHP exists. While the code's semantic remains after decompilation, the syntax changes due to PHP's native and the encoders' additional bytecode optimization. Due to limitations of our proof of concept implementation (see Section 5.3), our prototype does not always

produce syntactically correct code and a comparison of successful unit tests of a decompiled application is not applicable. Hence, we developed a basic metric based on PHP tokens [23]. We categorized all tokens into one of seven groups:

1. **DATA**: tokens of literals, constants, and variables (`T_VARIABLE`)
2. **EQUAL**: tokens of assignment operators, such as `T_PLUS_EQUAL`
3. **COMP**: tokens of comparison operators, such as `T_EQUAL` and `T_ISSET`
4. **CAST**: tokens of type casts, such as `T_INT_CAST` and `T_STRING_CAST`
5. **INCL**: tokens of include statements, such as `T_INCLUDE` and `T_REQUIRE`
6. **PROC**: tokens of procedural code, such as `T_FUNCTION` and `T_NEW`
7. **FLOW**: tokens of jump and loop statements, such as `T_IF` and `T_WHILE`

Tokens that do not fall into one of these categories were ignored. We also ignored encapsulated variables and constants, comments, whitespaces, logical operators, and inline HTML. Next, we compiled the three most popular PHP projects *Wordpress*, *Drupal*, and *Joomla* with the most complex encoder ionCube with default optimization level. Then, we used our prototype for decompiling the protected code again. We used PHP's built-in tokenizer [24] to collect the number of tokens in all PHP files of the original and the recovered source code and calculated the individual success rate for each token. In Table 4, we list the average similarity of each token category that was weighted by token popularity in each group. We observed a very similar amount for tokens that are not part of optimization. As expected, the number of tokens for optimized instructions or loops (FLOW) vary more significantly. Based on our results, we estimate a successful reconstruction rate of about 96%.

Table 4. Average token similarity (in %) for three compiled/decompiled applications.

Software	Version	EQUAL	DATA	COMP	CAST	INCL	FLOW	PROC
Wordpress	4.2.2	95.83	95.13	98.52	99.77	99.85	84.17	96.83
Joomla	3.4.1	96.45	95.33	99.76	99.53	99.77	82.33	97.36
Drupal	7.37	98.81	92.81	98.64	98.45	98.78	89.00	98.34

6.2 Protected Real-world Applications

In order to inspect protected code in real-world applications, we selected 10 popular encoded PHP applications. Our corpus is presented in Table 5. The number of evaluated software per encoder was chosen by the encoder's popularity. In total, we were able to recover more than 1 million lines of actual code (RELOC) in 3 942 protected PHP files. Bytecode optimization was enabled for some of the applications which led to errors when decoding optimized structures. These errors are very specific to the optimized code and cannot be generalized. Here, our prototype implementation requires improvement for a more precise reconstruction. However, errors in code nesting, such as the misplacement of curly braces, does not hinder to fully understand the recovered source code and to

retrieve sensitive information, such as cryptographic keys, or to detect security vulnerabilities. In the following, we present our findings. Note that due to the large corpus, only a fraction of code could be analyzed.

Table 5. Corpus of selected real-world applications that apply an encoder.

Software	Version	Category	Encoder	Files		RELOC
				Total	Protected	
WHMCS	5.3.13	hosting	ionCube	946	688	157 651
HelpSpot	3.2.12	helpdesk	ionCube	493	163	41 033
xt:Commerce	4.1	webshop	ionCube	4 090	118	35 864
PHP-Cart	4.11.4	webshop	ionCube	271	3	2 762
Precurio	4	intranet	ionCube	2 985	5	579
XT-CMS	1.8.1	CMS	SourceGuardian	320	87	21 653
Mailboarder	4.1.5	email	SourceGuardian	110	110	16 365
NagiosFusion	2014R1.0	monitoring	SourceGuardian	294	15	3 337
Scriptcase	8.0.047	development	Zend Guard	3 751	2 676	726 552
gSales	rev1092	billing	Zend Guard	206	77	34 012

License Systems In all 10 analyzed applications, the protection is primarily used to hide a license system. It can limit the application's use to a specific time (7), number of users (5), domain or MAC address (4), software version (3), or restrict software features of a demo version (5). By decompiling the protected sources, we can reveal the keys and algorithms used. For example, we could recover the static secret in PHP-Cart (MD5 salt), HelpSpot (RC4), gSales (SHA1 salt) and Mailborder (AES 128bit) that is used to validate or decrypt the license data. In NagiosFusion, we discovered a custom decoding algorithm that is used to infer the installation's restrictions from the license key. The decompilation of these sensitive sources does not only allow to fake a valid runtime environment and license, but also to remove these checks completely.

Vulnerabilities Furthermore, we detected critical security vulnerabilities in the decompiled source codes which could be confirmed against the original protected applications. For example, we detected multiple *path traversal* vulnerabilities in HelpSpot and scriptcase which allow to remotely retrieve any file from the server, and multiple *SQL injection* vulnerabilities in HelpSpot, xt:Commerce, and gSales which allow to extract sensitive data from the database. We believe that these vulnerabilities remained previously undetected for the reason of unavailable source code. It is controversial whether this is more helpful for the vendor or the attackers [18, 28]. Arguably, some vulnerabilities could be also detected without the source code. However, some vulnerabilties are hard to exploit in a blackbox scenario, for example, a detected *second-order file inclusion* vulnerability [5] in Mailborder or *PHP object injection* vulnerabilities [6] in xt:Commerce, PHP-Cart, and HelpSpot. Clearly, a vendor should not rely on source code protectors to assume security issues remain undetected. We reported all identified issues responsibly to the corresponding vendor.

Pingbacks and Backdoors Next to vulnerabilities, we looked for suspicious functionalities of the protected applications. We found rather harmless pingback features, for example in xt:Commerce, that send information about the installation environment and license to a SOAP-based web service. While this can be used to check for updates, it is also a good way to observe active installations. More severe is that xt:Commerce also sends the user's PayPal API credentials in plaintext to its server via HTTP. Precurio collects information about the application's server and owner and sends it via CURL request to the Precurio website, in case the ionCube license does not match to the server or is expired.

However, we also detected an odd vulnerability in Precurio. The following three lines of code determine if the request path is a file and in that case output its content. Thus, by requesting for example the URL `/index.php/index.php` from the server, the PHP source code of the index file is leaked.

```
$filename = $root . '/public/' . $_SERVER['PATH_INFO'];
if ( is_file($filename) )
    echo file_get_contents($filename);
```

Moreover, the code allows to retrieve any file from Precurio's web directory, including user files and the license file. Additionally, we found that the `ErrorController` in Precurio implements a `downloadAction`. Thus, the URL `/error/download` allows the Precurio team, as well as any other remote user, to download the log file of the Precurio installation which leaks detailed stack traces and exceptions. We informed Precurio about both issues.

7 Related Work

Obfuscation of software systems is used in practice to increase the costs of reverse engineering, for example in the context of digital rights management systems [27] or IP protection. As a result, many different types of obfuscation techniques were developed in the past years and most of them focus on binary executables (e.g., [3,11,15,21]). So called *executable packers* implement different obfuscation and encryption strategies to protect a given binary. Note that obfuscation is also commonly used by adversaries to hamper analysis of malicious software samples. To counter such tactics, several methods for automated deobfuscation were developed [4,16,20,29], and we observe an ongoing arms race.

Similar obfuscation strategies are used to protect PHP source code and commercial tools are available to implement such strategies. To the best of our knowledge, there is no academic work on PHP obfuscation. Our work is most closely related to a talk by Esser [8], who provided an overview of PHP source code encryption and ideas on how source code could be recovered. Saher presented in a talk some reverse engineering details for ionCube [17]. We reverse engineered the three most popular PHP code protection products and provide detailed information about their internals, together with a decompiler approach.

Static and dynamic code analysis techniques can detect security vulnerabilities and are an important research topic. We complement this field by demonstrating how to access protected PHP code such that an analysis can be performed. In other areas, obfuscation/protection mechanisms have been broken by reverse engineering and binary instrumentation techniques (e.g., [27]) and we show that such attacks are also viable against PHP obfuscation tools.

8 Conclusion

In this paper, we evaluated and documented the level of protection provided by current IP protection tools available for PHP source code. We studied the internals of the three most popular encoder and demonstrated an attack against a shared weakness by a proof-of-concept implementation. As a result, we showed that our decompiler is able to recover 96% of the protected PHP code which would enable an attacker to crack license systems and identify previously unknown vulnerabilities and backdoors. Therefore, we argue that currently available encoder products are no appropriate solutions for intellectual property protection and more elaborated obfuscation approaches are necessary to better protect PHP source code.

Ethical Considerations: We would like to clarify that our work was not motivated by the intention to perform illegal activities, such as copyright violation or server compromise, nor to ease these activities for others. For this reason, we do not publish our decompilation tool and we reported all detected vulnerabilities responsibly to the vendors. Moreover, we presented only key insights of the analyzed products and specific details are intentionally left out, while presented keys and constants are likely subject of change. Thus, we feel that an attacker is still left with a high reverse engineering effort in order to reproduce our attacks for the latest encoders. Rather, we hope that our research helps in building better encoders that do not suffer from our attacks and to advance the state-of-the-art.

References

1. D. Brumley, J. Lee, E. J. Schwartz, and M. Woo. A Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *USENIX Security Symposium*, 2013.
2. C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.
3. C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical report, University of Auckland, New Zealand, 1997.
4. K. Coogan, G. Lu, and S. Debray. Deobfuscation of Virtualization-obfuscated Software: a Semantics-based Approach. In *ACM Conference on Computer and Communications Security (CCS)*, pages 275–284, 2011.
5. J. Dahse and T. Holz. Static Detection of Second-Order Vulnerabilities in Web Applications. In *USENIX Security Symposium*, 2014.
6. J. Dahse, N. Krein, and T. Holz. Code Reuse Attacks in PHP: Automated POP Chain Generation. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.

7. M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A Survey on Automated Dynamic Malware Analysis Techniques and Tools. *ACM Comput. Surv.*, 44(2), Mar. 2008.
8. S. Esser. Vulnerability Discovery in Closed Source / Bytecode Encrypted PHP Applications. Power Of Community, 2008.
9. ionCube Ltd. ionCube PHP Encoder. `https://www.ioncube.com/php_encoder.php?page=features`, as of May 2015.
10. I. Ivanov. API Hooking Revealed. The Code Project, 2002.
11. C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *ACM Conference on Computer and Communications Security (CCS)*, 2003.
12. J. Miecznikowski and L. Hendren. Decompiling Java Bytecode: Problems, Traps and Pitfalls. In *International Conference on Compiler Construction*, 2002.
13. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.
14. phpSHIELD. About phpSHIELD. PHP Encoder by SourceGuardian. `https://www.phpshield.com/about.html`, as of May 2015.
15. I. V. Popov, S. K. Debray, and G. R. Andrews. Binary Obfuscation Using Signals. In *USENIX Security Symposium*, 2007.
16. P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Annual Computer Security Applications Conference (ACSAC)*, 2006.
17. M. Saher. Stealing From Thieves: Breaking IonCube VM to RE Exploit Kits. BlackHat Abu Dhabi, 2012.
18. G. Schryen and R. Kadura. Open Source vs. Closed Source Software: Towards Measuring Security. In *ACM Symposium On Applied Computing (SAC)*, 2009.
19. 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 *IEEE Symposium on Security and Privacy (S&P)*, 2010.
20. M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic Reverse Engineering of Malware Emulators. In *IEEE Symposium on Security and Privacy (S&P)*, 2009.
21. M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Symposium on Network and Distributed System Security (NDSS)*, 2008.
22. SourceGuardian Ltd. PHP Encoder Features. `https://www.sourceguardian.com/protect_php_features.html`, as of May 2015.
23. The PHP Group. List of Parser Tokens. `http://php.net/tokens`, as of May 2015.
24. The PHP Group. Tokenizer. `http://php.net/tokenizer`, as of May 2015.
25. The PHP Group. Zend API: Hacking the Core of PHP. `http://php.net/manual/en/internals2.ze1.zendapi.php`, as of May 2015.
26. The PHP Group. Zend Engine 2 Opcodes. `http://php.net/manual/internals2.opcodes.php`, as of May 2015.
27. R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Steal This Movie: Automatically Bypassing DRM Protection in Streaming Media Services. In *USENIX Security Symposium*, 2013.
28. B. Witten, C. Landwehr, and M. Caloyannides. Does open source improve system security? *Software, IEEE*, 18(5):57–61, 2001.
29. B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.
30. Zend Technologies Ltd. PHP Obfuscator, PHP Encoder & PHP Encryption from Zend Guard. `http://www.zend.com/products/guard`, as of May 2015.