I Forgot Your Password: Randomness Attacks Against PHP Applications

George Argyros[*] Aggelos Kiayias[*†]

Abstract

We provide a number of practical techniques and algorithms for exploiting randomness vulnerabilities in PHP applications.We focus on the predictability of password reset tokens and demonstrate how an attacker can take over user accounts in a web application via predicting or algorithmically derandomizing the PHP core randomness generators. While our techniques are designed for the PHP language, the principles behind our techniques and our algorithms are independent of PHP and can readily apply to any system that utilizes weak randomness generators or low entropy sources. Our results include: algorithms that reduce the entropy of time variables, identifying and exploiting vulnerabilities of the PHP system that enable the recovery or reconstruction of PRNG seeds, an experimental analysis of the Håstad-Shamir framework for breaking truncated linear variables, an optimized online Gaussian solver for large sparse linear systems, and an algorithm for recovering the state of the Mersenne twister generator from any level of truncation. We demonstrate the gravity of our attacks via a number of case studies. Specifically, we show that a number of current widely used web applications can be broken using our techniques including Mediawiki, Joomla, Gallery, osCommerce and others.

1 Introduction

Modern web applications employ a number of ways for generating randomness, a feature which is critical for their security. From session identifiers and password reset tokens, to random filenames and password salts, almost every web application is relying on the unpredictability of these values for ensuring secure operation. However, usually programmers fail to understand the importance of using cryptographically secure pseudorandom number generators (PRNG) something that opens the potential for attacks. Even worse, the same trend holds for whole programming languages; PHP for example lacks a built-in cryptographically secure PRNG in its core and until recently, version 5.3, it tottaly lacked a cryptographically secure randomness generation function.

This left PHP programmers with two options: They will either implement their own PRNG from scratch or they will employ whatever functions are offered by the API in a "homebrew" and ad-hoc fashion. In addition, backwards compatibility and other issues (cf. section 2), often push the developers away even from the newly added randomness functions, making their use very limited. As we will demonstrate and heavily exploit in this work, this approach does not produce secure web applications.

Observe that using a low entropy source or a cryptographically weak PRNG to produce randomness does not necessarily imply that an attack is feasible against a system. Indeed, so far there have been a very limited number of published attacks based on the insecure usage of PRNG functions in PHP, while popular exploit databases[1] contain nearly zero exploits for

[*] Department of Informatics and Telecommunications, University of Athens, argyros.george@gmail.com, aggelos@di.uoa.gr. Research partly supported by ERC Project CODAMODA.

[†] Computer Science and Engineering, University of Connecticut, Storrs, USA

[1] e.g. http://www.exploit-db.com

such vulnerabilities (and this may partially explain the delay in the PHP community adopting secure randomness generation functions). Showing that such attacks are in fact very practical is the objective of our work.

In this paper we develop generic techniques and algorithms to exploit randomness vulnerabilities in PHP applications. We describe implementation issues that allow one to either predict or completely recover the initial seed of the PRNGs used in most web applications. We also give algorithms for recovering the internal state of the PRNGs used by the PHP system, including the Mersenne twister generator and the glibc LFSR based generator, even when their output is truncated. These algorithms could be used in order to attack hardened PHP installations even when strong seeding is employed, as it is done by the Suhosin extension for PHP and they may be of independent interest.

We also conducted an extensive audit of several popular PHP applications. We focused on the security of password reset implementations. Using our attack framework we were able to mount attacks that take over arbitrary user accounts with practical complexity. A number of widely used PHP applications are affected (see Figure 13), while we believe that the impact is even larger in less known applications.

Our results suggest that randomness attacks should be considered practical for PHP applications and existing systems should be audited for these vulnerabilities. Weak randomness is a grave vulnerability in any secure system as it was also recently demonstrated in the widely publicized discovery of common primes in RSA public-keys by Lenstra et al. [14]. We finally stress that our techniques apply in any setting beyond PHP, whenever the same PRNG functions are used and the attack vector relies on predicting a system defined random object.

Overview of the paper. Next, (in section 1.1) we present the attack model that we assume in this work, followed by an overview of our attacks in section 1.2. Then, in section 2 we discuss the specifics of the PHP system including the functions that we target. One main source of entropy for the PHP system is time – we focus on this in section 3 where we also describe two adversarial strategies for reducing this entropy source substantially. Utilizing this and a number of other techniques we present our seed attacks which are attacks that recover the seed for the randomness generators of PHP. The conditions that enable obtaining the seed may not feasible in many cases — this motivates section 5 where we show how one can calculate entirely the generator state given sufficient leaks from the generator. In order to achieve this we perform an experimental analysis of the Håstad-Shamir framework ([8]), develop an *online* Gaussian solver for large sparse linear systems and an algorithm for recovering the state of truncated Mersenne twister sequences. Our case studies are presented in section 6 and include a number of popular and current applications including Mediawiki, Gallery, Joomla, osCommerce and others. In section 7 we present how to defend against our attacks and related work is provided in section 8.

1.1 Attack model

In Figure 1 we present our general attack template. An attacker is trying to predict the password reset token in order to gain another user's privileges (say an administrator's). Each time the attacker makes a request to the web server, his request is handled by a web application instance, usually represented by a specific operating system process, which contains some process specific state. The web application uses a number of application objects with values depending on its internal state, with some of these objects leaking to the attacker through the web server responses. Examples of such objects are session identifiers and outputs of PRNG functions. Although our focus is in password reset functions, the principles that we use and the techniques that we develop can be readily applied in other contexts when the application relies on

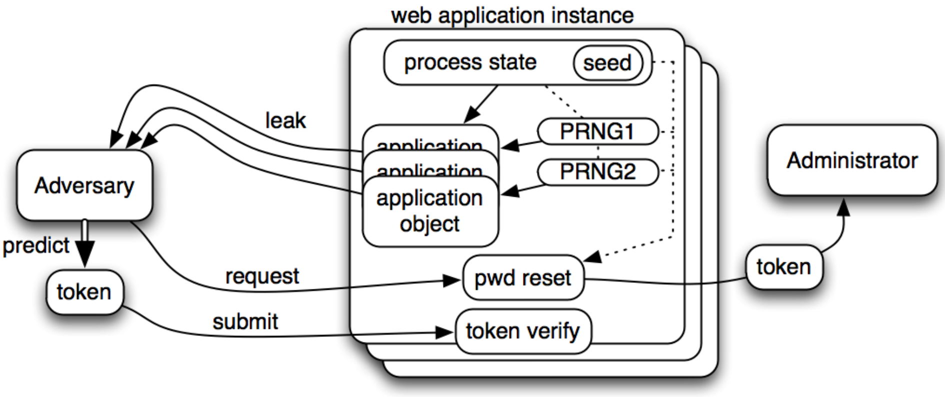

Figure 1: Attack template.

the generation of random values for security applications. Examples of such applications are CAPTCHA's and the production of random filenames.

Attack complexity. Since we present explicit practical attacks, we define next the complexity under which an attack should be consider practical. There are two measure of complexity of interest. The first is the time complexity and the second is the query or communication complexity. For some of our attacks the main compuational operation is the calculation of an MD5 hash. With current GPU technologies an attacker can perform up to 2^{30} MD5 calculations per second with a $250 GPU, while with an additional $500 can reach up to 2^{32} calculations [9]. These figures suggest that attacks that require up to 2^{40} MD5 calculations can be easilty mounted. In terms of communication complexity, most of our attacks have a query complexity of a few thousand requests at most, while some have as little as a few tens of requests. Our most communication intensive attacks (section 5) require less than 35K($\approx 2^{15}$) requests. Sample benchmarks that we performed in various applications and server installations show that on average one can perform up to 2^{22} requests in the course of a day.

1.2 Overview of our attacks

The PHP system utilizes a number of entropy sources in order to produce random numbers. The web application accesses these sources through a number of functions which we describe in section 2 that also include pseudorandom number generators seeded with them. We describe attacks against both the entropy sources and the pseudorandom number generators. Our attacks can be classified in three different types. The first type is attacks that enable the attacker to reduce the entropy of the sources and specifically of time. These attacks, namely Adversarial Time Synchronization (ATS) and Request Twins are presented in section 3. The second type is attacks against the seeding of the random number generators. The goal of these attacks is to produce the seed of the generators. We exploit two different principles, that is the reusage of entropy sources among different objects of the PHP system, and the small size of the seeds. The former enables us to reconstruct the seed (section 4.2) while the latter to bruteforce (recover) the seed value (sections 4.1,4.3). In the third type, we exploit the linearity of the pseudorandom number generators in order to recover their internal state, after observing a sufficient number of outputs. We describe two attacks against the main PRNG's used by the PHP system, i.e., `mt_rand()` (section 5.3) and `rand()` (section 5.4). These attacks need to obtain all the PRNG outputs from the same process generator. To meet this goal we design a generic process distinguisher that enable the attacker to lock into a specific server process. The overview of all attacks is presented in figure 2.

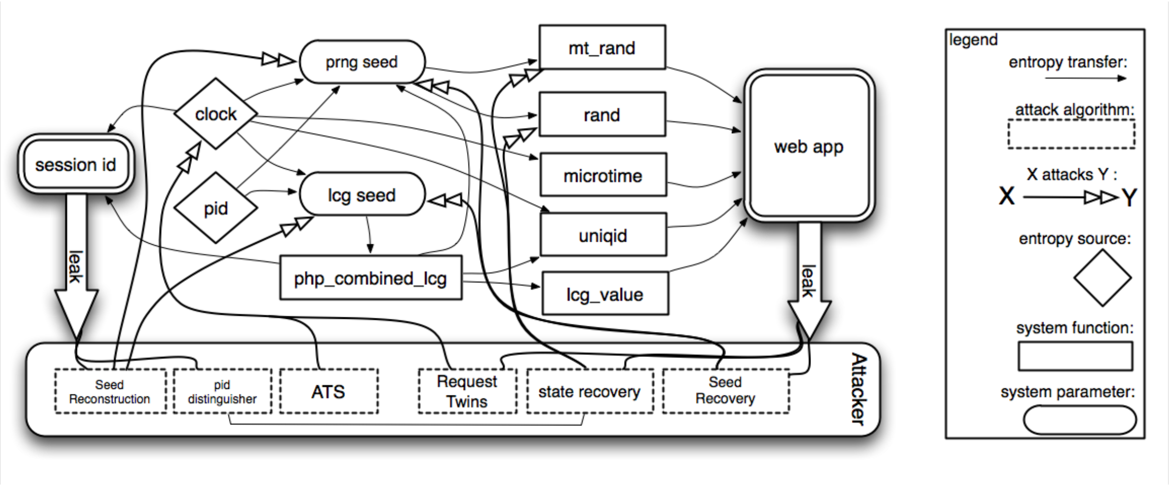

Figure 2: Attacks overview

2 PHP System

We will now describe functionalities of the PHP system that are relevant to our attacks. We first describe the different modes in which PHP might be running, and then we will do a description of the randomness generation functions in PHP. We focus our analysis in the Apache web server, the most popular web server at the time of this writing, however our attacks are easily ported to any webserver that meets the configuration requirements that we describe for each attack.

2.1 Proccess management

There are different ways in which a PHP script is executed. These ways affect its internal states, and thus the state of its PRNGs.

- **mod_php:** Under this installation the Apache web server is responsible for the process management. When the server is started a number of child proccesses are created and each time the number of occupied processes passes a certain threshold a new process is created. Conversely, if the idle proccesses are too many, some processes are killed. One can specify a maximum number of requests for each process although this is not enabled by default. Under this setting each PHP script runs in the context of one of the child processes, so its state is preserved under multiple connections unless the process is killed by the web server process manager. The configuration is similar in the case the web server uses threads instead of processes.

- **CGI:** Under this installation, scripts are executed with the Common Gateway Interface (CGI). Each time a request is dispatched to the web server, the server assosiates the request with an executable and executes it with arguments from the request. Under this installation each request is handled by a fresh PHP process.

- **FastCGI:** In order to avoid the overhead of creating one process per request, the FastCGI protocol was created. Under this protocol, a process manager is created and each request is dispatched to an external process. However, after completing the processing, the process does not die, rather it handles other subsequent requests. The setup is like mod_php although under this protocol, depending on the configuration, it is more common to kill a process after a specific number of requests.

Keep-Alive requests. The HTTP protocol offers a request header, called Keep-Alive. When this header is set in an HTTP request, the web server is instructed to keep the connection alive after the request is served. Under mod_php installations this means that any subsequent request will be handled from the same process. This is a very important fact, that we will use in our attacks. However in order to avoid having a process hang from one connection for infinite time, most web servers specify an upper bound on the number of consequent keep-alive requests. The default value for this bound in the Apache web server is 100.

2.2 Randomness Generation

In order to satisfy the need for generating randomness in a web application, PHP offers a number of different randomness functions. We briefly describe each function below.

- `php_combined_lcg()/lcg_value()`: the `php_combined_lcg()` function is used internally by the PHP system, while `lcg_value()` is its public interface. This function is used in order to create sessions, as well as in the uniqid function described below to add extra entropy. It uses two linear congruential generators (LCGs) which it combines in order to get better quality numbers. The output of this function is 64 bits.
- `uniqid(prefix, extra_entropy)`: This function returns a string concatenation of the seconds and microseconds of the server time converted in hexadecimal. When given an additional argument it will prefix the output string with the prefix given. If the second argument is set to true, the function will suffix the output string with an output from the `php_combined_lcg()` function. This makes the total output to have length up to 15 bytes without the prefix.
- `microtime()`, `time()`: The function `microtime()` returns a string concatenation of the current microseconds divided by 10^6 with the seconds obtained from the server clock. The `time()` function returns the number of seconds since Unix Epoch.
- `mt_srand(seed)/mt_rand(min, max)`: `mt_rand` is the interface for the Mersenne Twister (MT) generator [15] in the PHP system. In order to be compatible with the 31 bit output of `rand()`, the LSB of the MT function is discarded. The function takes two optional arguments which map the 31 bit number to the $[\texttt{min}, \texttt{max}]$ range. The `mt_srand()` function is used to seed the MT generator with the 32 bit value `seed`; if no seed is provided then the seed is provided by the PHP system.
- `srand(seed)/rand(min, max)`: rand is the interface function of the PHP system to the `rand()` function provided by libc. In unix, `rand()` additive feedback generator (resembling a Linear Feedback Shift Register (LFSR)), while in Windows it is an LCG. The numbers generated by `rand()` are in the range $[0, 2^{31} - 1]$ but like before the two optional arguments give the ability to map the random number to the range $[\texttt{min}, \texttt{max}]$. Like before the `srand()` function seeds the generator similarly to the `mt_srand()` function.
- `openssl_random_pseudo_bytes(length, strong)`: This function is the only function available in order to obtain cryptographically secure random bytes. It was introduced in version 5.3 of PHP and its availability depends on the availability of the openssl library in the system. In addition, until version 5.3.4 of PHP this function had performance problems [1] running in Windows operating systems. The `strong` parameter, if provided, is set to `true` if the function returned cryptographically strong bytes and `false` otherwise. For these reasons, and for backward compatibility, its use is still very limited in PHP applications.

In addition the application can utilize an operating system PRNG (such as `/dev/urandom`). However, this does not produce portable code since `/dev/urandom` is unavailable in Windows OS.

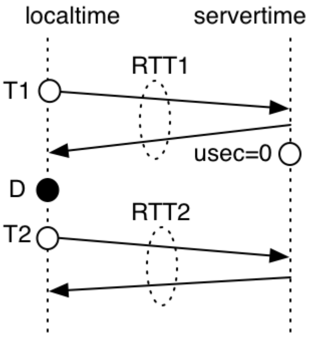

Figure 3: ATS.

Configuration		ATS			Req. Twins		
CPU(GHz)	RTT(ms)	min	max	avg	min	max	avg
1 × 3.2	1.1	0	4300	410	0	1485	47
4 × 2.3	8.2	5	76693	4135	565	1669	1153
1 × 0.3	9	53	39266	2724	1420	23022	4849
2 × 2.6	135	73	140886	83573	2	1890	299

Figure 4: Effectiveness of our time entropy lowering techniques against four servers of different computational power and RTT. Time measurements are in microseconds.

3 The entropy of time measurements

Although ill-advised (e.g., [3]) many web applications use time measurements as an entropy source. In PHP, time is accessed through the `time()` and `microtime()` functions. Consider the following problem. At some point a script executing a request made by the attacker makes a time measurement and use the results to, say, generate a password reset token. The attacker's goal it to predict the output of the measurement made by the PHP script. The `time()` function has no entropy at all from an attacker point of view, since the server reveals its time in the HTTP response header as dictated by the HTTP protocol. On the other hand, microtime ranges from 0 to 10^6 giving a maximum entropy of about 20 bits. We develop two distinct attacks to reduce the entropy of `microtime()` that have different advantages and mostly target two different scenarios.

The first one, Adversial Time Synchronization, aims to predict the output of a specific time measurement when there is no access to other such measurements. The second, Request Twins, exploits the fact that the script may enable the attacker to generate a correlated leak to the target measurement.

Adversarial Time Synchronization (ATS). As we mentioned above, in each HTTP response the web server includes a header containing the full date of the server including hour, minutes and seconds. The basic observation is that although we get no leak regarding the microseconds from the HTTP date header we know that when a second changes the microseconds are zeroed. We use this observation to narrow down their value.

In the following we represent time by a tuple $T = (T_{sec}, T_{usec})$ as it is usually represented in operating system time structures. In addition we define RTT as the time interval between sending an HTTP request and receiving the response. Finally we denote by $DATE_{sec}$ the epoch corresponding to the Date HTTP header received. The algorithm proceeds as follows: We connect to the web server and issue pairs of HTTP requests $R1$ and $R2$ in corresponding times $T1$ and $T2$ until a pair is found in which the date HTTP header of the corresponding responses is different. At that point we know that between the processing of the two HTTP requests the microseconds of the server were zeroed. We proceed to approximate the time of this event S in localtime, denoted by the timestamp D, by calculating the average RTT of the two requests and offsetting the middle point between $T2$ and $T1$ by this value divided by two. Now, whenever the localtime is given by a timestamp D', the server time is approximated by $S' = S + (D' - D)$.

In the Apache web server the date HTTP header is set after processing the request of the user. If the attacker requests a non existent file, then the point the header is set is approximatelly the point that a valid request will start executing the PHP script. It follows that if the attacker uses ATS with HTTP requests to not existent files then he will synchronize approximately with the beggining of the script's execution.

If the attacker tries to determine the output of the `microtime()` function at a point deeper in a PHP script, then the script's execution time should be taken into account as well as the position within the script that the function is called. Calls early in the script are much easier to synchronize with, than nested calls which are preceded with file and database operations. In principle, the attacker can improve the accuracy of the algorithm by performing offline experiments and estimate the average execution time of the script.

Given a steady network where each request takes $\frac{RTT}{2}$ time to reach the target server, our algorithm deviation depends only on the rate that the attacker can send HTTP requests. In practice, we find that the algorithm's main source of error is the network distance between the attacker's system and the server cf. Figure 4. The above implementation we described is a proof-of-concept and various optimizations can be applied to improve its accuracy.

Request Twins. Consider the following setting: an application uses `microtime()` to generate a password token for any user of the system. The attacker has access to a user account of the application and tries to take over the account of another user. This allows the attacker to obtain password reset tokens for his account and thus outputs of the `microtime()` function. The key observation is that if the attacker performs in rapid succession two password reset requests, one for his account and one for the target user's account, then these requests will be processed by the application with a very small time difference and thus the conditional entropy of the target user's password reset token given the attacker's token will be small. Thus, the attacker can generate a token for an account he owns and in fast succession a token for the target account. Then the `microtime()` used for generating the token of his account can be used to approximate the `microtime()` output that was used for the token of the target account.

Experiments. We conducted a series of experiments for both our algorithms using the following setup. We created a PHP "time" script that prints out the current seconds and microseconds of the server. To evaluate the ATS algorithm we first performed synchronization between a client and the server and afterwards we sent a request to the time script and tried to predict the value it would return. To evaluate the Request Twins algorithm we submitted two requests to the time script in fast succession and measured the difference between the output of the two responses.

In Figure 4 we show the time difference between the server's time and our client's calculation for four servers with different CPU's and RTT parameters. Our experiments suggest that both algorithms significantly reduce the entropy of microseconds (up to an average of 11 bits with ATS and 14 bits with Request Twins) having different advantages each. Specifically, the ATS algorithm seems to be affected by large RTT values while it is less affected by differences in the CPU speed. The situation is reversed for Request Twins where the algorithm is immune to changes in the RTT however, it is less effective in old systems with low processing speed.

4 Seed Attacks

In this section we describe attacks that allow either the recovery or the reconstruction of the seeds used for the PHP system's PRNGs. This allows the attacker to predict all future iterations of these functions and hence reduces the entropy of functions `rand()`,`mt_rand()`, `lcg_value()` as well as the extra entropy argument of `uniqid()` to zero bits. We exploit two properties of the seeds used in these functions. The first one is the reusage of entropy sources between different seeds. This enables us to reconstruct a seed without any access to outputs of the respective PRNG. The second is the small entropy of certain seeds that allows one to recover its value by bruteforce.

We present three distinct attacks. The first attack allows one to recover the seed of the internal LCG seed used by the PHP system using a session identifier. Using that seed our second attack reconstructs the seed of `rand()` and `mt_rand()` functions from the elements of the LCG seed without any access to outputs of these functions. Finally, we exploit the fact that the seed used in these functions is small enough for someone with access to the output of these functions to recover its value by bruteforce.

Generating fresh processes. Our attacks on this section rely on the ability of the attacker to connect to a process with a newly initialized state. We describe a generic technique against `mod_php` in order to achieve a connection to a fresh process. Recall that in mod_php when the number of occupied processes passes a certain threshold new processes are created to handle the new connections. This gives the attacker a way to force the creation of fresh processes: The attacker creates a large number of connections to the target server with the keep-alive HTTP header set. Having occupied a large number of processes the web server will create a number of new processes to handle subsequent requests. The attacker, keeping the previous connections open, makes a new one which, given that the attacker created enough connections, will be handled by a fresh process. In addition, some of our attacks are applicable to cgi mode where by default each request is handled by a fresh process.

4.1 Recovering the LCG seed from Session ID's

In this section we present a technique to recover the `php_combined_lcg()` seed using a PHP session identifier. In PHP, when a new session is created using the respective PHP function (`session_start()`), a pseudorandom string is returned to the user in a cookie, in order to identify that particular session. That string is generated using a conjuction of user specific and process specific information, and then is hashed using a hash function which is by default MD5, however there is an option to use other hash functions such as SHA-1. The values contained in the hash are:

- Client IP address (32 bits).
- A time measurement: Unix epoch and microseconds (32 + 20 bits).
- A value generated by `php_combined_lcg()` (64 bits).

For the following denote by T_0 the time measurement used above. The maximum entropy of the above values is 148 bits in total, way out of the bruteforce range for any system. However, because in the context of our attack model the attacker controls each request, he knows exactly most of the values. Specifically, the client IP address is the attacker's IP address and the Unix Epoch can be determined through the date HTTP header.

In addition, if `php_combined_lcg()` is not initialized at the time the session is created, as it happens when a fresh process is spawned, then it is seeded. The state of the `php_combined_lcg()` is two registers s_1, s_2 of size 32 bits each, which are initialized as follows. Let T_1 and T_2 be two subsequent time measurements. Then we have that

$$s_1 = T_1.sec \oplus (T_1.usec \ll 11) \text{ and } s_2 = pid \oplus (T_2.usec \ll 11)$$

where *pid* denotes the current process id, or if threads are used the current thread id [2].

Process id's have a range of 2^{15} values in Linux systems [3]. In Windows systems the process id's (resp. threads) are also at most 2^{15} unless there are more than 2^{15} active processes (resp. threads) in the system which is a very unlikely occurence.

Observe now that the session calculation involves three time measurements T_0, T_1 and T_2. Given that these three measurements are conducted succesivelly it is advantageous to estimate

[2] In PHP versions before 5.3.2 the seed used only one time measurement which made it even weaker.

[3] On Linux systems thread id's are 32 bits which makes the attack infeasible. However mod_php in most distributions run exclusively with Apache with multiple processes (prefork server).

their entropy by examining the random variables $T_0, \Delta_1 = T_1 - T_0, \Delta_2 = T_2 - T_1$. We conducted experiments in different systems to estimate the range of values for Δ_1 and Δ_2. Our experiments suggest that $\Delta_1 \in [1, 4]$ while $\Delta_2 \in [0, 3]$. We also found a positive linear correlation in the values of the two pairs. This enables a cutdown of the possible valid pairs. These results suggest that the additionally entropy introduced by the two Δ variables is at most 5 bits.

To summarize, the total remaining entropy of the session identifier hash is the sum of the microseconds entropy from T_0 ($\approx$ 20 bits) the two Δ variables ($\approx$ 5 bits) and the process identifier(15 bits). These give a total of 40 bits which is tractable cf. section 1.1. Furthermore the following improvements can be made: (1) Using the ATS algorithm the microseconds entropy can be reduced as much as 11 bits on average. (2) The attacker can make several connections to fresh processes instead of one, in rapid succession, obtaining session identifiers from each of the processes. Because the requests were made in a small time interval the preimages of the hashes obtained belong into the same search space, thus improving the probability of inverting one of the preimages proportionally to the number of session identifiers identifiers obtained. Our experiments with the request twins technique suggest that at least 4 session identifiers can be obtained from within the same search space thus offering a reduction of at least two bits. Adding these improvements reduces the search time up to 2^{27} MD5 computations which is dramatically shorter than the perceived 2^{148} and can be achieved in a matter of a few minutes using a modern GPU.

In summary the technique described in this section allows an attacker to obtain the following process specific information given access to a session id from a fresh process:

- The process id of the process handling the request.
- The seed of `php_combined_lcg()` function.

4.2 Reconstructing the PRNG Seed from Session ID's

In this section we exploit the fact that the PHP system reuses entropy sources between different generators, in order to reconstruct the PRNG seed used by `rand()` and `mt_rand()` functions from a PHP session identifier. In order to predict the seed we only need to find a preimage for the session id, using the methods described in the previous section. One advantage of this attack is that it requires no outputs from the affected functions.

When a new process is created the internal state of the functions `rand()` and `mt_rand()` is uninitialized. Thus, when these functions are called for the first time within the script a seed is constructed as follows:

$$\mathit{seed} = (\mathit{epoch} \times \mathit{pid}) \oplus (10^6 \times \texttt{php\_combined\_lcg}())$$

where *epoch* denotes the seconds since epoch and *pid* denotes the process id of the process handling the request. It it easy to notice, that an attacker with access to a session id preimage has all the information needed in order to calculate the seed used to initialize the PRNGs since:

- *epoch* is easily obtained through the HTTP Date header.
- *pid* is known from the seed of the `php_combined_lcg()` obtained through the preimage of the session identifier from section 4.1.
- `php_combined_lcg()` is also known, since the attacker has access to its seed, he can easily predict the next iteration after the initial value.

In summary the technique of this section allows the reconstruction of the seed of the `mt_rand()` and `rand()` functions given access to a PHP session id of a fresh process. The time complexity of the attack is the same as the one described in section 4.1 while the query complexity is one request, given that the attacker spawned a fresh process (which itself requires only a handful of requests).

4.3 Recovering the Seed from Application leaks

In contrast to the technique presented in the previous section, the attack presented here recovers the seed of the PRNG functions `rand()` and `mt_rand()` when the attacker has access to the output of these functions. We exploit the fact that the seed used by the PHP system is only 32 bits. Thus, an attacker who connects to a fresh process and obtains a number of outputs from these functions can bruteforce the 32 bit seed that produces the same output.

We emphasize that this attack works even if the outputs are truncated or passed through transformations like hash functions. The requirements of the attack is that the attacker can define a function from the set of all seeds to a sufficiently large range and can obtain a sample of this function evaluated on the seed that the attacker tries to recover. Additionally for the attack to work this function should behave as a random mapping.

Given a sample of this function the attacker tries to find a preimage that will recover the seed of the PRNG function. The attacker can try all possible seeds until one matches the output he obtained. In order for the seed obtained to be the correct one the function should be near collision free. Assuming that the function is a random map we can use the following lemma to bound the size of the range of the mapping as the function of the probability of whether a collision exists for the given mapping.

Lemma 4.1. *Let f be a random function $f : [n] \to [m]$. Then the probability that f is $1-1$ is at least $1-\epsilon$ assuming $\log m \geq 2\log n + \log \epsilon^{-1}$.*

Proof. The probability of the event A that f is $1-1$ is

$$Pr[A] = \frac{(m)_n}{m^n} \geq (1 - \frac{m}{n})^n \geq 1 - \frac{n^2}{m}$$

By requiring this probability to be greater than $1-\epsilon$ we obtain the lemma statement.

Using the above lemma the attacker can define his function to have a large enough range for his search to be succesful with very high probability. Note that in case of an incorrect seed calculation due to a collision, the attacker may continue to recover other preimages.

Consider the following example. The attacker has access to a user account of an application which generates a password reset token as 6 symbols where each symbol is defined as $g($`mt_rand()`$)$ where g is a table lookup function for a table with 60 entries containing alphanumeric characters. The attacker defines the function f to be the concatenation of two password reset tokens generated just after the PRNG is initialized. The attacker samples the function by connecting to a fresh process and resetting his password two times. Since the table of function g contains 60 entries, the attacker obtains 6 bits per token symbol, giving a total range to the function f of 72 bits. Using the above lemma we get that the function defined by the attacker has an upper bound on the collision probability of 2^{-8}. This means that the first preimage returned by the bruteforce search is correct with probability at least $1-2^{-8}$ and if not he may continue the search to the next preimage.

The time complexity of the attack is 2^{32} calculations of f however, we can reduce the online complexity of the attack using a time-space tradeoff. In this case the online complexity of the attack can be as little as 2^{16}. The query complexity of the attack depends on the number of requests needed to obtain a sample of f. In the example given above the query complexity is two requests.

5 State recovery attacks

One can argue that randomness attacks can be easily thwarted by increasing the entropy of the seeding for the PRNG functions used by the PHP system. For example, the suhosin PHP

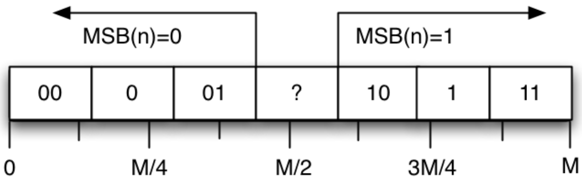

Figure 5: Mapping a random number $n \in [M]$ to 7 buckets and the respective bits of n that are revealed given each bucket.

hardening extension replaces the `rand()` function with a Mersene Twister generator with separate state from `mt_rand()` and offers a larger seed for both generators getting entropy from the operating system[4].

We show that this is not the case. We exploit the algebraic structure of the PRNGs used in order to recover their internal state after a sufficient number of past outputs (leaks) have been observed by the attacker. Any such attack has to overcome two challenges. First, web applications usually need only a small range of random numbers, for example to sample a random entry from an array. To achieve that, the PHP system maps the output of the PRNG to the given range, an action that may break the linearity of the generators. Second, in order to collect the necessary leaks the attacker may need to reconnect to the same process many times to collect the leaks from the same generator instance. Since, usually, there are many PHP processes running in the system, this poses another challenge for the attacker.

In this section we present state recovery algorithms for the truncated PRNG functions `rand()` and `mt_rand()`. The algorithm for the latter function is novel, while regarding the former we implement and evaluate the Håstad-Shamir cryptanalytic framework [8] for truncated linearly related variables. We begin by discussing the way truncation takes place in the PHP system. Afterwards, we tackle the problem of reconnecting into the same server process. Finally we present the two algorithms against the generators.

5.1 Truncating PRNG sequences in the PHP system

As mentioned in section 2.2 the `rand()` and `mt_rand()` functions can map their output to a user defined range. This has the effect of truncating the functions' output. Here we discuss the process of truncating the output and its implications for the attacker.

Let $n \in [M] = \{0, \ldots, M-1\}$ be a random number generated by `rand()` or `mt_rand()`, where $M = 2^{31}$ in the PHP system. In order to map that number in the range $[a, b]$ where $a < b$ the PHP system maps n to a number $l \in [a, b]$ in the following way:

$$l = a + \frac{n \cdot (b - a + 1)}{M}$$

We can view the process above as a mapping from the set of numbers in the range $[M]$ to $b - a + 1$ "buckets." Our goal is to recover as many bits as possible of the original number n. Observe that given l it is possible to recover immediately up to $\lfloor \log(b - a + 1) \rfloor$ most significant bits (MSB) of the original number n as follows:

Given that n belongs to bucket l we obtain the following range for possible values for n:

$$\lfloor \frac{(l-a) \cdot M}{b-a+1} \rfloor \leq n \leq \lfloor \frac{(l-a+1) \cdot M}{b-a+1} \rfloor - 1$$

[4] The suhosin patch installed in some Unix operating systems by default does not include the randomness patches, rather than it mainly offers protection from memory corruption attacks. The full extension is usually installed separately from the PHP packages.

Therefore, given a bucket number l we are able to find an upper and lower bound for the original number denoted respectively by L_l and U_l. In order to recover a part of the original number n one can simply find the number of most significant bits of L_l and U_l that are equal and observe that these bits would be the same also in the number n. Therefore, given a bucket l we can compare the MSBs of both numbers and set the MSBs of n to the largest sequence of common most significant bits of L_l, U_l.

Notice that in some cases even the most significant bit of the two numbers are different, thus we are be unable to infer any bit of the original number n with absolute certainty. For example, in Figure 5 given that a number falls in bucket 3 we have that $920350134 \leq n \leq 1227133512$. Because $920350134 < 2^{30}$ and $1227133512 > 2^{30}$ we are unable to infer any bit of the original number n.

Another important observation is that this specific truncation algorithm allows the recovery of a fragment of the MSBs of the original number. Therefore, in the following sections we will assume that the truncation occurs in the MSBs and we will describe our algorithms based on MSB truncated numbers. However, all algorithms described work for any kind of truncation.

5.2 Process distinguisher

As we mentioned in section 2.1, if one wants to receive a number of leaks from the same PHP process one can use keep-alive requests. However, there is an upper bound that limits the number of such requests (by default 100). Therefore, if the attacker needs to observe more outputs beyond the keep-alive limit the connection will drop and when the attacker connects back to the server he may be served from a different process with a different internal state. Therefore, in order to apply state recovery attacks (which typically require more than 100 requests), we must be able to submit all the necessary requests to the same process. In this section, we will describe a generic technique that finds the same process over and over using the PHP session leaks described in section 4.1.

While we cannot avoid disconnecting from a process after we have submitted the maximum number of keep-alive requests, we can start reconnecting back to the server until we hit the process we were connected before and continue to submit requests. The problem in applying this approach is that it is not apparent to distinguish whether the process we are currently connected to is the one that was serving us in the previous connection. To distinguish between different processes, we can use the preimage from a session identifier. Recall that the session id contains a value from the `php_combined_lcg()` function, which in turn uses process specific state variables. Thus, if the session is produced from the same process as before then the `php_combined_lcg()` will contain the next state from the one it was before. This gives us a way to find the correct process among all the server processes running in the server. In summary the algorithm will proceed as follows:

1. The attacker obtains a session identifier and a preimage for that id using the techniques discussed in section 4.1.

2. The attacker submits the necessary requests to obtain leaks from the PRNG he is attacking, using the keep-alive HTTP header until the maximum number of requests is reached.

3. The attacker initiates connections to the server requesting session identifiers. He attempts to obtain a preimage for every session identifier using the next value of the `php_combined_lcg()` from the one used before or, if the server has high traffic, the next few iterations. If a preimage is obtained the attacker repeats step 2, until all necessary leaks are obtained.

Notice that obtaining a preimage after disconnecting requires to bruteforce a maximum number of 20 bits (the microseconds), and thus testing for the correct session id is an efficient procedure. Even if the application is not using PHP sessions, or if a preimage cannot be obtained, there are other, application specific, techniques in order to find the correct process. We describe some possibilites:

- Getting a leak from the `lcg_value()` or `uniqid()` with the extra entropy option set to *true*. This will have the same effect of leaking an output of the `php_combined_lcg()` to the attacker.
- Combining vulnerabilities. For example, a file read vulnerability in a Linux system will allow one to read the world readable `/proc/self/stat` file which contains the pid of the process handling the request.

A generic technique for Windows. In the case of Windows systems the attacker can employ another technique to collect the necessary leaks from the same process in case the server has low traffic. In unix servers with apache preforked server + `mod_php` all idle processes are in a queue waiting to handle an incoming client. The first process in the queue handles a client and then the process goes to the back of the queue. Thus, if an attacker wants to reconnect to the same process without using some process distinguisher he will need to know exactly the number of processes in the system and if there are any intermediate requests by other clients while the attacker tries to reconnect to the same process. However, in Windows prethreaded server with `mod_php` things are slightly better for an attacker. Threads are in a priority queue and when a thread in the first place of the queue handles a request from a client it returns again in that first place and handles the first subsequent incoming request. Thus, an attacker which manages to connect to that first thread of the server, can rapidly close and reopen the connections thus leaving a very small window in which that thread could be occupied by another client. Of course, in high traffic servers the attacker would have a difficulty connecting in a time when the server is idle in the first place. Nevertheless, techniques exist [16] to remotely determine the traffic of a server and thus allow the attacker to find an appropriate time window within which he will attempt this attack.

Based on the above, in the following sections we will assume that the attacker is able to collect the necessary number of leaks from the targeted function.

5.3 State recovery for mt_rand()

The `mt_rand()` function uses the Mersenne Twister generator in order to produce its output. In this section we give a description of the Mersenne Twister generator and present an algorithm that allows the recovery of the internal state of the generator even when the output is truncated. Our algorithm also works in the presence of non consecutive outputs as in the case resulting from the buckets truncation algorithm of the PHP system (cf. section 5.1).

Mersenne Twister. Mersenne Twister, and specifically the widely used MT19937 variant, is a linear PRNG with a 624 32-bit word state. The MT algorithm is based on the following recursion: for all k,

$$x_{k+n} = x_{k+m} \oplus ((x_k \wedge 0x80000000)|(x_{k+1} \wedge \text{0x7fffffff}))A$$

where $n = 624$ and $m = 397$. The logical AND operation with 0x80000000 discards all but the most significant bit of x_k while the logical AND with 0x7fffffff discards only the MSB of x_{k+1}. A is a 32×32 matrix for which multiplication by a vector x is defined as follows:

$$xA = \begin{cases} (x \gg 1) & \text{if } x^{31} = 0 \\ (x \gg 1) \oplus a & \text{if } x^{31} = 1 \end{cases}$$

Here $a = (a^0, a^1, ..., a^{31}) = $ 0x9908B0DF is a constant 32-bit vector (note that we use x^{31} to denote the LSB of a vector x). The output of this recurrence is finally multiplied by a 32×32 non singular matrix T, called the tempering matrix, in order to produce the final output $z = xT$.

State recovery. Since the tempering matrix T is non singular, given 624 outputs of the MT generator one can easily compute the original state by multiplying the output z with the inverse matrix T^{-1} thus obtaining the state variable used as $x_i = z_i T^{-1}$. After recovering 624 state variables one can predict all future iterations. However, when the output of the generator is truncated, predicting future iterations is not as straightforward as before because it is not possible to locally recover all needed bits of the state variables given the truncated output.

The key observation in recovering the internal state is that due to the fact that the generator is in GF(2) the truncation does not introduce non linearity even though there are missing bits from the respective equations. Thus, we can express the output of the generator as a set of linear equations in GF(2) which, when solved, yield the initial state that produced the observed sequence. From the basic recurrence of MT we can derive the following equations for each individual bit:

Lemma 5.1. *Let $x_0, x_1, \ldots$ be an MT sequence and $j > 0$. Then the following equations hold for any $k \geq 0$:*

1. $x^0_{jn+k} = x^0_{(j-1)n+k+m} \oplus (x^{31}_{(j-1)n+k+1} \wedge a^0)$
2. $x^1_{jn+k} = x^1_{(j-1)n+k+m} \oplus x^0_{(j-1)n+k} \oplus (x^{31}_{(j-1)n+k+1} \wedge a^1)$
3. $\forall i, 2 \leq i \leq 31 : x^i_{jn+k} = x^i_{(j-1)n+k+m} \oplus x^{i-1}_{(j-1)n+k+1} \oplus (x^{31}_{(j-1)n+k+1} \wedge a^i)$

Proof. The equations follow directly from the basic recurrence.

In addition since the tempering matrix is only a linear transformation of the bits of the state variable x_i, we can similarly express each bit of the final output of MT as a linear equation of the bits of the respective state variable. The explicit form of the equations of the tempering matrix is given in Appendix A.

To recover the initial state of MT, we generate all equations over the state bit variables $x_0, x_1, \ldots, x_{19936}$. To map any position in the MT sequence in an equation over this set of variables, we apply the equations of the lemma above recursively until all variables in the right hand side have index below 19937.

Depending on the positions observed in the MT sequence the resulting linear system will be different. We provide a visualization of the system for different truncation levels in Appendix C. The question that remains is whether that system is solvable. Regarding the case of the 31-bit truncation, i.e. only the MSB of the output word is revealed, we can use known properties of the generator in order to easily prove the following:

Lemma 5.2. *Suppose we obtain the MSB of 19937 consecutive words from the MT generator. Then the resulting linear system is uniquely solvable.*

Proof. It is known that the MT sequence is 19937-distributed to 1-bit accuracy[5]. The linear system is uniquely solvable iff the rows are linearly independent. Suppose that a set $k \leq 19937$ of rows are lineary dependent. Then the last row of the set k obtained is computable from the other members of the k-set something that contradicts the order of equidistribution of MT. □

The above result is optimal in the sense that this is the minimum number of observed outputs needed for the system to become fully determined. In the case we obtain non consecutive outputs due to truncation or application behavior, linear dependencies may arise between the resulting equations and therefore we may need a larger number of observed outputs.

Because we cannot know in advance when the system will become solvable or the equations that will be included, we employ an online version of Gaussian elimination in order to form and

[5] Suppose that a sequence is k-distributed to u-bit accuracy. Then knowledge of the u most significant bits of l words does not allow one to make any prediction for the u bits of the next word when $l < k$. This is the cryptographic interpretation of the "order of equidistribution" whose exact definition can be found in [15].

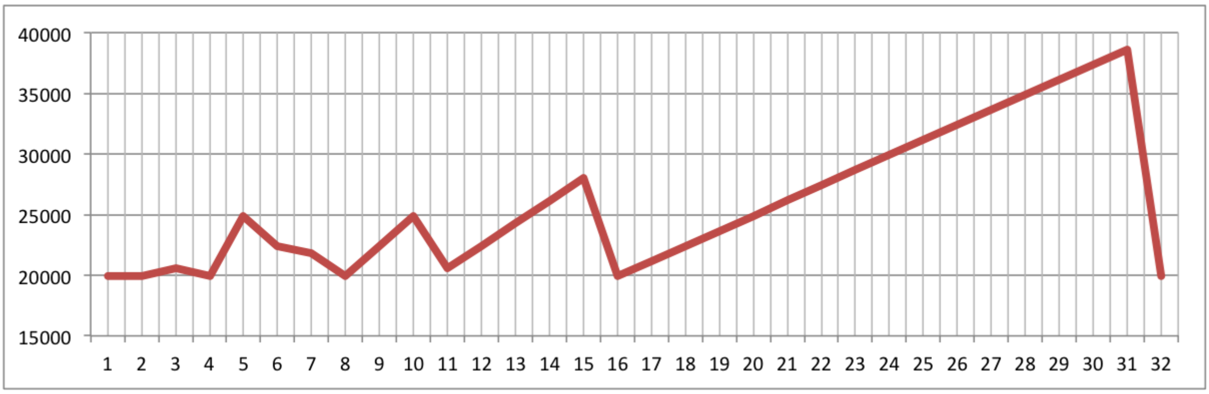

Figure 6: Solving MT; y-axis:number of equations; x-axis: number of buckets (logarithm – powers of 2).

solve the resulting system. In this way, the attacker can begin collecting leaks and gradually feed them to our Gaussian solver until he is notified that a sufficient number of independent equations have been collected. Note that regular Gaussian elimination uses both elementary row and elementary column operations. However, because we do not have in advance the entire linear system we cannot use elementary column operations. Instead we make Gaussian elimination using only elementary row operations and utilize a bookkeeping system to enter equations in their place as they are produced by the leaks supplied to the solver. The pseudocode is shown in Appendix B. Our solver employs a sparse vector representation and is capable of solving overdetermined sparse systems of tens of thousands of equations in a few minutes.

We ran a sequence of experiments to determine the solvability of the system when a different number of bits is truncated from the output. In addition we ran experiments when the outputs of the MT generator is passed through the PHP truncation algorithm, with different user defined ranges. All experiments were conducted in a 4 × 2.3 GHz machine with 4 GB of RAM.

In Figure 6 we present the number of equations needed for recovering the state of the MT sequence when a constant number of bits is truncated from the output. As it is demonstrated in the figure the number of equations needed fluctuates at or above 19937 (which is the theoretical lower bound). For the same run of experiments in Figure 7 we present the running time as a function of the number of bits obtained. Notice that worse running times occur when a large number of bits is truncated. This is probably related to the fact that the system tends to get denser when a large number of bits is truncated, cf. Appendix C.

In Figure 8 we present the number of equations needed when the PHP truncation algorithm is used. In the x-axis we have the logarithm of the number of buckets. We also show the standard deviation appearing as vertical bars. It can be seen that the number of equations needed is much higher than the theoretical lower bound of 19937 and fluctuates between 27000 and 33000. In Figure 9 we show how the number of leaks required to recover the state decreases as the number of buckets increase. Finally, for the same set of data, we present running time in seconds plotted against the number of buckets in Figure 10. As before it can be seen that the worse running times occur at smaller bucket numbers. Note that the worse running times here are bigger than the ones in Figure 7 because not only we have uneven levels of truncation but also non-consecutive leaks.

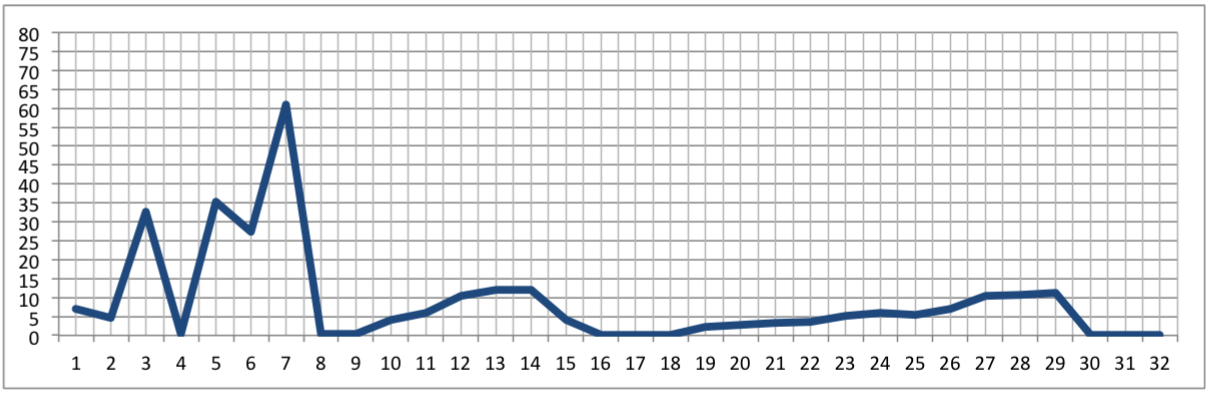

Figure 7: Solving MT; y-axis:time in seconds; x-axis: number of buckets (logarithm – powers of 2).

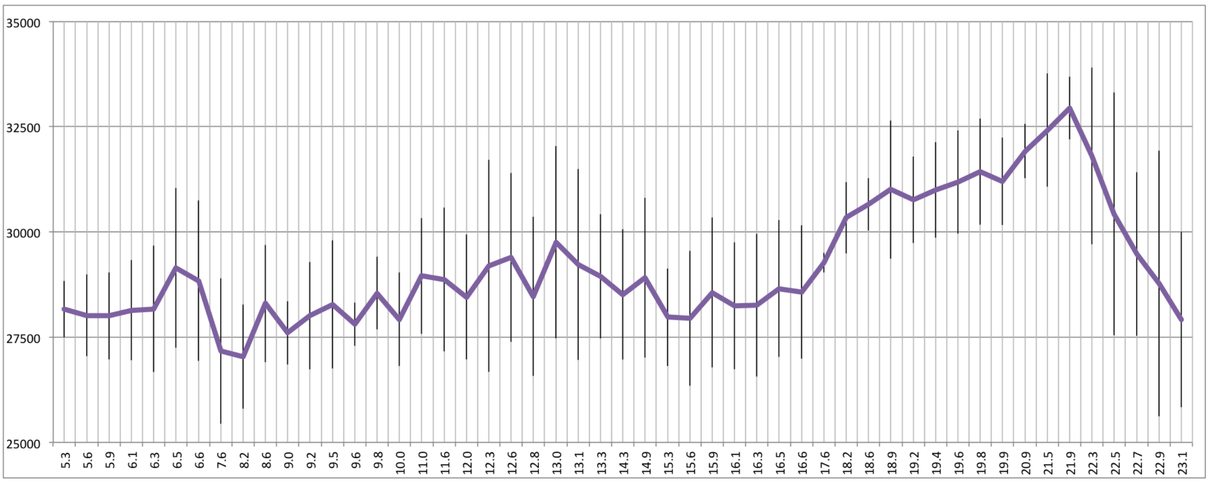

Figure 8: Solving MT; y-axis:number of equations; x-axis: number of buckets (logarithm). Standard deviation shown as vertical bars.

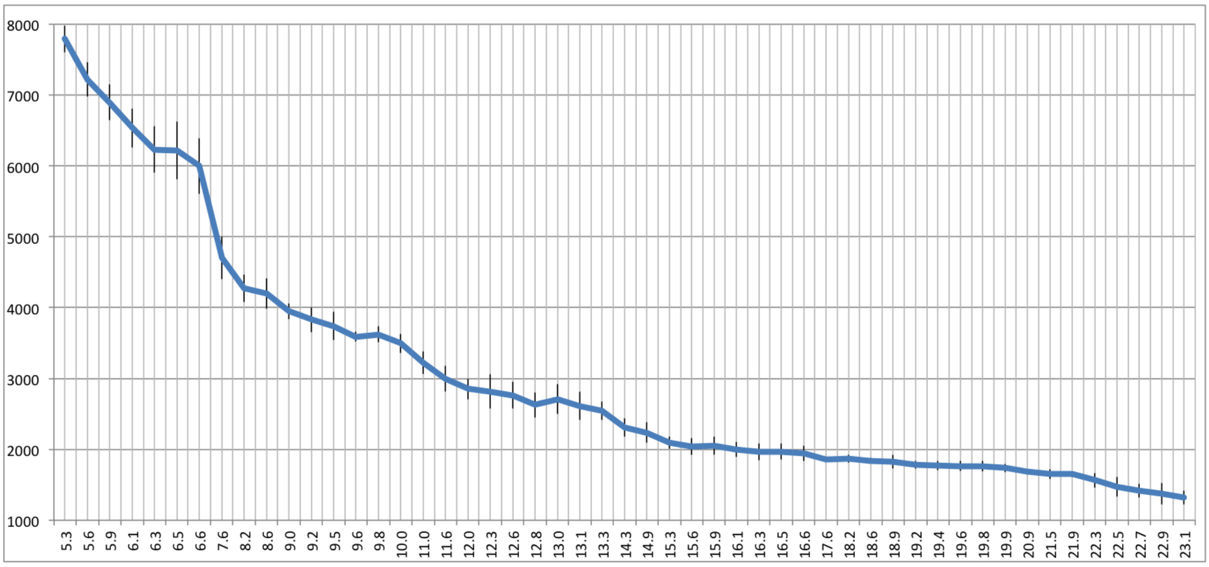

Figure 9: Solving MT; y-axis:number of leaks needed; x-axis: number of buckets (logarithm). Standard deviation shown as vertical bars.

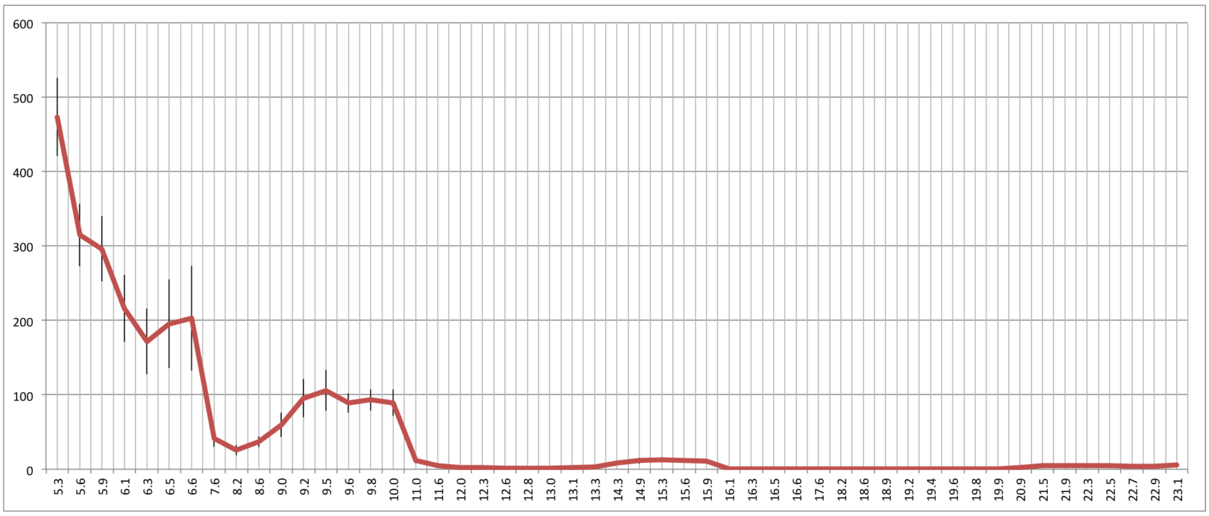

Figure 10: Solving MT; y-axis:run time in seconds; x-axis: number of buckets (logarithm). Standard deviation shown as vertical bars.

Implementation error in the PHP system. The PHP system up to current version, 5.3.10, has an error in the implementation of the Mersenne Twister generator (we discovered this during the testing of our solver). Specifically the following basic recurrence is effectively used in the PHP system due to a programming error:

$$x_{k+n} = x_{k+m} \oplus ((x_k \wedge 0x80000000)|(x_{k+1} \wedge \mathrm{0x7ffffffe})|(x_k \wedge \mathrm{0x1}))A$$

As a result the PHP system uses a different generator which, as it turns out, has slightly more linear dependencies than the MT generator. This means that probably the randomness properties of the PHP generator are poorer compared to the original MT generator.

5.4 State recovery for rand()

We turn now to the problem of recovering the state of `rand()` given a sequence of leaks from this generator. While `mt_rand()` is implemented within the PHP source code and thus is unchanged across different enviroments, the `rand()` function uses the respective function defined from the standard library of the operating system. This results in different implementations across different operating systems. There are mainly two different implementations of `rand()` one from the glibc and one from the Windows library.

Windows rand(). The `rand()` function defined in Windows is a Linear Congruential Generator (LCG). An LCG is defined by a recurrence of the form

$$X_{n+1} = (aX_n + c) \bmod m$$

Although LCGs are fast and require a small memory footprint there are many problems which make them insufficient for many uses, including of course cryptographic purposes. The parameters used by the Windows LCG are $a = 214013, c = 2531011, m = 2^{32}$. In addition, the output is truncated by default and only the top 15 bits are returned. If PHP is running in a threaded server in Windows then the parameters of the LCG used are $a = 1103515245, c = 12345, m = 2^{15}$.

Glibc rand(). In the past, glibc also used an LCG for the `rand()` function. Subsequently an LFSR-like "additive feedback" design was adopted. The generator has a state of 31 words (of 32 bits each), over which it is defined by the following recurrence:

$$r_i = (r_{i-3} + r_{i-31}) \bmod 2^{32}$$

In addition the LSB of each word is discarded and the output returned to the user is $o_i = r_i \gg 1$. An interesting note is that the man page of `rand()` states that rand is a non-linear generator. Nevertheless, the non linearity introduced by the truncation of the LSB is negligible and one can easily recover the initial values given enough outputs of the generator.

State recovery. Notice that if the generators used have a small state such as the Windows LCGs then state recovery is easy, by applying the attack from section 4 to bruteforce the entire state of the generator. However, on the Glibc generator, which has a state of 992 bits, these attacks are infeasible assuming that the state is random. Although LCGs and the Glibc generators are different, they both fall into the same cryptanalytic framework introduced by Håstad and Shamir in 1985 for recovering values of truncated linear variables. This framework allows one to uniquely solve an underdefined system of linear equations when the values of the variables are partially known. In this section we will discuss our experiences with applying this technique in the two aforementioned generators: The LCG and the additive-feedback generator of glibc. We will briefly describe the algorithm for recovering the truncated variables in order to discuss our experiments and results. The interested reader can find more information about the algorithm in the original paper [8].

Suppose we are given a system with l linear equations on k variables modulo m denoted by $x_1, x_2, \ldots, x_k$,

$$a_1^1 x_1 + a_2^1 x_2 + \cdots + a_k^1 x_k = 0 \bmod m$$
$$a_1^2 x_1 + a_2^2 x_2 + \cdots + a_k^2 x_k = 0 \bmod m$$
$$\cdots$$
$$a_1^l x_1 + a_2^l x_2 + \cdots + a_k^l x_k = 0 \bmod m$$

where $l < k$ and each variable x_i is partially known. We want to solve the system uniquely by utilizing the partial information of the k variables x_i.

We use the coefficients of the l equations to create a set of l vectors, where each vector is of the form $\mathbf{v}_i = (a_1, \ldots, a_k)$. In addition we add to this set the k vectors $m \cdot \mathbf{e}_i, 0 < i \leq k$. The cryptanalytic framework exploits properties of the lattice L that is defined as the linear span of these vectors. Observe that the dimension of L is k and in addition for every vector $\mathbf{v} \in L$ we have that

$$\sum_{i=1}^{k} v_i x_i = 0 \bmod m$$

Given the above the attack works as follows: first a lattice is defined using the recurrence that defines the linear generator; then, a lattice basis reduction algorithm is employed to create a set of linearly independent equations modulo m with small coefficients; finally, using the partially known values for each variable, we convert this set of equations to equations over the integers which can be solved uniquely. Specifically, we use the LLL [13] algorithm in order to obtain a reduced basis B for the lattice L. Now because $B = \{\mathbf{w}_j\}$ is a basis, the vectors of B are linearly independent. The key observation is that the lattice definition implies that $\mathbf{w}_j \cdot \mathbf{x} = \mathbf{w}_j \cdot (\mathbf{x}_{\text{unknown}} + \mathbf{x}_{\text{known}}) = d_j \cdot m$ for some unknown d_j. Now as long as $\mathbf{x}_{\text{unknown}} \cdot \mathbf{w}_j < m/2$ (this is the critical condition for solvability) we can solve for d_j and hence recover k equations for $\mathbf{x}_{\text{unknown}}$ which will uniquely determine it.

The original paper provided a relation between the size of $\mathbf{x}_{\text{known}}$ and the number of leaks required from the generator so that the upper bound of $m/2$ is ensured given the level of basis reduction achieved by LLL. In the case of LCGs the paper demanded the modulo m to be squarefree. However, as shown above, in the generators used it holds that $m = 2^{32}$ and thus their arguments do not apply. In addition, the lattice of the additive generator of glibc is different than the one generated by an LCG and thus needs a different analysis.

We conducted a thorough experimental analysis of the framework focusing on the two types of generators above. In each case we tested the maximum possible value of $\mathbf{x}_{\text{known}}$ to see if the $m/2$ bound holds for the reduced LLL basis. In the following paragraphs we will briefly discuss the results of these experiments for these types of generators.

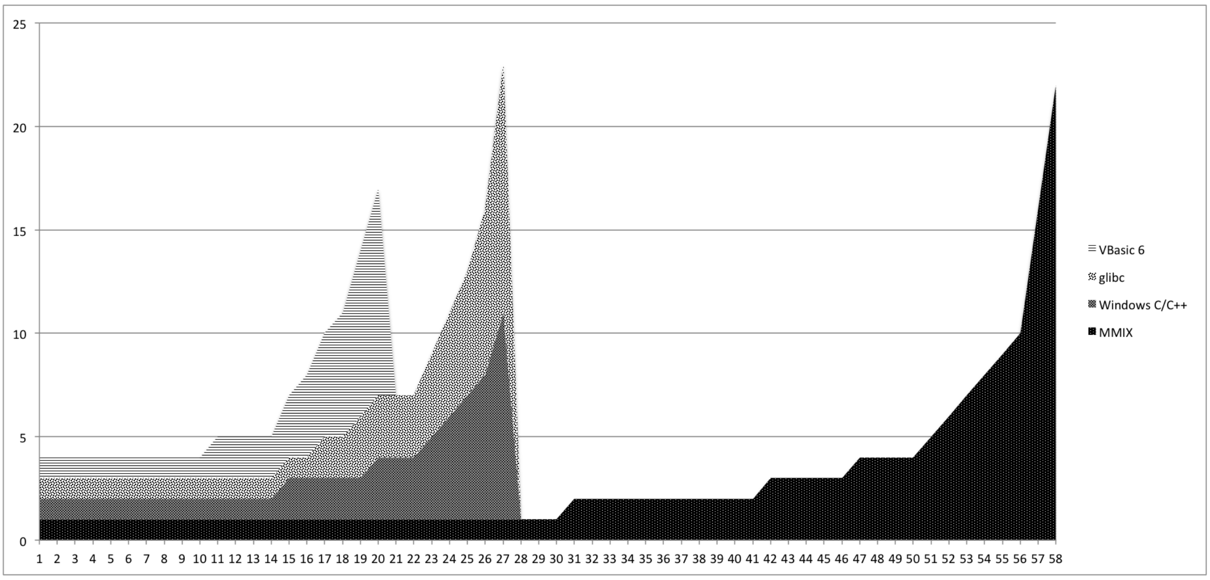

Figure 11: Solving LCGs with LLL; y-axis:number of leaks; x-axis: number of bits truncated.

In Figure 11 we show the relationship between the number of leaks required for recovering the state with the lattice-attack and the number of leaks that are truncated for four LCGs: the Windows LCG, the glibc LCG (which are both 32 bits), the Visual Basic LCG (which is 24 bits) and an LCG used in the MMIX of Knuth (which is 64 bits). It is seen that the number of leaks required is very small but increases sharply as more bits are truncated. In all cases the attack stops being useful once the number of truncated bits leaves none but the $\log w - 1$ most significant bits where w is the size of the LCG state. The logarithm barrier seems to be uniformly present and hints that the MSB's of a truncated LCG sequence may be hard to predict (at least using the techniques considered here). A similar logarithmic barrier was also found in the experimental analysis that was conducted by Contini and Shparlinski [2] when they were investigating Stern's attack [18] against truncated LCG's with secret parameters.

In Figure 12 (left) we present the results of applying the lattice-based framework against additive feedback generators similar to the one used in glibc. We demonstrate that there is a linear relation exhibited between the number of truncated bits and the number of leaks required. In the diagram, three "toy" generators with different depth of recursion (2,3 and 4) are targeted. The angle of the linear relation becomes sharper as the depth of recursion increases (for the glibc generator with its 32 depth the angle is much sharper - not shown here). In Figure 12(right) we present the running time of the algorithm for the glibc generator measured in seconds. Here the time complexity of LLL combined with the high dimension lattices required due to the 32 recursion depth of the glibc additive feedback generator take its toll in the running time. Our testing system (a 3.2GHz cpu with 2GB memory) ran out of memory when 7 bits were truncated. The version of LLL we employed (`SageMath 4.8`) has time complexity $O(k^5)$ where k is the dimension of the lattice (which represents roughly the number of leaks). The best time-complexity known is $O(k^3 \log k)$ derived from [12]; this may enable much higher truncation levels to be recovered for the glibc generator, however we were not able to test this experimentally as no implementation of this algorithm is publicly available.

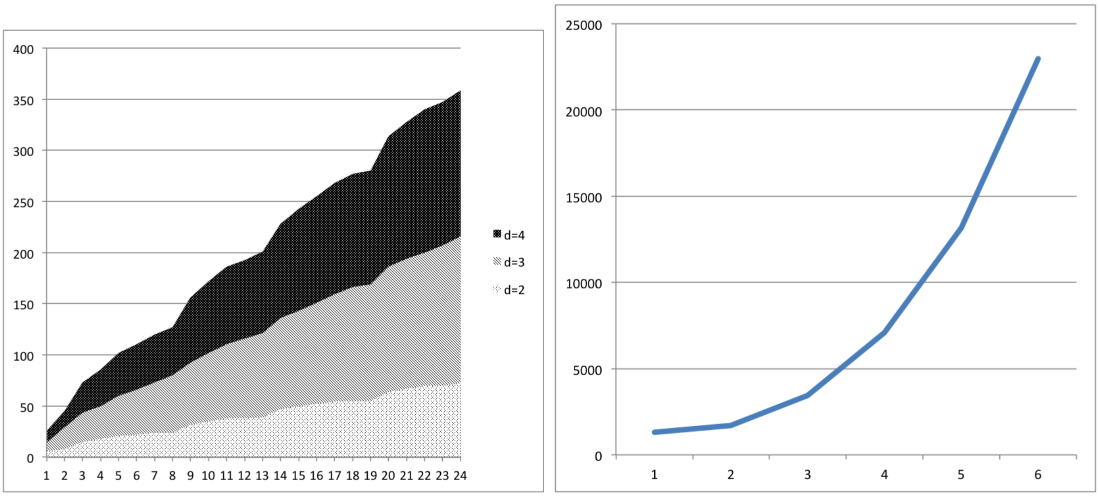

Figure 12: Solving additive feedback generators with LLL; y-axis: (left) number of leaks for "toy" generators, (right) time in seconds for glibc generator; x-axis: number of bits truncated.

We conclude that truncated LCG type of generators can be broken (in the sense of entirely recovering their internal state) for all but extremely high levels of truncation (e.g. in the case of 32-bit state LCG's modulo 2^{32} when they are truncated to 16 buckets or less). For additive feedback type of generators, such as the one in glibc, the situation is similar, however higher recursion depths require more leaks (with a linear relationship) that in turn affect the lattice dimension resulting in longer running times. Comparing the results between the LCGs and the additive feedback generators one may find some justification for the adoption of the latter in recent versions of glibc : it appears that - at least as far as lattice-based attacks are concerned - it is harder to predict truncated glibc sequences (compared to say, Windows LCG's) due to the higher running times of LLL reduction (note though that this does not mean that these are cryptographically secure).

6 Experimental results and Case studies

In order to evaluate the impact of our attacks on real applications we conducted an audit to the password reset function implementations of popular PHP applications. Figure 13 shows the results from our audit. In each case succesfully exploiting the application resulted in takeover of arbitrary user accounts [6] and in some cases, when the administrator interface was affected, of the entire application. In addition to identifiying these vulnerabilities we wrote sample exploits for some types of attack we presented, each on one affected application.

6.1 Selected Audit Results

Many applications we audited where trivially vulnerable to our attacks since they used the affected PRNG functions in a straightforward manner, thus making it pretty easy for an attacker to apply our techniques and exploit them. However some applications attempted to defend against randomness attacks by creating custom token generators. We will describe some attacks that resulted from using our framework against custom generators.

[6] The only exception to that is the HotCRP application where passwords were stored in cleartext thus there was no password reset functionality. However, in this case we were able to spoof registrations for arbitrary email accounts.

App / Attack	Time		Seed			State recovery	
Section	ATS	RT	4.1	4.2	4.3	5.3	5.4
mediawiki				●	●	●	
Open eClass				●	●		●
taskfreak				●	●	●	
zen-cart	●	●					
osCommerce 2.x	●	●					
osCommerce 3.x				●	●		●
elgg	●^{c}			●	●		
Gallery		●^{c}	●^{c}	●^{c}			
Joomla					●		
MyBB	○^{c}		○^{c}			○^{c}	
IP Board	●^{c}		●^{c}	●^{c}			
phorum				●	●	●	
HotCRP				●	●	●	
gazelle					●	●	
tikiWiki				●	●		●
SMF	○^{c}				○^{c}		

Figure 13: Summary of audit results. The c superscript denotes that the attack need to be used in combination with other attacks with the same superscript. The ● denotes a full attack while ○ denotes a weakness for which the practical exploitation is either unverified or requires very specific configurations.

Gallery. PHP Gallery is a very popular web based photo album organizer. In order for a user to reset his password he has to click to a link, which contains the security token. The function that generates the token is the following:

```
function hash($entropy="") {
  return md5($entropy . uniqid(mt_rand(), true));
}
```

The token is generated using three entropy sources, namely a time measurement from `uniqid()`, an output from the MT generator and an output from the `php_combined_lcg()` through the extra argument in the `uniqid()` function. In addition the output is passed through the MD5 hash function so its infeasible to recover the initial values given the output of this function. Since we do not have access to the output of the function, the state reconstruction attack seems an appropriate choice for attacking this token generation algorithm. Indeed, the Gallery application uses PHP sessions thus an attacker can use them to predict the `php_combined_lcg()` and `mt_rand()` outputs. In addition by utilizing the request twins technique from section 3 the attacker can further reduce the search space he has to cover to a few thousand requests.

MyBB. MyBB is a popular bulletin board application. In 2010, a bug was reported in the password reset functionality[5]. A patch was released shortly after that replaced the old generator with a new one that hides the output of MT from the user by xoring it with a random value and in addition it supplies a secure seed to the generator. The attack presented here works only in systems where `/dev/urandom` is unavailable and the suhosin extension is installed. This attack highlights the need for a universal secure PRNG that is OS independent and transparent to the developer.

The password reset token is an 8 character string produced by the custom PRNG of the application. To produce the string the application samples the characters randomly from an array which contains upper/lowercase letters and digits. The random number generator code follows:

```
function my_rand($min=null, $max=null,
                 $force_seed=false)
{
  static $seeded = false;
  static $obfuscator = 0;

  if($seeded == false || $force_seed == true)
  {
    mt_srand(secure_seed_rng());
    $seeded = true;
    $obf = abs((int) secure_seed_rng());
    [...]
  }
  [...]
  $distance = $max - $min;
  return $min+($distance+1)*
         (mt_rand()^ $obf)/(mt_getrandmax() + 1));
  [...]
}
```

```
function secure_seed_rng($count=8)
{
  $output = ’’;
  // Try the unix/linux method
  if(@is_readable(’/dev/urandom’) &&
  ($handle = @fopen(’/dev/urandom’, ’rb’)))
  {
    $output = @fread($handle, $count);
    @fclose($handle);
  }

  if(strlen($output) < $count)
  {
    $output = ’’;
    $unique_state = microtime().@getmypid();
    for($i = 0; $i < $count; $i += 16)
    {
      $unique_state = md5(microtime().$unique_state);
      $output .= pack(’H*’, md5($unique_state));
    }
  }

  $r=hexdec(substr(dechex(crc32(base64_encode($output))),
                0, $count));
  return $r;
}
```

In the case that `/dev/urandom` does not exist, or is inaccesible, the seed and obfuscator used are created using two time measurements and the process id of the process handling the request. In suhosin protected installations the seed is ignored by the PHP system, so we are interested only in the obfuscator. This random value is then xored with the output of the MT generator to hide its internal state. Even though the obfuscator does not have a very large entropy when xored with the output of MT gives a value which have a large enough entropy to be unpredictable for any practical purposes and indeed, if we just observe one output from this generator this seems to be the case. However, notice that the user obtains a token which is the result of 8 samples of that function, all using the same obfuscator. Thus, if we xor two such samples the obfuscator will be cancelled and we will be left with the result of xoring two outputs of the MT generator. This is enough to obtain the MT internal state, and subsequently bruteforce the obfuscator value.

Assume that $x_0, \dots, x_7$ are the 8 values returned in the security token to the attacker. For each of these values we have that $x_i = z_k \oplus obf$, where z_k is the $k - th$ output of the MT generator and obf is the obfuscator value which is common along the eight values returned. We define $r_i = x_i \oplus x_{i+1} = z_i \oplus z_{i+1}$. Notice that r_i is also a linear sequence like MT, so we can apply the same techniques from section 5.3 in order to recover its internal state. Thus, when we collect enough leaks we can recover the internal state of r_i and subsequently the MT internal state. All that is left now is to guess the obfuscator value which is used in the token generated for the target user. Notice that if the request that the attacker makes to reset the target user's password leaks back to the attacker even one output of my_rand() then the attacker, having access to the output of mt_rand() using the previous technique, can immediately recover the used obfuscator. This already presents a significant weakness of the generator. Nevertheless, to the best of our knowledge no such leak occurs in the generation of tokens. However, in systems where /dev/urandom is unavailable the attacker only needs to guess the process identifier along with two time measurements. The first one can be recovered through the session identifier using the techniques of section 4.1 while the time measurements can be approximated using the ATS algorithm.

Simple Machine Forums (SMF). Simple Machines Forums is another popular bulletin board application. In 2008 a bug was identified in the password reset function implementation as the application was leaking a rand() output through the token. In windows systems the state of rand() is 15 bits so it was fairly easy to use that output to predict the next token. However, that bug was exploitable even in Unix enviroments using our techniques. A patch was released afterwards that added an internal state variable and increased the overall entropy of the token. The SMF application is interesting because it utilizes many sources of entropy to generate the token including the PRNG of the MySQL database. All sources of entropy within the PHP system are predictable using our framework, however the database PRNG implementation details are not included within the MySQL documentation therefore, it is uncertain yet whether the application can be practically exploited. Nevertheless, we find interesting the approach in which an attacker can determine all other entropy sources. The code generating the security token is the following:

```
function generateValidationCode()
{
   [...]
   $t = sha1(microtime() . mt_rand() .
             $dbRand . $modSettings[’rand_seed’]);
   return substr(preg_replace(’/\W/’,
                              ’’, $t, 0, 10);
}
```

The \$*dbRand* variable is set by using the RAND() built-in function of MySQL while the rand_seed variable is the internal state variable. The value of rand_seed changes at random interval with the following code located in the main application file index.php

```
if (empty($modSettings['rand_seed'])
    || mt_rand(1, 250) == 69)
   smf_seed_generator();
```

where the smf_seed_generator() function is defined as follows:

```
function smf_seed_generator()
{
   global $modSettings;

   [...]

   $seed = ($modSettings['rand_seed'] +
           ((double) microtime() * 1000003)) & 0x7fffffff;
   mt_srand($seed);


   // Change the seed.
   updateSettings(array('rand_seed' => mt_rand()));
}
```

The application leaks outputs of mt_rand() in various places, either directly or in the form of an MD5 hash, thus a seed recovery attack is possible by spawning a new process and bruteforcing its seed. This allows to recover the mt_rand() part of the token. To recover the rand_seed one proceeds as follows: Since the interval in which the rand_seed is updated is determined from an output of mt_rand() the attacker can predict in which request the generator will update the seed and subsequently the state of the MT generator with the same value. Thus, the attacker can submit requests until the value that will update the seed will be generated. Afterwards, the attacker causes another mt_rand() leak and bruteforces the seed once again. This new seed that will be recovered is the value of the rand_seed variable. Since microtime() is easily bruteforcable using the time algorithms described all that is left to recover the security token is the *dbRand* value. The MySQL documentation does not mention any implementation details on the RAND() function, it mentions however that it is not cryptographically secure. Thus, we believe that is will be easy for an attacker with access to the same database, for example in a shared hosting enviroment, to recover the state of the database generator. For completely remote attacks, more investigation is needed to determine whether we can practically predict its output in the context of the SMF application.

Joomla. Joomla is one of the most popular CMS applications, and it also have a long history of weaknesses in its generation of password reset tokens[6, 11]. Until recently, the code for the random token generation was the following:

```
function genRandomPassword( $length=8 ) {
    $salt = abc...xyzABC...XYZ0123456789 ;
    $len = strlen ( $salt );
    $makepass = ‘‘’’;
-   $stat = @stat ( FILE ) ;
-   if (empty($stat) || !isarray($stat))
-      $stat=array(phpuname());
-   mt_srand(crc32(microtime().implode(|,$stat)));
    for($i=0;$i<$length;$i++){
       $makepass .= $salt[mt_rand(0,$len1)];
    }
    return $makepass;
}
```

In addition the output of this function is hashed using MD5 along a secret, 16 bytes, key (`config.secret`) which is created at installation using the function above. The `config.secret` value was also used to create a "remember me" cookie in the following way:

```
cookie = md5(config.secret+'JLOGIN REMEMBER')
```

Since the second part of the string is constant and the config.secret is generated through the genRandomPassword function which has only 2^{32} possible values for each length, one could bruteforce all possible values and recover `config.secret`. All that was left was the prediction of the output of the `genRandomPassword()` function in order to predict the security token used to reset a password. One then observes that although the contents of the `$stat` variable in the `genRandomPassword()` function are sufficiently random, the fact that `crc` is used to convert this value to a 4 byte seed allows one to predict the seed generated and thus the token. This attack was reported in 2010 in [11] and a year after, Joomla released a patch for this vulnerability which removed the custom seeding (dashed lines) from the token generation function. The idea was that because the generator is rolling constantly without reseeding one will be unable to recover the `config.secret` and thus the generator will be secure due to its secret state. Unfortunately, this may not be the case. If at the installation time the process handling the installation script is fresh, a fact quite probable if we consider dedicated servers that do not run other PHP applications, then the search space of the `config.secret` will be again 2^{32} and thus an attacker can use the same technique as before to recover it. After the `config.secret` is recovered, exploitation of the password reset implementation is straightforward using our seed recovery attack from section 4.3. A similar attack also holds when `mod_cgi` is used for script execution as each request will be handled by a fresh process again reducing the search space for `config.secret` in 2^{32} values.

However, the low entropy of the `config.secret` key is not the only problem of this implementation. Even if the key had enough entropy to be totally unpredictable, the generator would still be vulnerable. Notice that in case the `genRandomPassword()` is called with a newly initialized MT generator then there at most 2^{32} possible tokens, independently of the entropy of `config.secret`. This gives an interesting attack vector: We generate two tokens from a fresh process sequentially for a user account that we control. Then we start to connect to a fresh process and request a token for our account. If the token matches the token generated before then we can submit a second request for the target user's account which, since the first token matched the token we own, will match the second token that we requested before (recall that the tokens are not bound to users). Observe that if we generate only one pair of tokens this attack is expected to succeed after 2^{32} requests, assuming that the seed is random. Nevertheless, we can request more than one pair of tokens thus increasing our success probability. Specifically, if we have n pairs of tokens then at the second phase the attack is expected to succeed after $2^{32}/n$ requests. Therefore, if we denote by $r(n)$ the expected requests that the attack needs to hit a "good" token given n initial token pairs, then we have that $r(n) = 2n + 2^{32}/n$. Our goal is to minimize the function $r(n)$; this function obtains a positive minimum at $n = 2^{31/2}$, for which we have that $r(2^{31/2}) \approx 185000$. A simple bruteforcing framework that we wrote was able to achieve around 2500 requests per minute, a rate at which an attacker can compromise the application in a little more than one hour. To be fair, we have to add the requests that are required to spawn new processes but even if we go as far as to double the needed requests (and this is grossly overestimating) we still have a higly practical attack.

Gazelle. Gazelle is a torrent tracker application, which includes a frontend for building torrent sharing communities. It's been under active development for the last couple of years and its gaining increasing popularity. The interesting characteristic of the application's password reset implementation is that it uses two generators of the PHP system (namely `rand()` and `mt_rand()`. The code that generates a token is shown below:

```
function make_secret($Length = 32) {
   $Secret = '';
   $Chars='abcdefghijklmnopqrstuvwxyz0123456789';
   for($i=0; $i<$Length; $i++) {
      Rand = mt_rand(0, strlen($Chars)-1);
      $Secret .= substr($Chars, $Rand, 1);
   }
   return str_shuffle($Secret);
}
```

The code generates a random string using `mt_rand()` and then shuffles the string using the `str_shuffle()` function which internally uses the `rand()` function. If we try to apply directly the seed recovery attack, i.e. try to ask a question of the form "which seed produces this token" then we will run into problems because we have to take into account two seeds, and a total search space of 64 bits which is infeasible. The normal action would be to follow the same path as we did in the Gallery application where we had a similar problem and utilize the seed reconstruction attack which does not require an output of the PRNGs. However, the Gazelle application uses custom sessions (which are generated using the same function), and thus we cannot apply that attack either. The solution lies into slightly mofiying the seed recovery attack. Instead of asking the question "which seed produces this mt_rand() sequence", which is shuffled and thus affected by the second PRNG, we instead ask which seed produces the *unsorted* set which contains the characters of our string. This set is not affected by the shuffling and thus we can effectively bruteforce the `mt_rand()` seed independently. After recovering the `mt_rand()` seed we know the initial sequence that was produced and we can subsequently recover the seed of `rand()` using the same attack.

6.2 Attacks Implementation

In addition to auditing the applications, we implemented a number of our attacks targeting selected applications. In particular, we implemented a seed recovery attack against Mediawiki, a state reconstruction attack against the Phorum application and the request twins technique against Zen-cart. In the following sections we will briefly describe each vulnerability and the results of our attacks implementation.

Mediawiki. Mediawiki is a very popular wiki application used, among others, by Wikipedia. Mediawiki uses `mt_rand()` in order to generate a new password when the user requests a password reset. In order to predict the generated password we use the seed recovery attack of section 4.3. The function f that we sample is the one used to generate a CSRF token which is the following:

```
 function generateToken( $salt = '' ) {
   $token = dechex(mt_rand()).dechex(mt_rand());
   return  md5( $token . $salt );
 }
```

Our function f given a seed s first seeds the `mt_rand()` generator and then uses that generator to produce a token as the function above. To fully evaluate the practicality of the attack we implemented the attack online, without any time-space tradeoff. Our implementation was able to cover around 1300000 seed evaluations of f per second in a dual-core laptop with two 2.3 GHz processors. This allowed us to cover the full 2^{32} range in about 70 minutes. Of course, using a time-space tradeoff the search time could be further reduced to a few minutes.

Zen cart. Zen-Cart is a popular eCommerce application. At the time of this writing, a sample database which shops enter volunterily numbers about 2500 active e-shops [7]. In order to reset a user's password zen-cart first seeds the `mt_rand()` generator with the `microtime()` function and then uses the `mt_rand()` function to produce a new password for the user. Thus, there at most 10^6 possible passwords which could be produced. Our exploit used the request twins technique to reset both our password and the target user's password. Afterwards, we bruteforced the generated password for our account to recover the `microtime()` value that produced it. This takes at most a few seconds on any modern laptop. Then, our exploit bruteforces the passwords generated by `microtime()` values close to the one that generated our own new password. We ran our exploit in a network with RTT around 9 ms, and Zen-Cart was installed in a 4×2.3 GHz server. The average difference of the two passwords was about 3600 microseconds, and the exploit needed at most two times that requests since we don't know which password was produced first. With the rate of 2500 requests per minute that our implementation achieves, the attack is completed in a few minutes.

Phorum. Phorum is a classic bulletin board application. It was used, among others, by the eStream competition as an online discussion platform. In order for a user to reset his password the following function is used:

```
function phorum_gen_password($charpart=4, $numpart=3)
{
    $vowels = ... //[char array];
    $cons = ... //[char array];
    $num_vowels = count($vowels);
    $num_cons = count($cons);
    $password="";

    for($i = 0; $i < $charpart; $i++){
        $password .= $cons[mt_rand(0, $num_cons - 1)]
                  . $vowels[mt_rand(0, $num_vowels - 1)];
    }
    $password = substr($password, 0, $charpart);
    if($numpart){
        $max=(int)str_pad("", $numpart, "9");
        $min=(int)str_pad("1", $numpart, "0");
        $num=(string)mt_rand($min, $max);
    }
    return strtolower($password.$num);
}
```

What makes this function interesting in the context of state recovery is that at if called with no arguments (as it is in the application), at least four `mt_rand()` leaks are discarded in each call. We implemented the attack having the application installed in a Windows server with the Apache web server and we used our generic technique for Windows in order to reconnect to the same process. On average, the attack required around 1100 requests and 11 reconnections of our client. The running time was about 30 minutes, and the main source of overhead was the system solving. This fact is mainly explained from the small number of buckets and the lost leaks of each iteration. Neverthless, the attack remained highly practical, as we were able to compormise any user account (including the administrator) within half an hour.

[7] `http://www.zen-cart.com/index.php?main_page=showcase`

7 Defending Against Randomness Attacks

We believe that a major shortcoming of the PHP core is that it does not provide a native cryptographically secure PRNG and token generator. In fact, a pseudorandom function (PRF) would be the most suitable cryptographic primitive for generating random tokens based on program defined labels; PRF's can be constructed by PRNG's [7]. We feel that this is a shortcoming since developers tend to prefer functions from the core as they are compatible with every different enviroment PHP is running in. A possible solution would be to introduce a secure PRNG in the PHP core (as a new function). We proposed this solution to the PHP development team which informed us that the development overhead would be too big for supporting such a function and the solution of using `openssl_random_pseudo_bytes()` (which requires OpenSSL) is their recommendation.

On the other hand, administrators can take a number of precautions to defend against randomness attacks using current PHP versions. The Suhosin extension provides a secure seed in the `mt_rand()` and `rand()` functions. The seed exploits the fact that the Mersenne Twister has a large state and fills that state using a hash function. Because `rand()` may have a small state and is dependent from the operating system, the Suhosin extension replaces `rand()` with a Mersenne twister generator with a different state from `mt_rand()`. The hashed values of the seed used are a concatenation of predictable values such as process identifiers and timestamps, along with, potentially, unpredictable ones such as memory addresses of variables and input from `/dev/urandom`. Because the addresses in any modern operating system are randomized through ASLR, as a security precaution, using them as a seed should provide enough additional entropy to make the two seed attacks (sections 4.2, 4.3) infeasible (assuming ASLR addresses are unpredictable). In addition, the suhosin extension ignores the calls to the seeding functions `mt_srand(), srand()` in order to defend against weak seeding from the application. Although this may introduce a state recovery vulnerability, in the majority of our case studies, custom seeding was pretty weak and this measure (of securely seeding once and ignoring application based reseeding) increases security. We strongly believe that securely seeding the generators, when possible, is a very useful exploit mitigation for the attacks we presented. Although state recovery attacks would still be possible, these attacks are more complex than the seed attacks which require a handful of requests and commodity hardware to compromise the applications. Furthermore, creating a secure seed from such sources has a negligible performance overhead. Therefore, such measures should be employed by the PHP system as safeguards for applications that misuse the PHP core PRNGs.

Our session preimage attack (section 4.1) can be mitigated by utilizing an option (disabled by default) of PHP to add extra entropy, from a file, in the session identifier. By specifying `/dev/urandom` as the entropy file, a user can increase the entropy of a session arbitrarily thus making it infeasible for an attacker to obtain a preimage. In Windows, because `/dev/urandom` is not available this option gathers entropy using the same algorithm as in the `openssl_random_pseudo_bytes()` function. The PHP developement team informed us that the above option will be enabled by default in the upcoming version, PHP 5.4.

The above workarounds, if employed, will kill our seed attacks and the generic process distinguisher we devised. However, state recovery attacks would still be possible either through some application specific leak, or using the generic technique described for Windows operating systems (section 5.2). In addition, we find the possibility of the existence of other process distinguishers very probable; after all, the process identifier is not considered a cryptographic secret and could be leaked either through the application or the web server or even the operating system itself. Therefore, we feel that even using these workarounds, one should consider state recovery attacks practical.

With the present state of the PHP system, developers should avoid using directly the PRNGs of the PHP core for security purposes. Any application that requires a security token should employ a custom generator, that will either use the functions from the PHP extensions such as the `openssl_random_pseudo_bytes()`, if available, or it will use other entropy sources. We give an example of one such function in appendix D.

8 Related Work

The first randomness attack in PHP that we are aware of appeared in a blog post by Stefan Esser [3, 4], where he described basic system properties such as keep-alive connection handling by web server processes, and described how misusing `mt_srand()` could result in security vulnerabilities that he demonstrated in some popular applications. Shortly after, the same author released an update of the Suhosin extension which included the randomness features for strong seeding mentioned above. Our preimage attack on PHP sessions was insipired by an attack introduced by Samy Kamkar [10], in which he described some cases where an adversary would be able to guess a PHP session. However these attacks assumed a side-channel of server information. Finally Gregor Kopf [11] described, along other attacks, the vulnerability in the password reset implementation of Joomla. This work describes some type of seed recovery attacks but only for the case that a fresh seeding occurs within the PHP script executed.

9 Conclusions

We find the fact that the most popular programming language in a domain (cf. [19]) that has a clear need for cryptographically strong randomness does not have such a generator within its core system to be a security hazard. Still, even if such a generator existed in the language, the misuse of other functions would not disappear immediately as API misusage is a very common security problem in modern systems. Therefore, we believe that research in the practical exploitation of such insecure functions should be continued and extended to other environments even if they do offer better security features in their API than PHP. In this paper we explored the case of PHP installed in the Apache web server along with mod_php. We also showed the applicability of some of our attacks in cgi mode where each request is handled by a new process. However, the case of fast_cgi needs further investigation as its behavior depends highly on its configuration. In addition, it would be interesting to check other languages and web servers, such as PHP on an IIS web server, or Python and Ruby on Rails web applications in Apache. A problem that is also of theoretical interest is the development of faster algorithms for recovering truncated linear variables and finding an explanation for the logarithmic barrier we encountered when experimenting with the Håstad-Shamir framework. To conclude, despite the fact that linear generators are cryptographically insecure, the fact that developers misuse them for security critical features makes the analysis of their practical security within a certain application context an interesting research question which we believe needs further attention and awareness.

References

[1] Unknown Author. openssl_random_pseudo_bytes() painfully slow. PHP Bug 51636, `https://bugs.php.net/bug.php?id=51636`, 2010.

[2] Scott Contini and Igor Shparlinski. On stern's attack against secret truncated linear congruential generators. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP*, volume 3574 of *Lecture Notes in Computer Science*, pages 52–60. Springer, 2005.

[3] Stefan Esser. Lesser known security problems in php applications. In *Zend Conference*, 2008.

[4] Stefan Esser. mt_srand and not so random numbers. http://www.suspekt.org/2008/08/17/mt_srand-and-not-so-random-numbers/, 2008.

[5] Stefan Esser. Mybb password reset weak random numbers vulnerability. SektionEins GmbH, Security Advisory 2010/04/13, http://sektioneins.de/en/advisories/advisory-022010-mybb-password-reset-weak-random-numbers-vulnerability/, 2010.

[6] Stefan Esser. Joomla Weak Random Password Reset Token Vulnerability. SektionEins GmbH, Security Advisory 2008/09/11, http://www.sektioneins.de/advisories/SE-2008-04.txt, 2008.

[7] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

[8] Johan Håstad and Adi Shamir. The cryptographic security of truncated linearly related variables. In Robert Sedgewick, editor, *STOC*, pages 356–362. ACM, 1985.

[9] Robert ”Hackajar” Imhoff-Dousharm. Economics of password cracking in the gpu era. In *DEFCON 19*, 2011.

[10] Samy Kamkar. phpwn: Attacking sessions and pseudo-random numbers in php. In *Blackhat USA, Las Vegas, NV 2010*, 2010.

[11] Gregor Kopf. Non-obvious bugs by example. In *27th Chaos Communication Congress CCC*, 2010.

[12] Henrik Koy and Claus-Peter Schnorr. Segment lll-reduction of lattice bases. In Joseph H. Silverman, editor, *CaLC*, volume 2146 of *Lecture Notes in Computer Science*, pages 67–80. Springer, 2001.

[13] A.K. Lenstra, H.W.jun. Lenstra, and Lászlo Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.

[14] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Ron was wrong, whit is right. *IACR Cryptology ePrint Archive*, 2012:064, 2012.

[15] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[16] HD Moore and Valsmith. Tactical exploitation. In *DEFCON 15*, 2007.

[17] Matthew J. B. Robshaw and Olivier Billet, editors. *New Stream Cipher Designs - The eSTREAM Finalists*, volume 4986 of *Lecture Notes in Computer Science*. Springer, 2008.

[18] Jacques Stern. Secret linear congruential generators are not cryptographically secure. In *FOCS*, pages 421–426. IEEE Computer Society, 1987.

[19] W3 Techs. Usage of server-side programming languages for websites. W3 Web Technology Surveys, http://w3techs.com/technologies/overview/programming_language/all, 2012.

A Explicit Form of Tempering Matrix Equations

The tempering matrix T used by Mersenne Twister can be viewed as a set of linear equations. We use this representation to generate the linear equations in our state recovery algorithm. Below, we list the explicit form of these equations.

$z_0 = x_0 \oplus x_4 \oplus x_7 \oplus x_{15}$

$z_1 = x_1 \oplus x_5 \oplus x_{16}$

$z_2 = x_2 \oplus x_6 \oplus x_{13} \oplus x_{17} \oplus x_{24}$

$z_3 = x_3 \oplus x_{10}$

$z_4 = x_0 \oplus x_4 \oplus x_8 \oplus x_{11} \oplus x_{15} \oplus x_{19} \oplus x_{26}$

$z_5 = x_1 \oplus x_5 \oplus x_9 \oplus x_{12} \oplus x_{20}$

$z_6 = x_6 \oplus x_{10} \oplus x_{17} \oplus x_{21} \oplus x_{28}$

$z_7 = x_3 \oplus x_7 \oplus x_{11} \oplus x_{14} \oplus x_{18} \oplus x_{22} \oplus x_{29}$

$z_8 = x_8 \oplus x_{12} \oplus x_{23}$

$z_9 = x_9 \oplus x_{13} \oplus x_{20} \oplus x_{24} \oplus x_{31}$

$z_{10} = x_6 \oplus x_{10} \oplus x_{17}$

$z_{11} = x_0 \oplus x_{11}$

$z_{12} = x_1 \oplus x_8 \oplus x_{12} \oplus x_{19}$

$z_{13} = x_2 \oplus x_9 \oplus x_{13} \oplus x_{17} \oplus x_{20} \oplus x_{28}$

$z_{14} = x_3 \oplus x_{14} \oplus x_{18} \oplus x_{29}$

$z_{15} = x_4 \oplus x_{15}$

$z_{16} = x_5 \oplus x_{16}$

$z_{17} = x_6 \oplus x_{13} \oplus x_{17} \oplus x_{24}$

$z_{18} = x_0 \oplus x_4 \oplus x_{15} \oplus x_{18}$

$z_{19} = x_1 \oplus x_5 \oplus x_8 \oplus x_{15} \oplus x_{16} \oplus x_{19} \oplus x_{26}$

$z_{20} = x_2 \oplus x_6 \oplus x_9 \oplus x_{13} \oplus x_{17} \oplus x_{20} \oplus x_{24}$

$z_{21} = x_3 \oplus x_{17} \oplus x_{21} \oplus x_{28}$

$z_{22} = x_0 \oplus x_4 \oplus x_8 \oplus x_{15} \oplus x_{18} \oplus x_{19} \oplus x_{22} \oplus x_{26} \oplus x_{29}$

$z_{23} = x_1 \oplus x_5 \oplus x_9 \oplus x_{20} \oplus x_{23}$

$z_{24} = x_6 \oplus x_{10} \oplus x_{13} \oplus x_{17} \oplus x_{20} \oplus x_{21} \oplus x_{24} \oplus x_{28} \oplus x_{31}$

$z_{25} = x_3 \oplus x_7 \oplus x_{11} \oplus x_{18} \oplus x_{22} \oplus x_{25} \oplus x_{29}$

$z_{26} = x_8 \oplus x_{12} \oplus x_{15} \oplus x_{23} \oplus x_{26}$

$z_{27} = x_9 \oplus x_{13} \oplus x_{16} \oplus x_{20} \oplus x_{24} \oplus x_{27} \oplus x_{31}$

$z_{28} = x_6 \oplus x_{10} \oplus x_{28}$

$z_{29} = x_0 \oplus x_{11} \oplus x_{18} \oplus x_{29}$

$z_{30} = x_1 \oplus x_8 \oplus x_{12} \oplus x_{30}$

$z_{31} = x_2 \oplus x_9 \oplus x_{13} \oplus x_{17} \oplus x_{28} \oplus x_{31}$

B Online Gaussian Elimination

Algorithm 1 Online Gaussian Elimination

```
$Equations \leftarrow 0$
$\forall t, row[t].set \leftarrow false$
while $Equations < Variables$ do
  $eq \leftarrow getNextEquation()$
  while $eq.size > 0$ do
    $t \leftarrow eq.getFirstNonZeroTerm()$
    if $row[t].set$ then
      $eq \leftarrow eq \oplus row[t].eq$
    else
      $row[t].set \leftarrow true$
      $row[t].eq = eq$
      $Equations \leftarrow Equations + 1$
    end if
  end while
end while
```

Algorithm 1 gives the pseudocode of our online Gaussian elimination algorithm. Some explanation on the notation of the algorithm follows:

- The $getNextEquation()$ function fetches the next equation to be added to the system.
- The $getFirstNonZeroTerm()$ function returns the position of the first non zero term in the equation in ascending order.
- The *row* variable is an array which holds the equations as we add them to the system.
- The *eq.size* variable describes the number of non zero terms in the equation.

The algorithm works exactly as the ordinary Gaussian elimination with one difference: Instead of switching two columns to bring the non-zero term in the diagonal as we would do with the normal Gaussian elimination we place the current equation in a row where the non zero term it has falls in the diagonal. If that row is occupied by another equation then we proceed normally by XORing the two equations and we check the next non zero term. If the length of the equation reaches zero before we add it in the system then we conclude that it is linear dependent to the other equations already present in the system thus we skip it. This enables to complete the elimination while avoiding column operations altogether. This is particularly suitable for our setting where the linear system is built gradually as we obtain leaks from the generator function under attack.

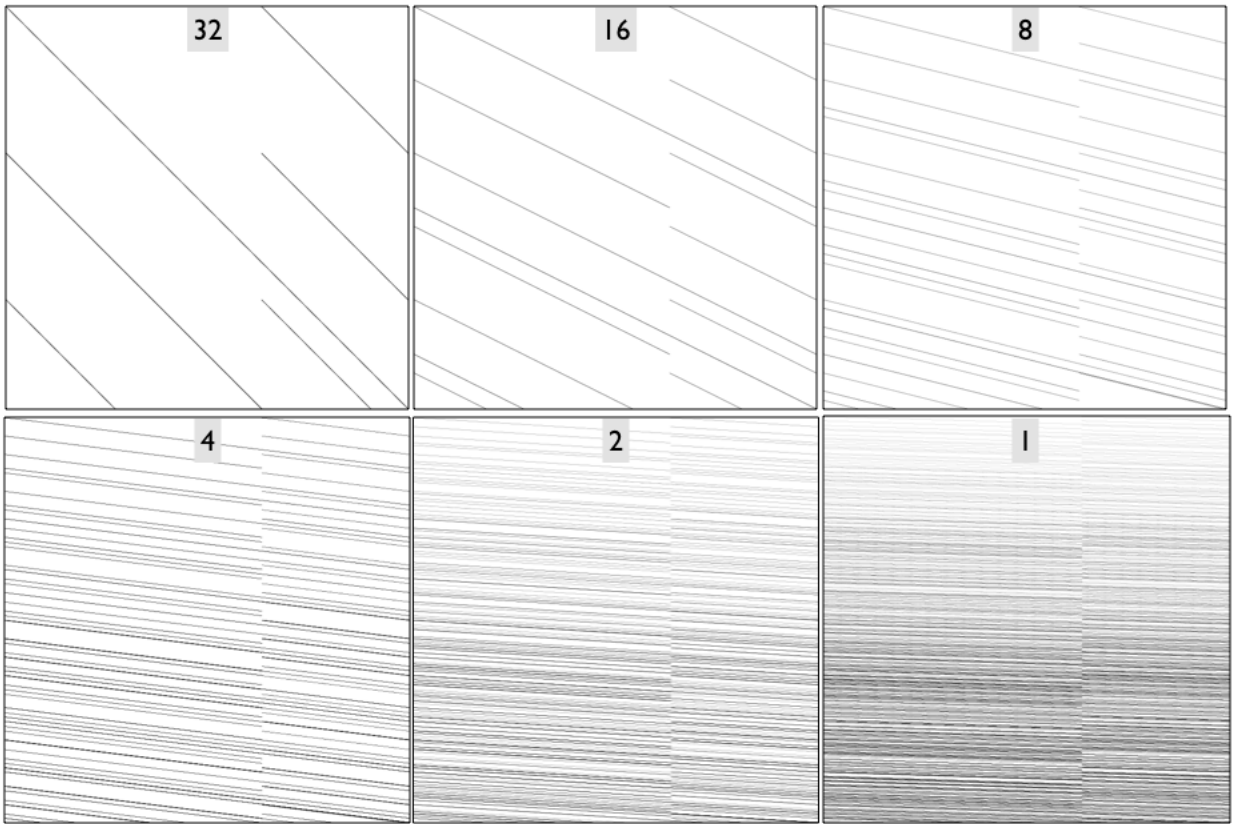

Figure 14: A visual representation of the system of linear equations produced by the MT algorithm for the cases of $32, 16, 8, 4, 2, 1$ bit leaks.

C A visualization of the MT system

To make the difficulties in solving the linear system of MT clearer, we created in Figure 14 a visualization of the system of linear equations created by unrolling the equations of Mersenne twister. The black pixels in the images represent variables with coefficient one, while the white pixels represent a coefficient zero. Notice that the more bits we truncate from each word the denser system one is called to solve. This justifies the increase in the time required to solve the system, since our implementation uses a sparse representation of each equation and thus tends to handle more poorly dense systems. Each picture is of size 19937×19937 pixels.

D A secure PHP token generation function

```
function secure_random_string($len = 20)
{
   // if a secure randomness generator exists and we don’t have a buggy PHP version use it.
   if (function_exists(’openssl_random_pseudo_bytes’) &&
       (version_compare(PHP_VERSION, ’5.3.4’) >= 0 || substr(PHP_OS, 0, 3) !== ’WIN’))
   {
      $str = bin2hex(openssl_random_pseudo_bytes(($len/2)+1, $strong));
      if ($strong == true)
         return substr($str, 0, $len);
   }
   //collect any entropy available in the system along with a number
   //of time measurements or operating system randomness.
   $str = ’’;
   $bits_per_round = 2;
   $msec_per_round = 400;
   $hash_len = 20; // SHA-1 Hash length
   $total = ceil($len/2); // total bytes of entropy to collect

   do
   {
      $bytes = ($total > $hash_len)? $hash_len : $total;
      $total -= $bytes;

      //collect any entropy available from the PHP system and filesystem
      $entropy = rand() . uniqid(mt_rand(), true);
      $entropy .= implode(’’, @fstat(fopen( __FILE__, ’r’)));
      $entropy .= memory_get_usage();
      if(@is_readable(’/dev/urandom’) && ($handle = @fopen(’/dev/urandom’, ’rb’)))
      {
         $entropy .= @fread($handle, $bytes);
         @fclose($handle);
      }
      else
      {
         // Measure the time that the operations will take on average
         for ($i = 0; $i < 3; $i ++)
         {
            $c1 = microtime() * 1000000;
            $var = sha1(mt_rand());
            for ($j = 0; $j < 50; $j++)
            {
               $var = sha1($var);
            }
            $c2 = microtime() * 1000000;
         $entropy .= $c1 . $c2;
         }
         if ($c1 > $c2) $c2 += 1000000;

         // Based on the above measurement determine the total rounds
         // in order to bound the total running time.
         $rounds = (int)(($msec_per_round / ($c2-$c1))*50);

         // Take the additional measurements. On average we can expect
         // at least $bits_per_round bits of entropy from each measurement.
         $iter = $bytes*(int)(ceil(8 / $bits_per_round));
         for ($i = 0; $i < $iter; $i ++)
```

```
            {
               $c1 = microtime();
               $var = sha1(mt_rand());
               for ($j = 0; $j < $rounds; $j++)
               {
                  $var = sha1($var);
               }
               $c2 = microtime();
               $entropy .= $c1 . $c2;
            }

         }
         // We assume sha1 is a deterministic extractor for the $entropy variable.
         $str .= sha1($entropy);
   } while ($len > strlen($str));

   return substr($str, 0, $len);
}
```

The implementation above provides a function that can be used as a drop-in replacement for any token generator. Since it does not keep any internal state it is easy to replace any function with it. It is secure under the assumption that SHA-1 is a deterministic extractor for the `$entropy` variable as defined in the source code, and that the `$entropy` variable itself has sufficient entropy. In summary, the function works as follows: If the `openssl_random_pseudo_bytes()` is available and the PHP version is not affected by the bug we mentioned before, then we use this function to get a random string. If the function is not available then we use all the PRNGs available in the system. Although these are predictable as this paper demonstrated, these calls are fast and may increase the entropy collected by a few bits. Afterwards, we add the `stat` structure of the file in which the source code is located as an entropy source. The `stat` structure has some bits of entropy since it contains some values that are difficult to predict including the inode of the file and the file creation and last modification timestamps. Then, the memory usage of the current script is added. This value varies from system to system because of differences in the system configuration, memory allocators and others. Although, these sources may provide some bits of entropy we cannot rely solely on them. Ideally, even if the attacker is running the function he should not be able to predict the result even given control of the environment. Therefore, we then try to utilize the operating system generator of Unix systems; if the `/dev/urandom` generator is available then we collect additional entropy from there. Otherwise, since we are left we no entropy source, we make a number of computationally expensive operations and measure the time they take to complete. Specifically, we calculate the SHA-1 of a random number for a number of times and measure the time this operation takes to complete. In order to keep a balance between running time and security, we set the total running time to be around 400 microseconds for each round. We conducted experiments in systems with different hardware and configurations which suggest that in each round one could extract two bits of entropy. Therefore, the running time for n bits of entropy would be $n/2 \times 400$. For the default entropy length (80 bits), the function is executed in about 15 miliseconds, which is quite acceptable for the desired level of security. Finally, we use SHA-1 to extract the entropy from the `$entropy` variable and return a substring of that hash depending on the user requested length.

To produce a secure token it is prudent that applications use also user-specific data combined with the token to avoid attacks such as the one we demonstrated for Joomla (cf. section 6). Thus, the final token could look like this:

```
$token = sha1(secure_random_string() . $old_user_password_hash);
```

where the `$old_user_password_hash` variable denotes the password hash for the user that requested the password reset.