

Tools of the Trade

- Nobody likes to write exploits in C
- A lot of tools with a python interface in order to exploit vulnerabilities attacks
 - ChainLadder.com
 - Exploit-execute (for exploit generation)
 - Framework for tool-level scripting tool
- Check http://exploit-db.com for the website

Randomness Attacks Mitigation

- PHP 5.4 added new strategy to the random generation
 - random strategy (single random by default)
 - Suggested to all users switching to all PRNGs in the PHP core
 - PHP security team: "This is an application specific problem"
 - Secure PRNGs from operations are rarely and tight now
- A deep to replace them the very robust generation can be found in
 - Java: cryptorandom
 - Check for cryptorandom PRNGs in the PHP system
 - Otherwise utilize strategy from various sources

Related work

Stefan Esser

- mt_rand and not so random numbers.

Samy Kamkar

- How I met your girlfriend.

Gregor Kopf

- Non obvious bugs by example.

Summary

- Randomness attacks affect a very large number of PHP applications.
- Exploit mitigations are needed for these attacks.
- Crypto bugs are becoming a trend for exploitation.

Thank You!

Questions?

PRNG: PseudoRandom number Generators (in PHP applications)

George Argyros Aggelos Kiayias

PRNG:

Pwning Random Number Generators

(In  Applications)

George Argyros*

Aggelos Kiayias

University Of Athens

*Census Inc.

A Small Challenge

```
<?php
    echo mt_rand();

    $bet = $_GET['bet'];
    if ($bet == mt_rand()) {
        system($_GET['cmd']);
    }
?>
```

Would you put this
code on your server?



code on your server:

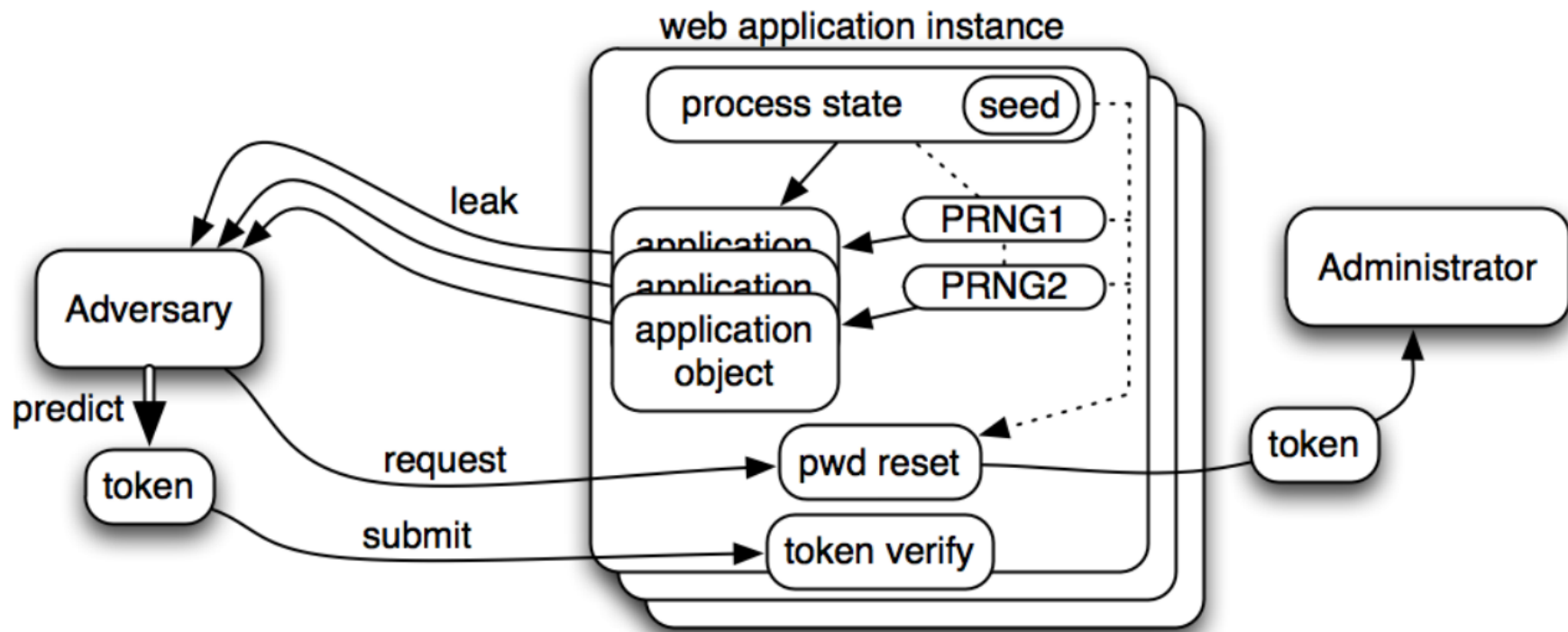


Can you exploit this code
on a target server?

Goal:

Provide practical techniques for the exploitation of randomness vulnerabilities in PHP applications

Attack Template



Entropy Generation in PHP

PHP Core

`mt_rand()/mt_srand():`

- PRNG based on the Mersenne Twister generator.
- `mt_srand()` seeds the generator with a 32 bit value
- `mt_rand()` produces a 32 bit but PHP truncates the LSB.
- Can return a number in a smaller range.

rand()/srand()

- Uses the rand() function from the OS
- seeded with a 32 bit value and produces 31 bit outputs.
- Can return a number in a smaller range.

php_combined_lcg()/lcg_value()

- php_combined_lcg() is used internally by the PHP system.
- lcg_value is its public interface.
- Combines two 32 bit LCGs and produces a 64 bit output.
- Seeded only once the first time it is called.

uniqid(prefix, extra_entropy)

- When called without arguments produces a timestamp with seconds and microseconds since unix epoch.
- The first argument adds a user supplied prefix to the timestamp.
- The second argument adds an output of `php_combined_lcg()` as suffix.

PHP extensions

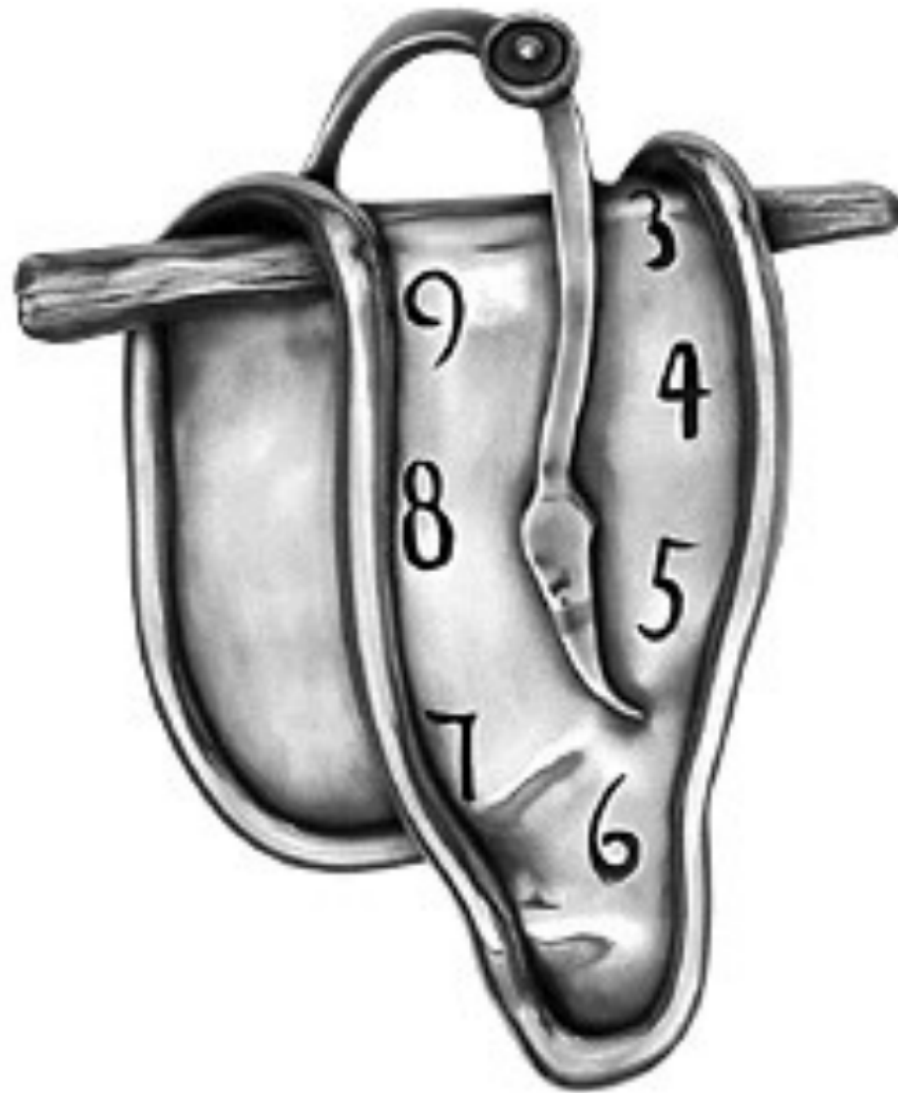
openssl_random_pseudo_bytes()

- Interface to the openssl function with the same name.
- Returns a number of pseudorandom bytes.
- A flag is used to tell if the bytes are crypto strong.
- Requires openssl extension.

mcrypt_create_iv()

- Gathers entropy from the operating system generators
 - /dev/random, /dev/urandom
- Requires mcrypt extension.

The Entropy Of Time Measurements



Timestamps make really good PRNG seeds, provided you're willing to kill everyone who owns a clock.

-- Matthew Green

Timestamps Facts:

- Epoch (time up to seconds accuracy) is leaked to the client

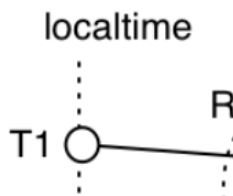
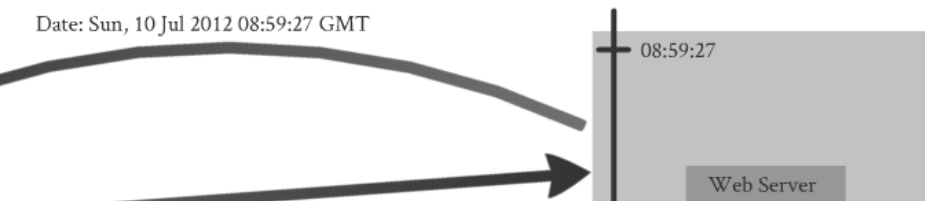
Timestamps Facts:

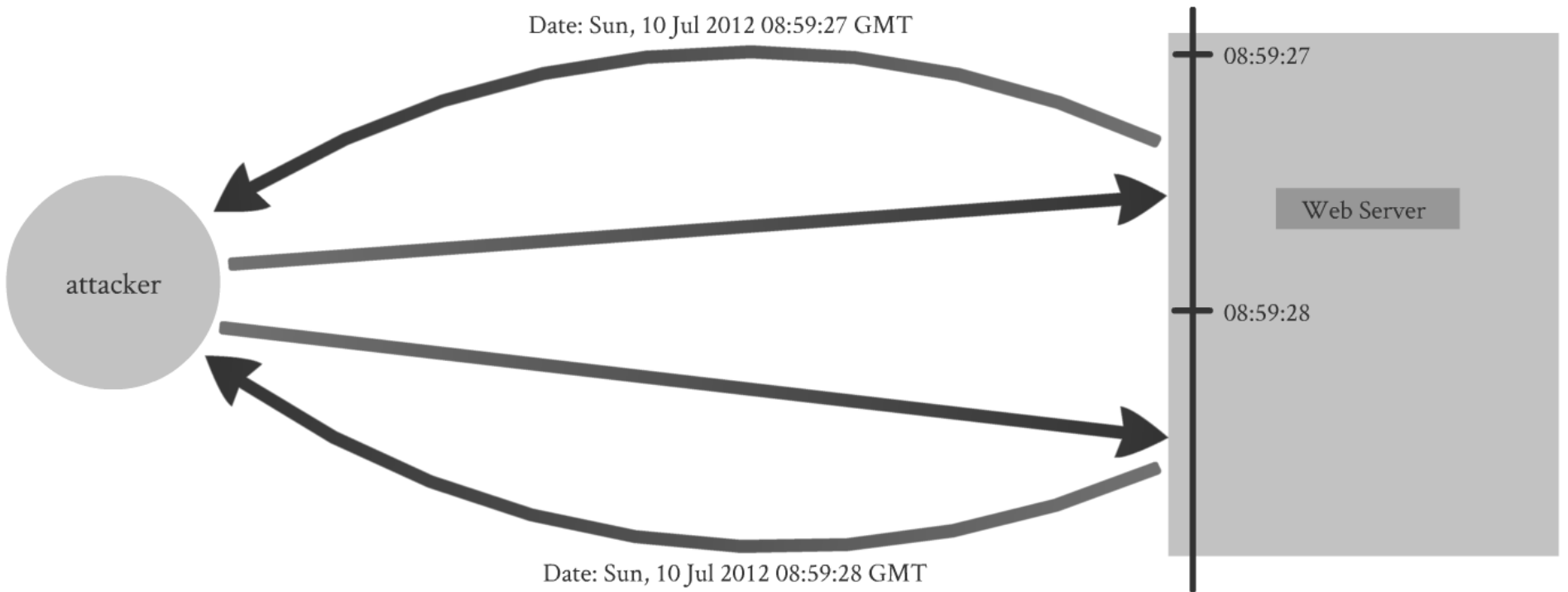
- Epoch (time up to seconds accuracy) is leaked to the client through the HTTP Data Header.
- Microseconds range from 0 to 10^6
 - Therefore a trivial bruteforce will succeed after 500k requests on average.

Can we do better than 500k ?

- Adversial Time Synchronization
- Request Twins

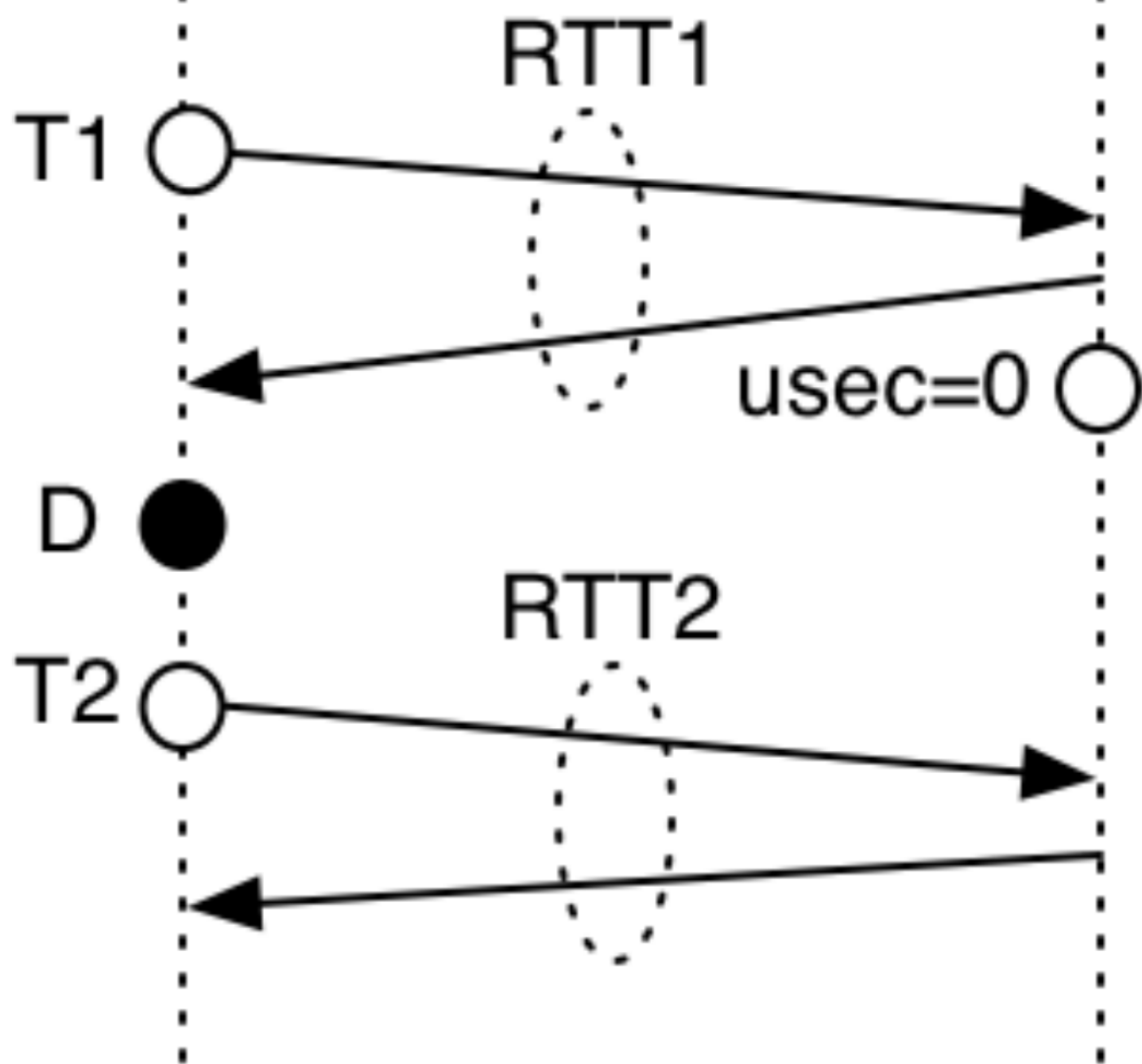
Adversarial Time Synchronization (ATS)



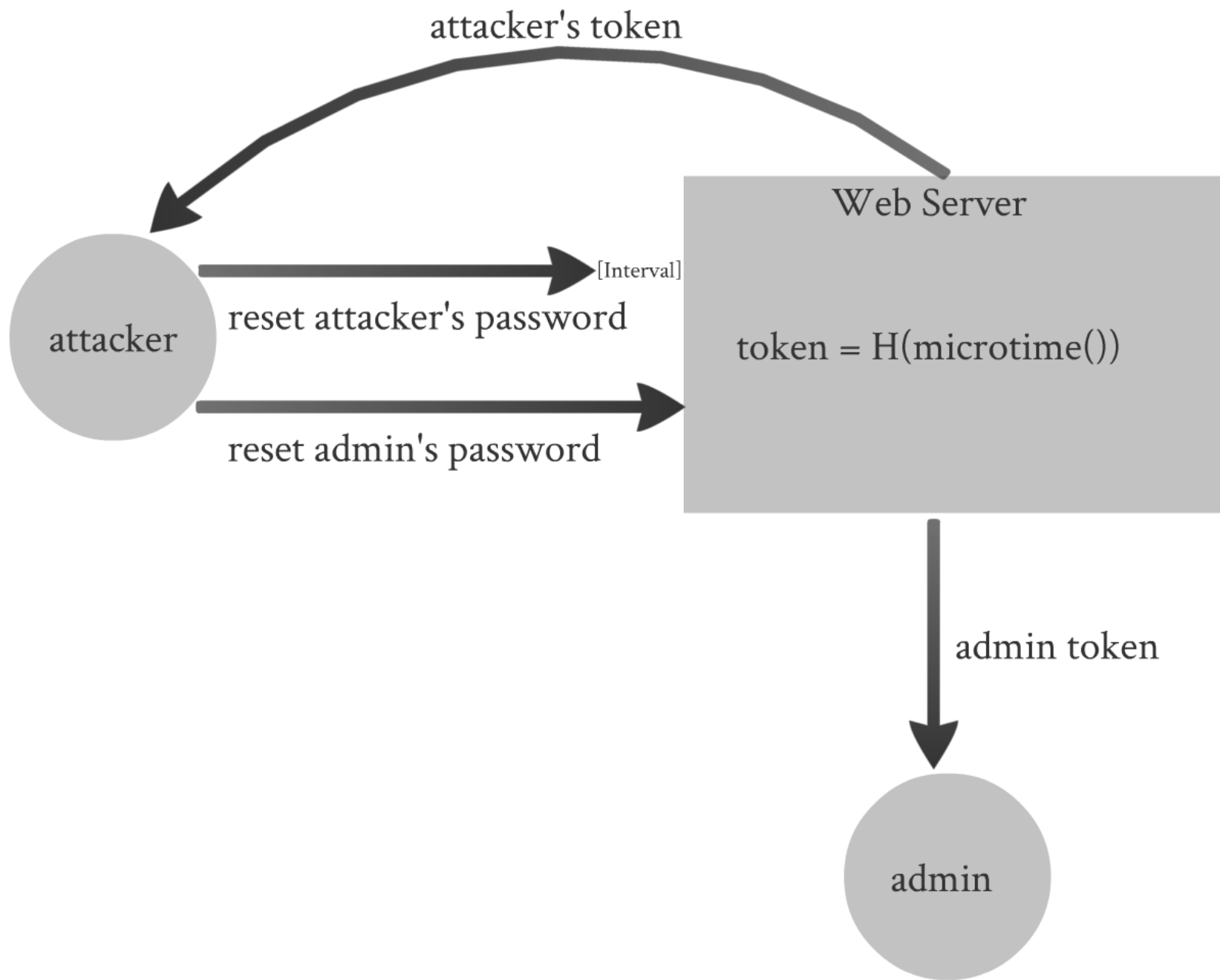


localtime

servertime



Request Twins



Evaluation

Experiment

Predict the output of the following script:

```
<?php  
    echo microtime();  
?>
```

Results

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

Time is in microseconds

Case study



zen**cart**

The art of e-commerce

- Token generation:
 - seeds `mt_rand` with `microtime()`.
 - Produces token using `mt_rand()`.
- Configuration:
 - 2 cores system
 - RTT ≈ 10 ms

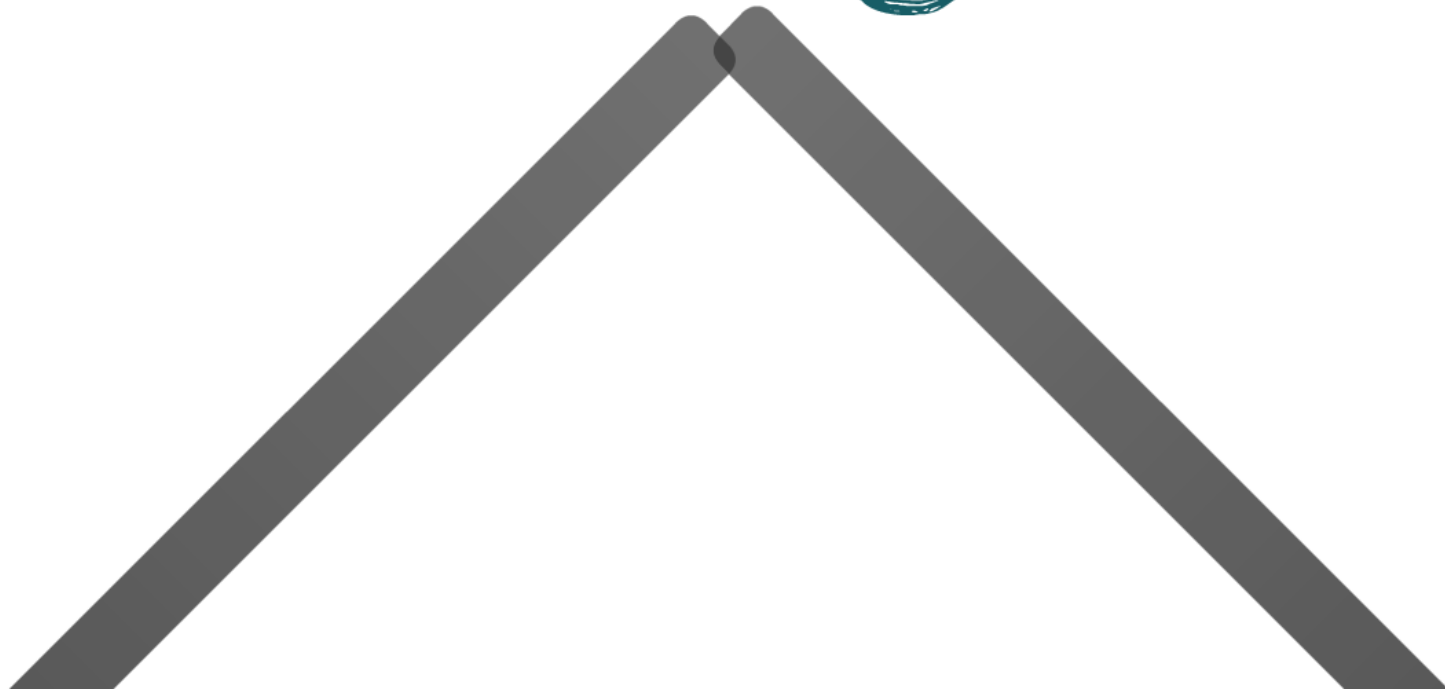
Results:

On average 7k requests to
compromise the application

bonus:


code execution on the server!

Attacking a PRNG





Seed Attacks

A large, solid green arch shape that spans the width of the slide, positioned at the top. It has a smooth, rounded top and a slightly irregular bottom edge.

In order to attack the seed we need the ability to interact with newly seeded generators. This usually happens when a fresh process is created.

Process management

We will focus here!



Apache handler (mod_php):

- PHP runs as an Apache web server module.

CGI:

- There is a new PHP process spawned for each request and terminated after the request is served.

Fast CGI:

- There are a number of PHP processes serving requests repeatedly. They are usually killed after they served a predefined number of requests.

Keep-Alive Requests

- When the Connection HTTP header is set to Keep-Alive the web server keep the connection open.
- There is a maximum number of keep-alive requests.
- In mod_php all requests within the same connection are handled by the same process.
 - Multiple requests to the same PHP process.

Generating fresh PHP processes

- In mod_php when the number of occupied processes reaches a certain threshold the server creates new processes to handle subsequent requests.
 - The default threshold in Apache is to have less than 5 idle processes.
- We can exploit this functionality to force the creation of new PHP processes!

Technique

- Create a large number of connections using the Keep-Alive HTTP header.
- While keeping these connections alive make a new connection to the server.
- The new connection is very likely to be handled by a fresh PHP process.

Hacking your own PHP session identifier



```
session_id = MD5(client IP address || time_of_day() || php_combined_lcg())
```

- If the total entropy is "small" then we can obtain a preimage by bruteforce.
- This gives us the value of `php_combined_lcg()`.

But what does small means?

- Since we have access to the MD5 sum we can perform the bruteforcing on the CPU rather over the network.
 - 250\$ GPU --> $2^{\{30\}}$ MD5 / sec.
 - 750\$ GPU --> $2^{\{32\}}$ MD5 /sec.
- Entropies up to 40 bits are easily handled...

```
session_id = MD5(client IP address || time_of_day() || php_combined_lcg())
```



Known since its the
attacker's IP address.



Provides up to 20 bits of entropy which
can be reduced using the ATS algorithm.

► php_combined_lcg has a 64 bits output so bruteforcing the output is not feasible.

Since we can generate fresh processes we can try to predict the first output which is simply one round of the generator with the seed.

php_combined_lcg has two 32 bit registers s_1, s_2 .

Seeding:

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

- T_1, T_2 : two subsequent timestamps
- pid : PHP process identifier.

0 bits



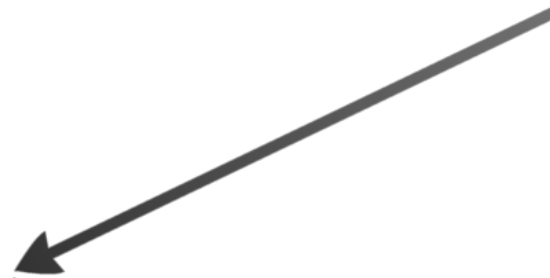
< 20 bits



15 bits



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



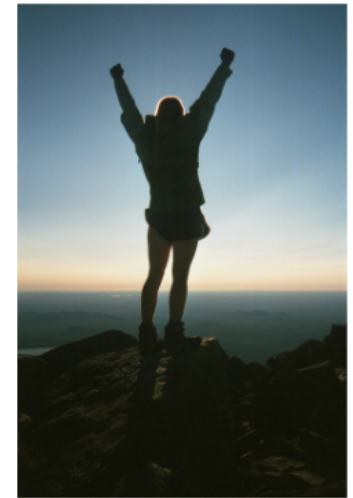
Calculated as $\Delta = T_2 - T_1$.

Entropy = ~ 3 bits.

Total entropy of session identifier
in a fresh process is about 40 bits.

By bruteforcing the session identifier of a fresh process we can obtain:

- the seed of `php_combined_lcg`
- the process identifier of the PHP process.





what about the other PRNGs?

`/mt_rand()`

Seeding in rand()/mt_rand()

- These generators can be seeded with the respective functions mt_srand() and srand().
- If the generator is not seeded, then the following 32 bit seed is produced:

```
seed = (time()*pid) ^ (10{6}*php_combined_lcg())
```



Assume we have a preimage for a session identifier.

Leaked by the Date
HTTP Header

Obtained through the
session id preimage

$$\text{seed} = (\text{time()} * \text{pid}) \wedge (10^{\{6\}} * \text{php_combined_lcg}())$$

Obtained through the
session id preimage

A session identifier preimage completely determines the seed of the `mt_rand()` and `rand()` PRNGs!

The attack does not require any outputs from the targetted PRNGs!

Case study



```
$user->hash = random::hash()
```

```
$user->hash = random().hash()
```

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

Attack:

- Obtain a preimage for a session identifier
 - This will give uniqid's extra entropy and mt_rand
- Use request twins to bruteforce uniqid's timestamp.



The bruteforce idea of `session_id` also applies on the seed of `rand()` and `mt_rand()`.

Assume we are connected to a fresh process and obtain some outputs are dependent on `rand()/mt_rand()`.

Then we can do an offline bruteforcing of all $2^{\{32\}}$ possible seeds to find which one generated the observed outputs.

Contrary to the previous attacks that relied on information obtained **online** this attack relies only on the small size of the seed.

Therefore the process can be further optimized using an application specific rainbow table.

Case study



Joomla!™

Password reset algorithm in Joomla

- 2008: mt_rand() seeded with microtime().
- 2010: mt_rand() seeded with the crc32 of an unpredictable string along with an installation time generated key produced the same way.
- 2011: Seeding removed, and default PHP seeding was use along with the secret key.

```
$token = JApplication::getHash(JUserHelper::genRandomPassword());
```

```
$registry->set('secret', JUserHelper::genRandomPassword(16));
```

```
public static function genRandomPassword($length = 8)
{
    $salt = "abcdef[...]QRSTUVWXYZ0123456789";
    $len = strlen($salt);
    $makepass = "";

    for ($i = 0; $i < $length; $i++)
    {
        $makepass .= $salt[mt_rand(0, $len - 1)];
    }

    return $makepass;
}
```

```
function getHash($seed)
{
    return md5(JFactory::getConfig()->get('secret') . $seed);
}
```


Notice:

If the configuration script was executed on a fresh PHP process then the entropy of the secret key is 32 bits regardless of its length!

secret key is used for "remember me" cookies.

```
setcookie(self::getHash('JLOGIN_REMEMBER'), $rcookie, ... );
```

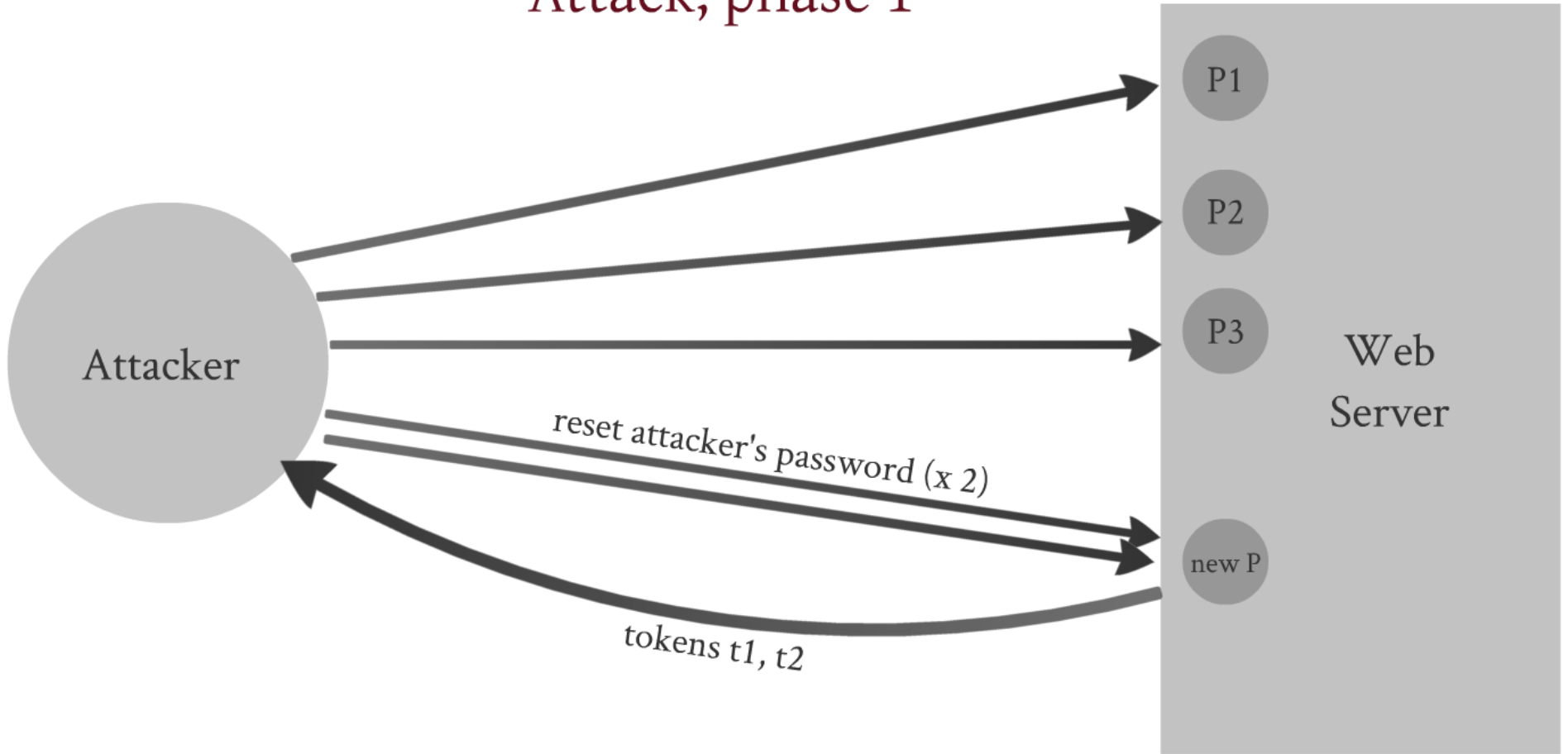
Vulnerability no.1

- Get a "remember me" cookie and bruteforce all $2^{\{32\}}$ possible secret keys.
- Once the secret key is recovered use one of the previously mentioned attacks to predict future generated tokens.

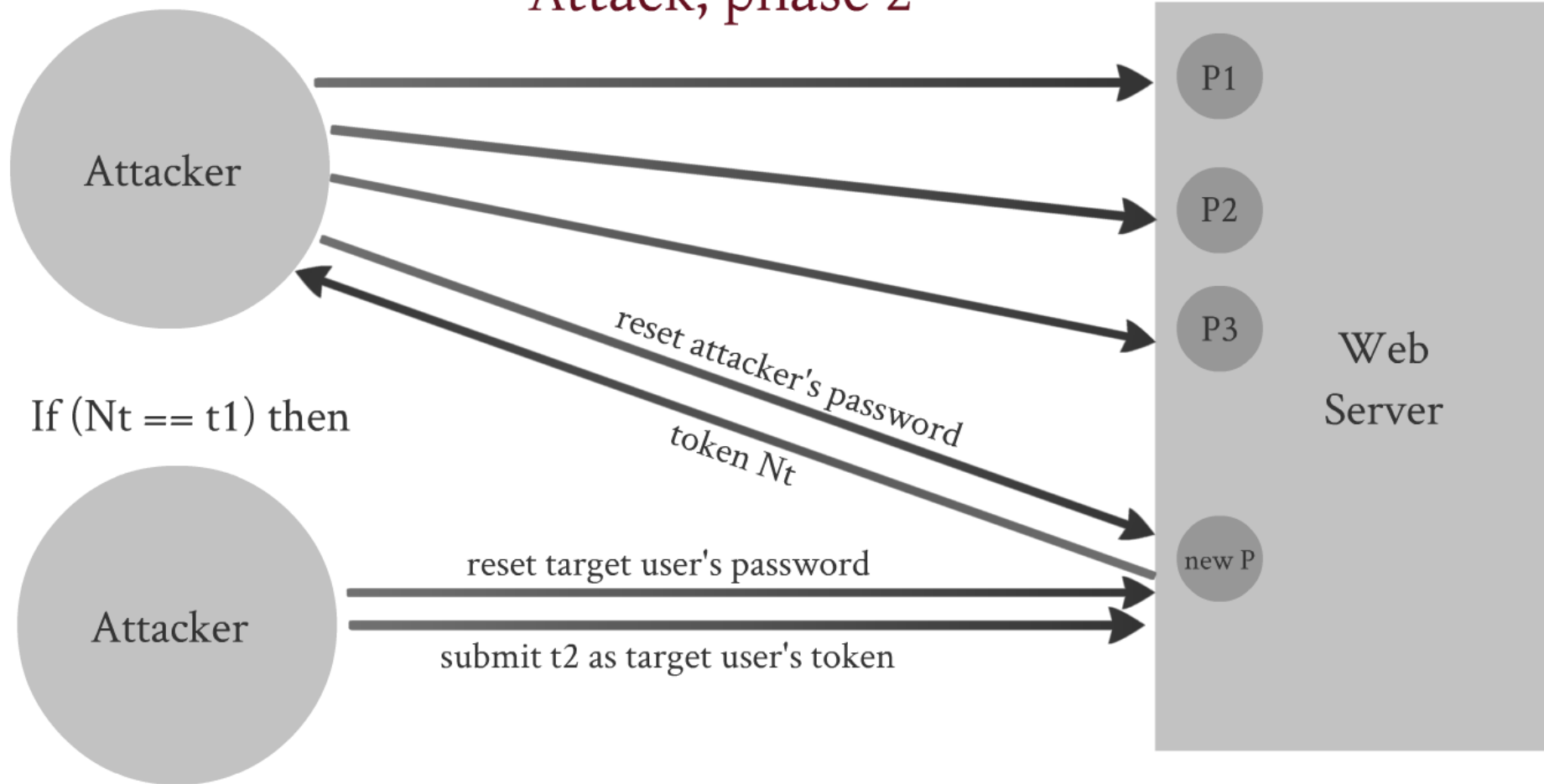
The attack works only when the secret key is generated from a fresh process.

But let's assume that the key is totally random.

Attack, phase 1



Attack, phase 2



Expected Number of requests: 2^{32} :- (

We can request more than one token
in the first phase of the attack.

1 token $\rightarrow 2^{\{32\}}$ requests

2 tokens $\rightarrow 2^{\{32\}}/2$ requests

k tokens $\rightarrow 2^{\{32\}}/k$ requests

Total number of requests as a function of the token pairs we will request:

$$F(x) = 2x + 2^{\{32\}} / x$$



2 requests for
2 tokens

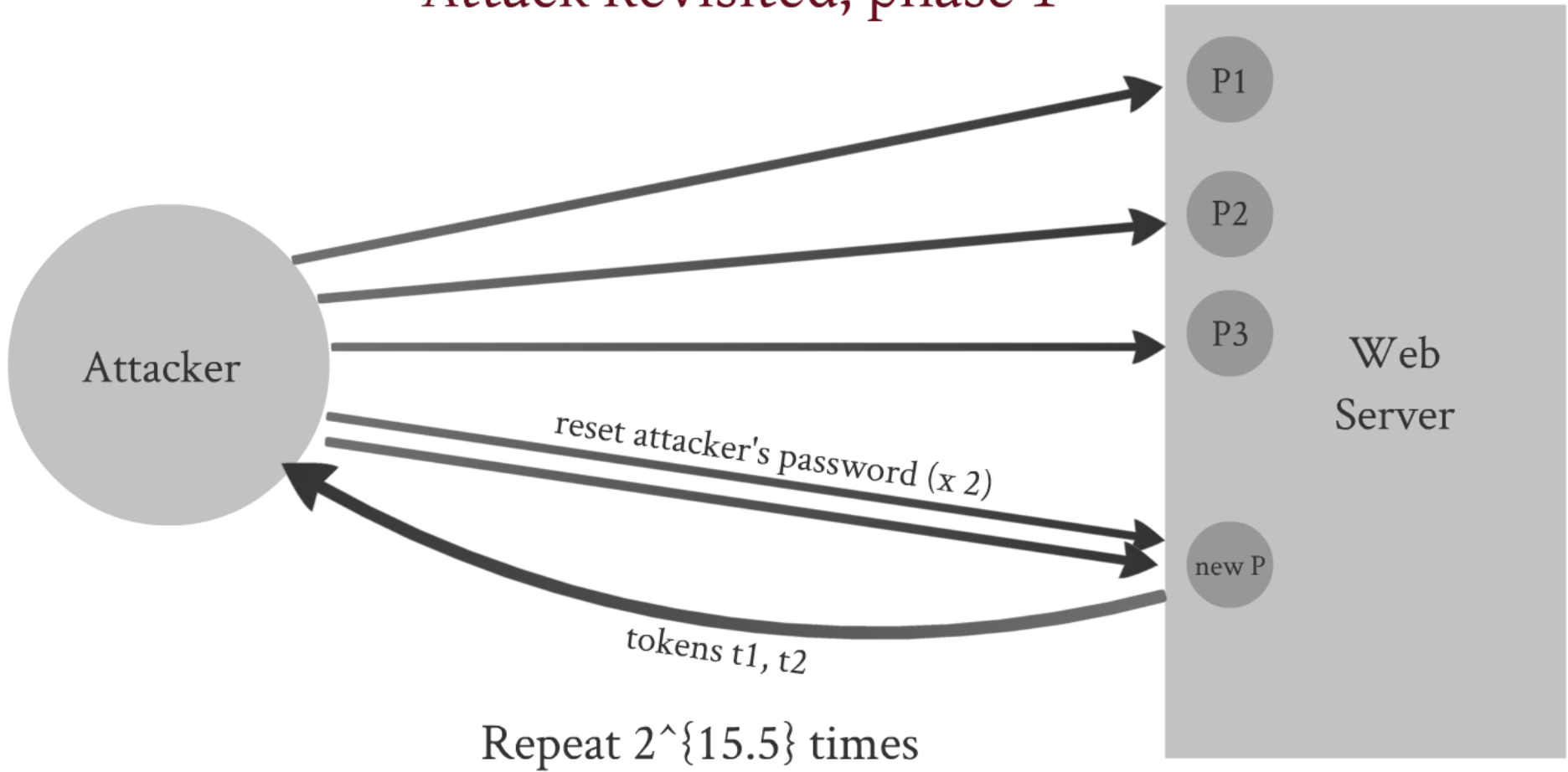


number of requests to hit one
of the tokens we have obtained.

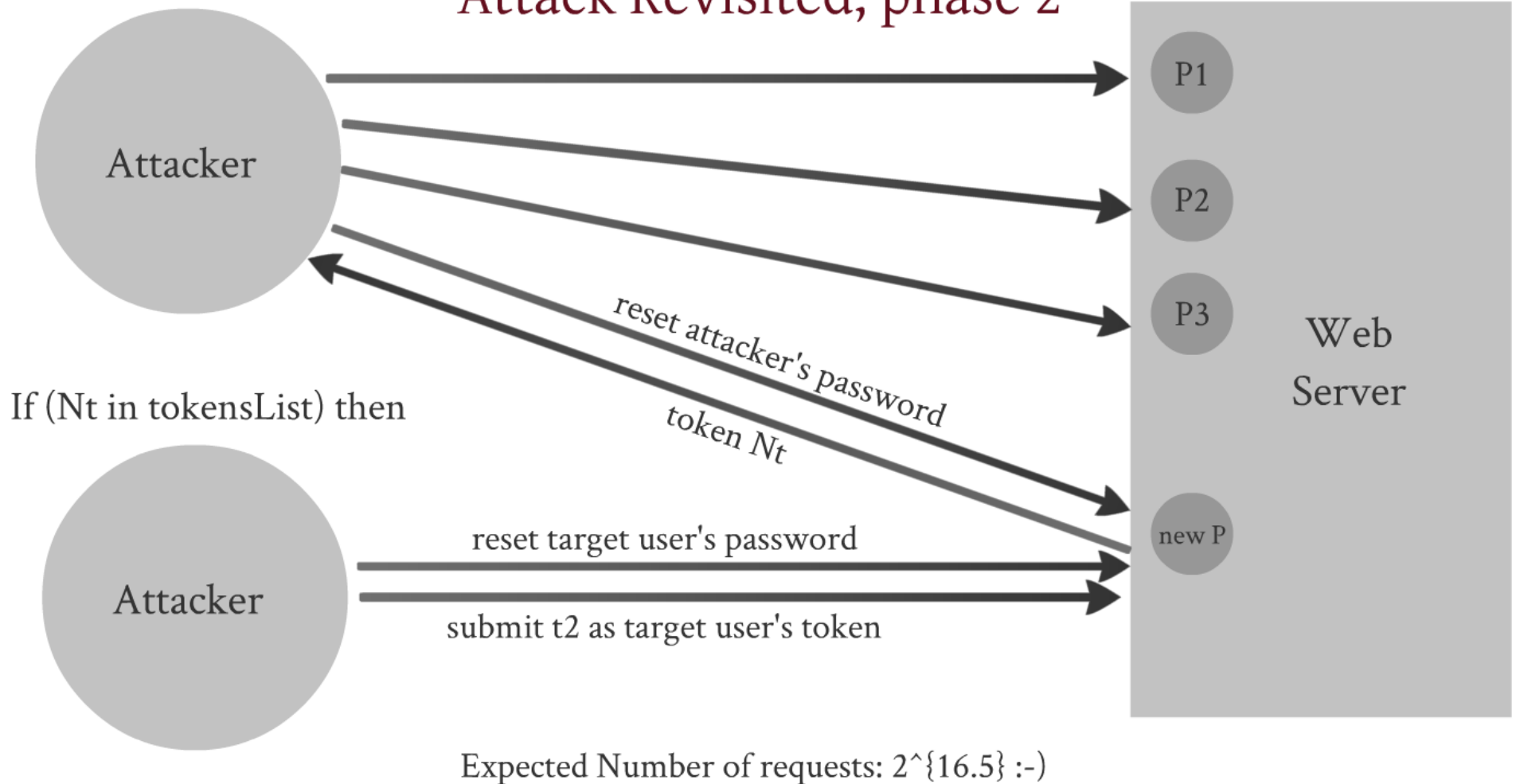
Our goal is to minimize function F.

- F minimizes at $x = 2^{\{15.5\}}$ for which we have that $F(x) \approx 185\text{k}$ requests.
- New process creation will incur a 10% overhead in default Apache installations.

Attack Revisited, phase 1



Attack Revisited, phase 2



This attack shows that a general class of otherwise secure token generation algorithms are vulnerable due to the insecure seed of PHP.

ex. `$token = AES($very_random_key, mt_rand().mt_rand());`

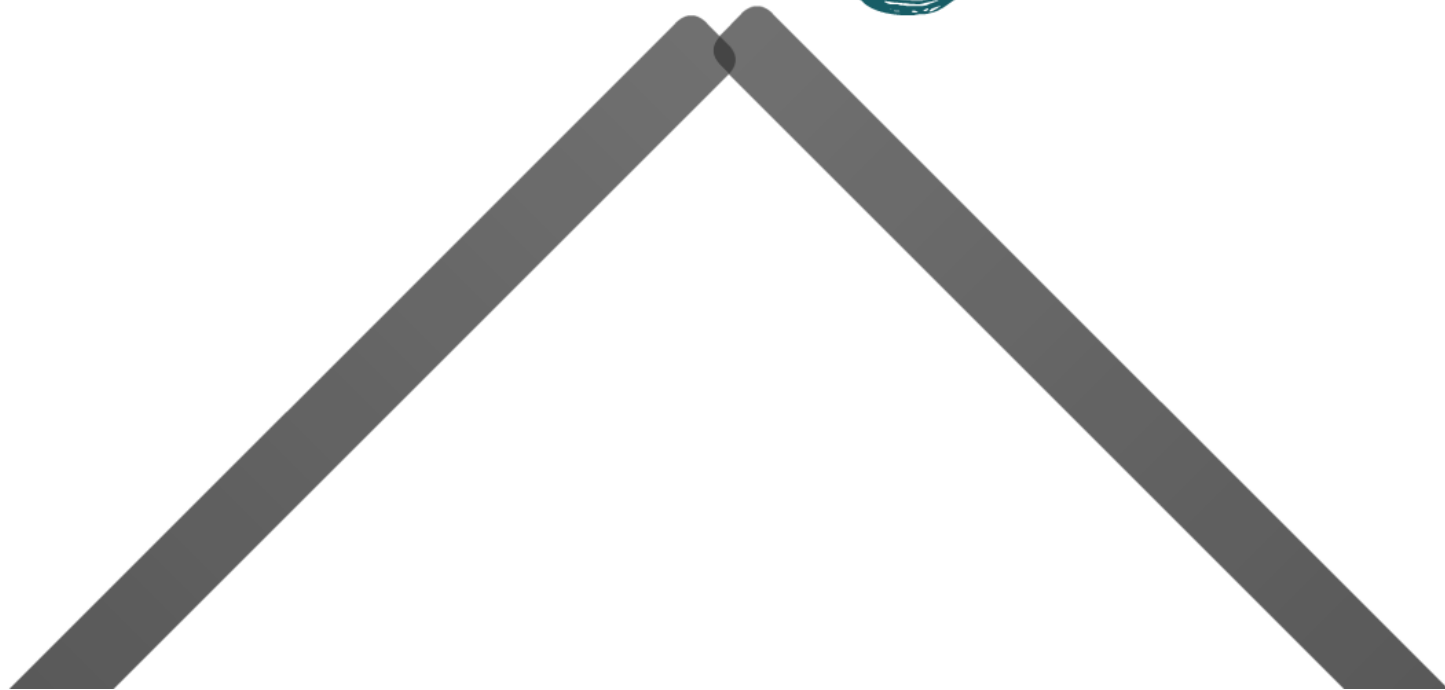


수호신

Suhosin

- Hardening extension for PHP.
- Replace `rand()` with a Mersenne Twister generator with different state than `mt_rand()`.
- Disables `srand()` and `mt_srand()` functions.
- Seeds the generators once at process startup with a secure seed.
 - Entropy gathered from the operating system.

Attacking a PRNG





State Recovery Attacks

All PRNGS in the PHP core are linear

Challenges in predictability

Truncation:

- The output may be truncated to a smaller range.
- This may introduce non linearity to the generator.

Truncation in PHP

In order to truncate a number n from $[M] = \{0, \dots, M-1\}$ to a range $[a, b]$ PHP does the following:

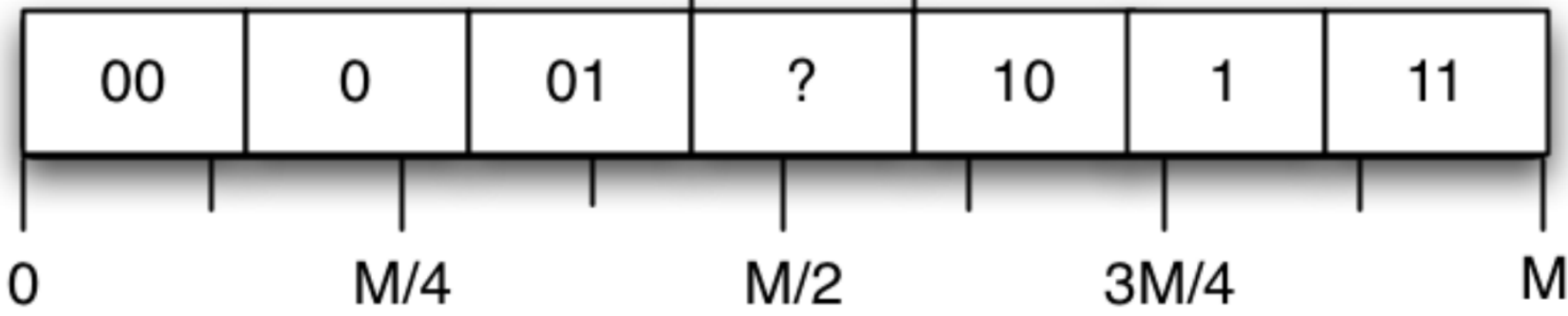
$$l = a + \frac{n \cdot (b - a + 1)}{M}$$

We can view this process as one that puts M values into $b-a+1$ buckets based on their MSBs.

Given a bucket number we can determine a range for the original number n

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

Depending on the number of bits common in the upper and lower bound we can determine some of the MSBs of the original number.



Challenges in predictability

Process identification:

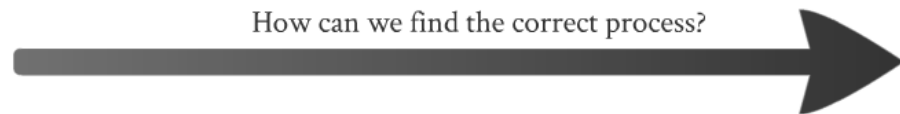
- We want to get all outputs from the same generator.
- Because of the Keep-Alive limit the server might close the connection before the necessary leaks are collected.

peaks are collected.

Process Distinguisher

mit we will be disconnected
number of requests.

How can we find the correct process?



Need access t

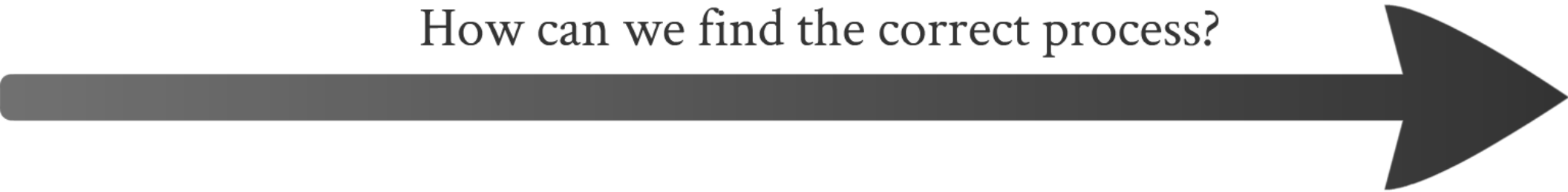
Due to the Keep-alive limit we will be disconnected from our process after a number of requests.

- However we can try to reconnect afterwards.

Algorithm:

- Connect to a fresh process and obtain a session id prein
- Obtain PRNG leaks until the server closes the connecti

How can we find the correct process?





Need access to some process specific state

Idea:

Use the session id preimage as a process specific state to distinguish between server processes.

Algorithm:

- Connect to a fresh process and obtain a session id preimage.
- Obtain PRNG leaks until the server closes the connection.
- Start reconnecting to the server and requesting session identifiers.
 - For each session check if it is generated using the next round of `php_combined_lcg` than the one used in the preimage we have.
 - If a match is found then we have connected to our process.

state recovery for rand()

Ad-Shamir Framework

reduces the problem of uniquely solving an

High level description

- Define a lattice over the coefficients of the

rand() implementation depends on the OS:

> On windows a 15 bit LCG is used

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

> On *nix systems an additive feedback generator is used:

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

> Truncation introduces non linearity.

Hstad-Shamir Framework

Solves the problem of uniquely solving an underdetermined system of linear modular equations when part of the variables is known.

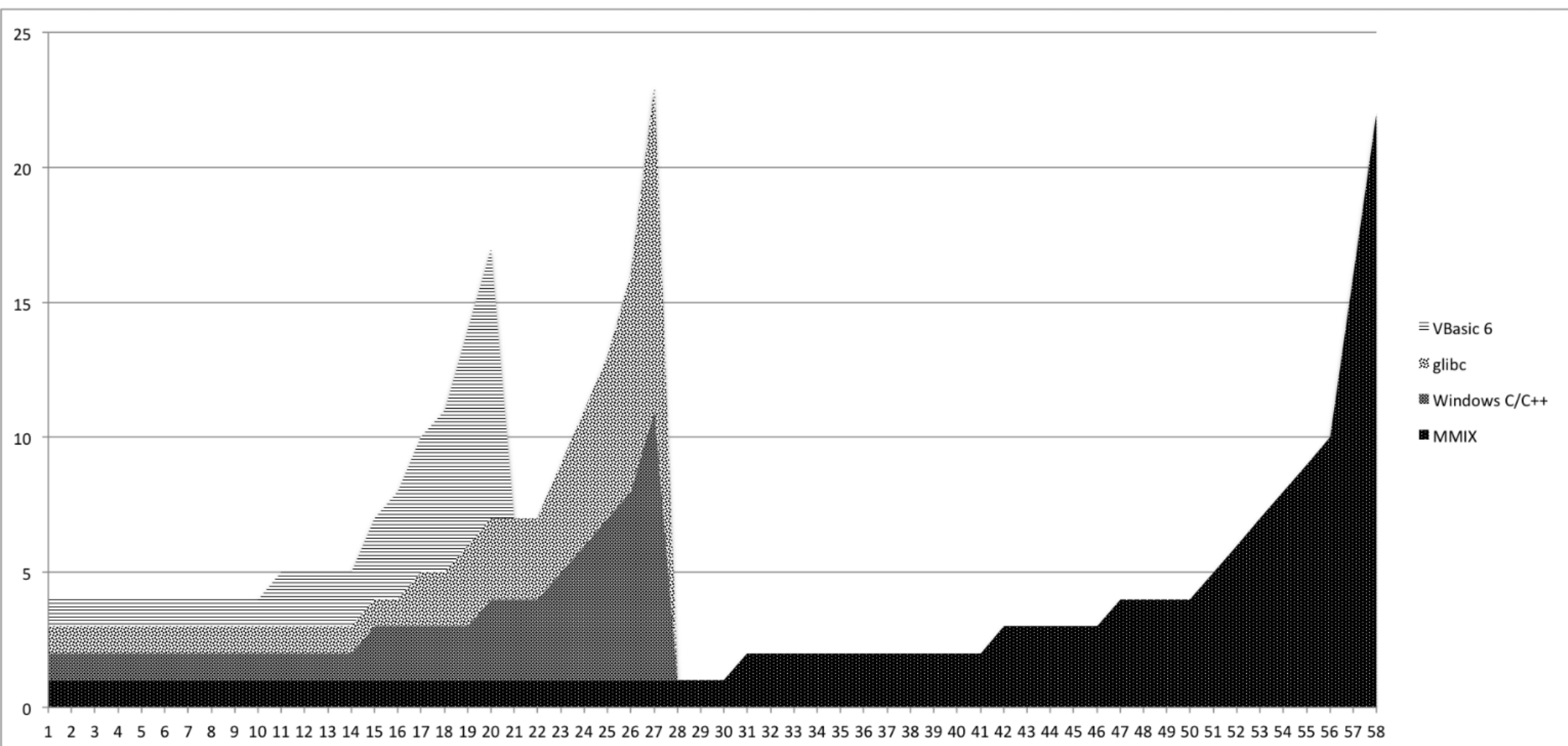
High level description

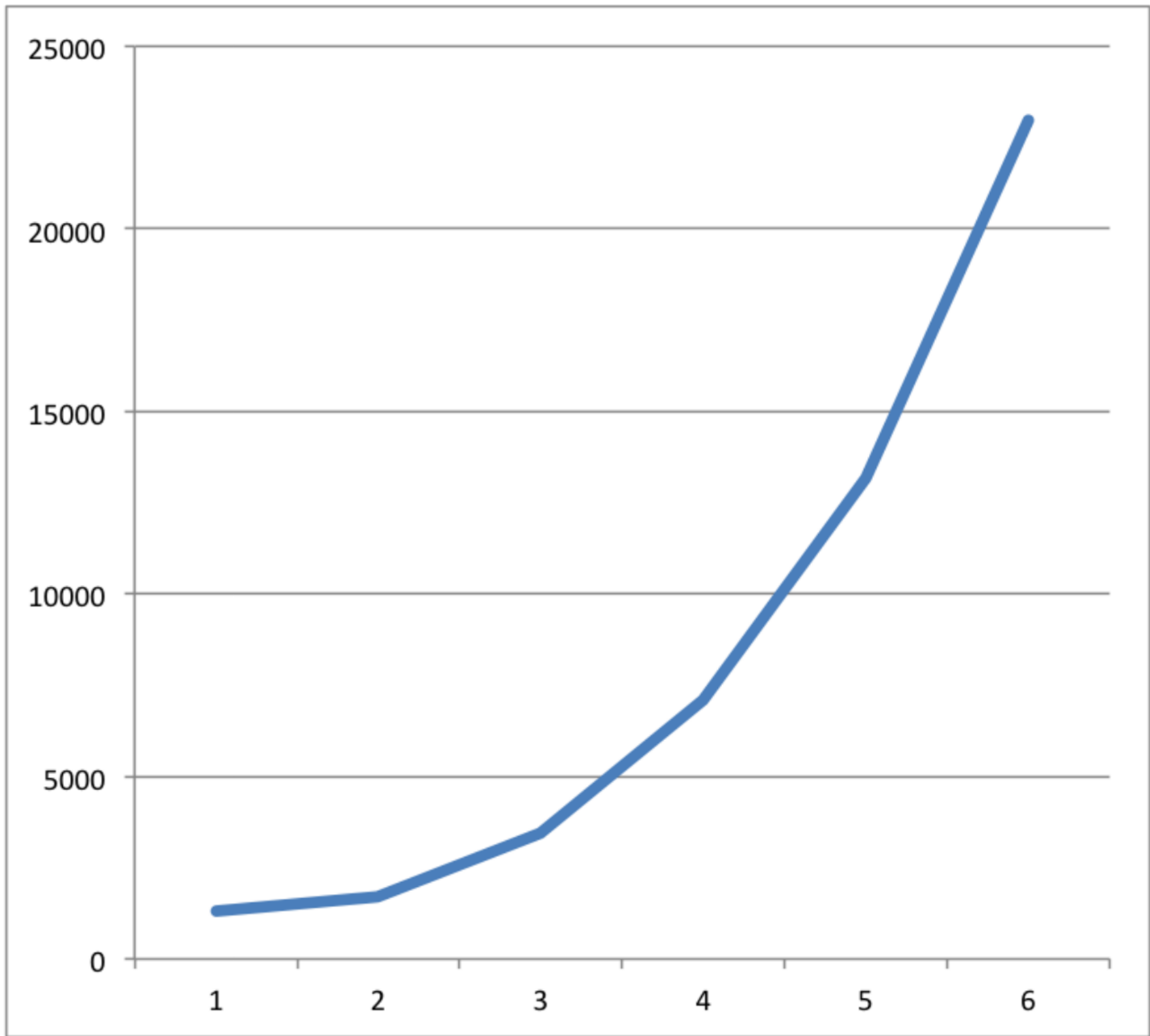
- Define a lattice over the coefficients of the equations
- Find a reduced basis of the lattice using the LLL algorithm.
- Use the fact that the basis vectors are small to uniquely solve the system over the integers.

Bottleneck point:

- Lattice base reduction of a lattice with dimension equal to the number of leaks needed.
- Public LLL algorithm implementations have complexity $O(d^5)$.
 - A $O(d^3 \log d)$ variant exists, but without any public implementation.

Implementation experiments





Summary:

- LCGs can be efficiently recovered unless we have extremely large truncation levels.
- Similar behavior is observed in the glibc generator however the LLL complexity did not allow to recover more than 6 bits.

State recovery for mt_rand()

Notice:

Truncation does not introduce non linearity to the generator.

- Each output bit can be expressed as a linear equation to the internal state.
- Take each output bit obtained and use it to create a linear system.
- If the system has a unique solution, the state can be recovered.

Mersenne Twister:

Based on a linear recurrence over GF(2):

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

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

Huge state of 19937 bits. (a Mersenne Prime)

To improve randomness properties the output of the recurrence is multiplied by an invertible matrix:

$$z = xT$$

T is called the tempering matrix.

Notice:

Truncation does not introduce non linearity to the generator.

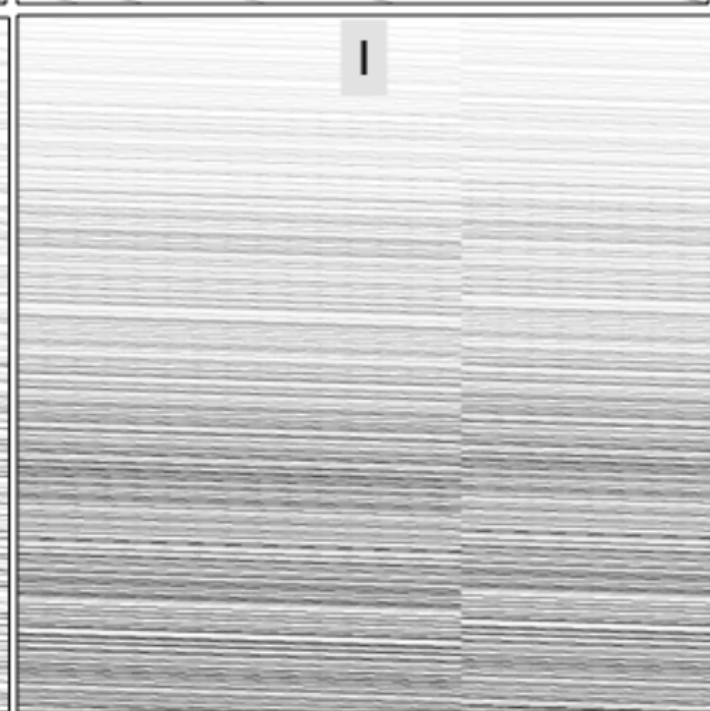
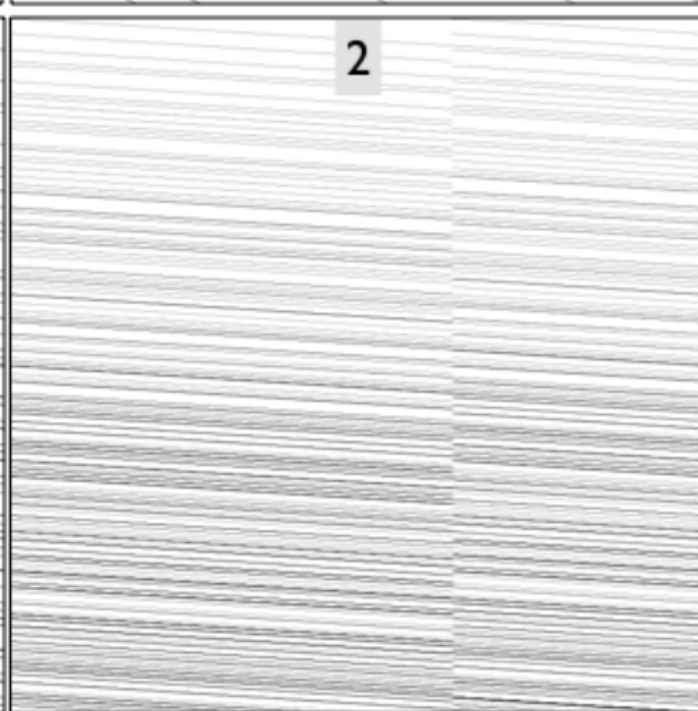
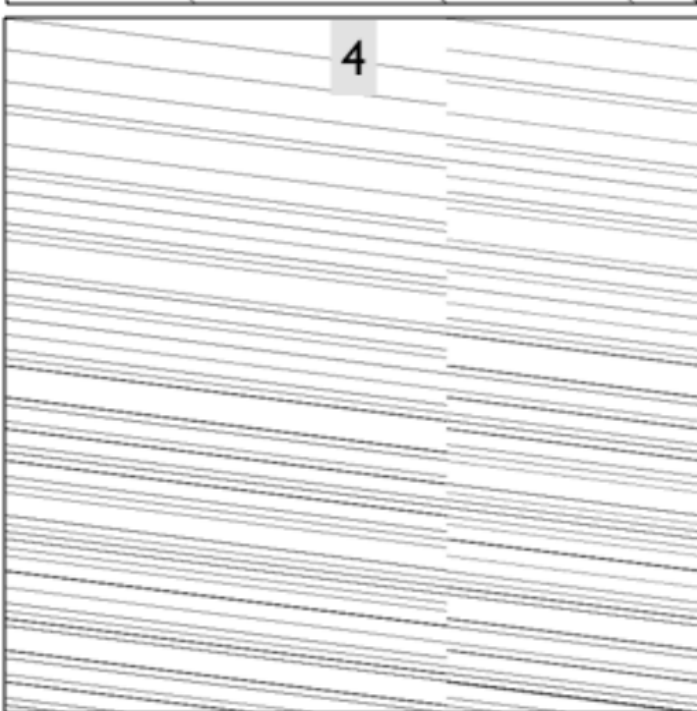
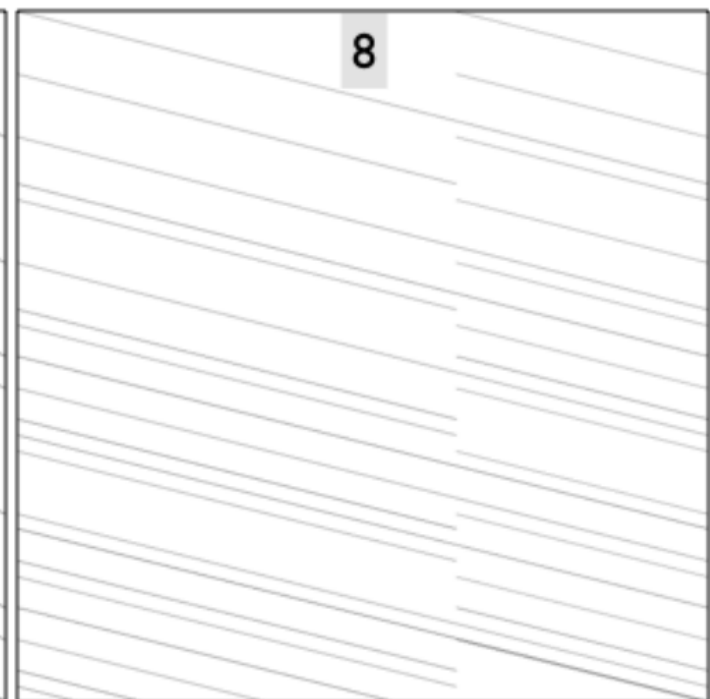
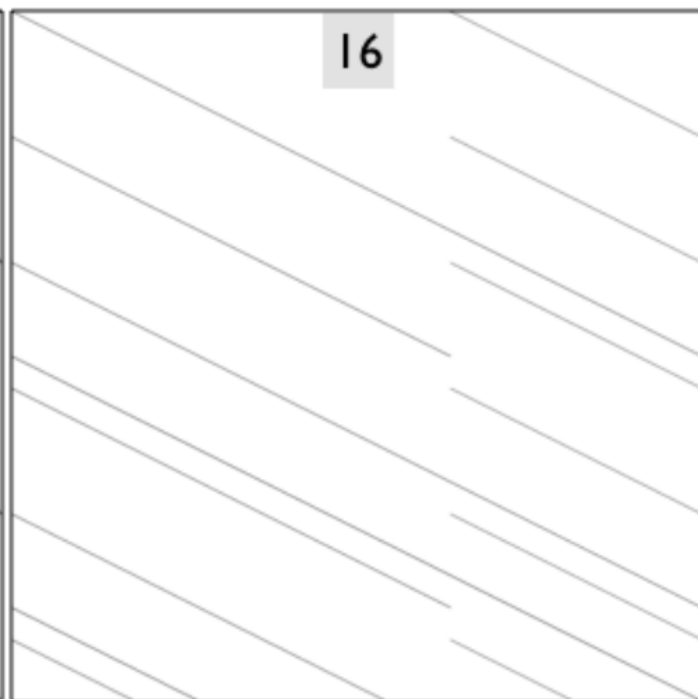
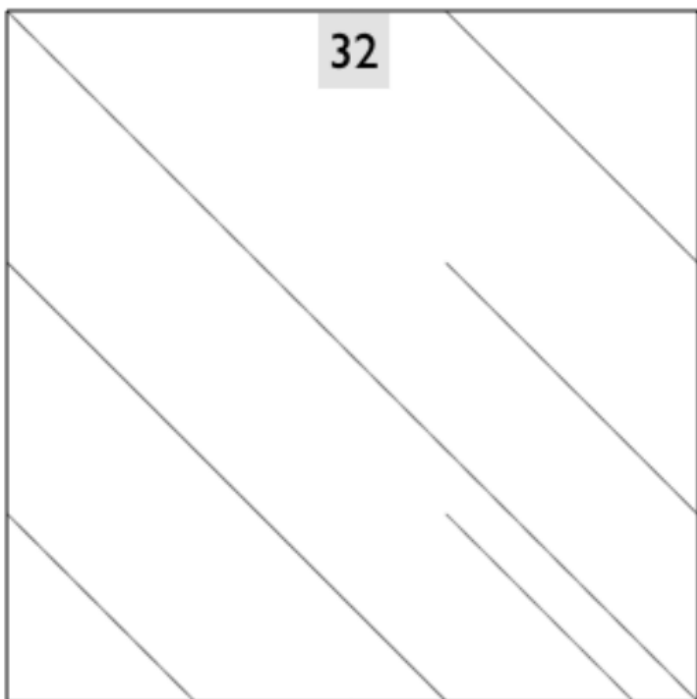
- Each output bit can be expressed as a linear equation to the internal state.
- Take each output bit obtained and create a linear system.
- If the system has a unique solution the solution will give the internal state of the generator.

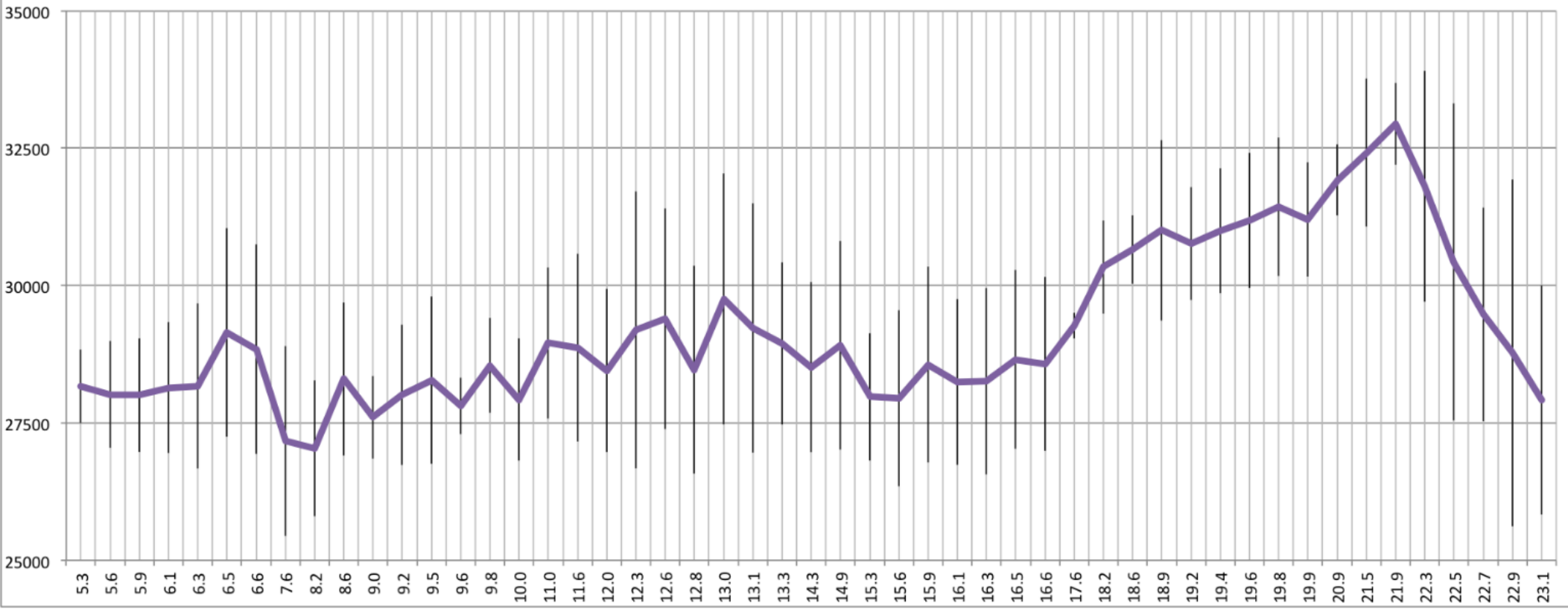
How can we know that the system has a unique solution in advance?

Employ an online gaussian solver:

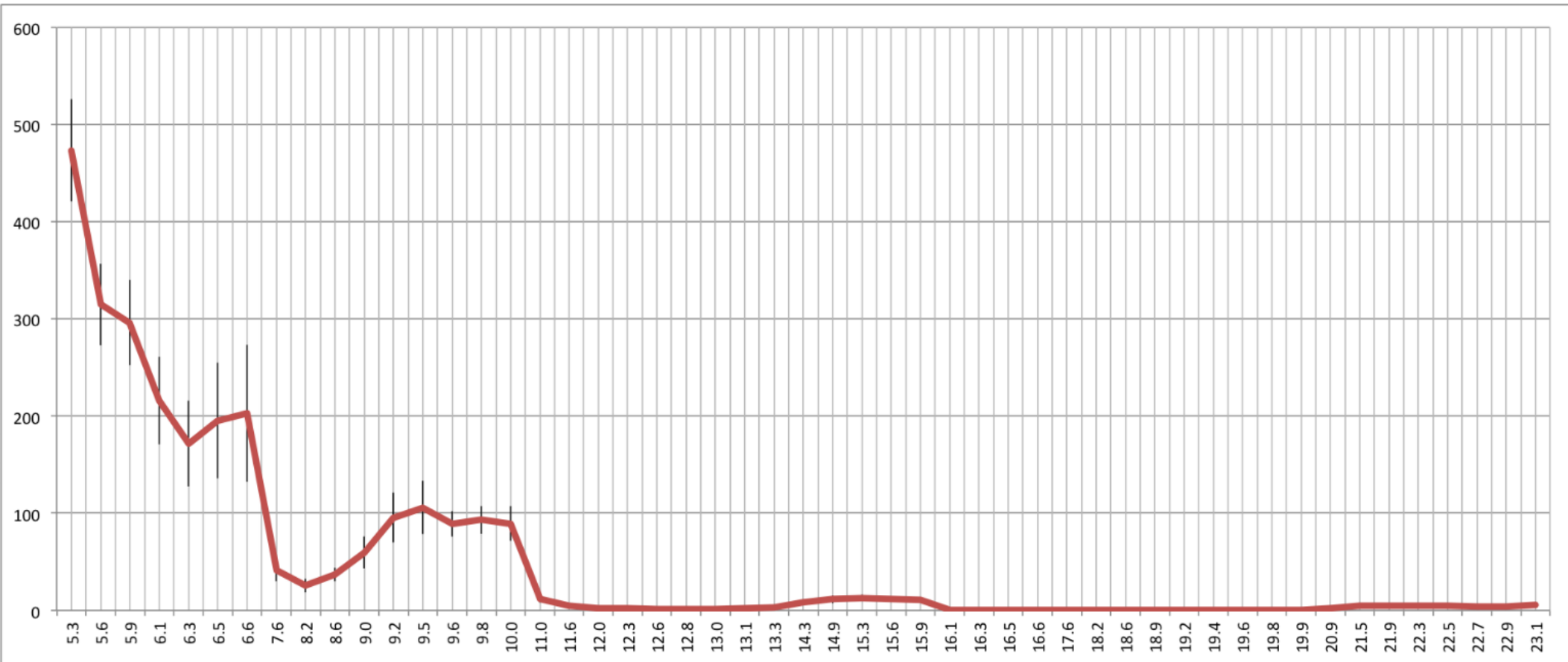
- As equations are obtained from the server add them to the system.
- Stop when the system becomes uniquely solvable.

Implementation experiments





Equations vs truncation.



Time vs Truncation.

Case study

PHORUM

SPEED & POWER


```
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);
}
```

At least 4 `mt_rand()` outputs skipped in each call.

- The resulting system is very dense.

Results:

On average:

- 11 reconnections of the client.
- 30 minutes to compromise the application.

Tools of the Trade

- Nobody likes to write exploits in C!
- A set of tools with a python interface in order to exploit randomness attacks:
 - Online Gaussian solver.
 - Lightweight rainbow tables implementation.
 - Programmable web bruteforcing tool.
- check <http://crypto.di.uoa.gr> for a release.

Randomness Attacks Mitigation

- PHP 5.4 added extra entropy to the session identifier.
 - `session.entropy_length` enabled by default.
- Suggested to add secure seeding to all PRNGs in the PHP core.
 - PHP security team: "This is an application specific problem".
 - Secure PRGs from extensions are rarely used right now.
- A drop in replacement for any token generator can be found in <http://crypto.di.uoa.gr>
 - Checks for crypto strong PRGs in the PHP system
 - Otherwise collects entropy from various sources.

Related Work

Stefan Esser

- `mt_srand` and not so random numbers.

Samy Kamkar

- How I met your girlfriend.

Gregor Kopf

- Non obvious bugs by example.

Summary

- Randomness attacks affect a very large number of PHP applications.
- Exploit mitigations are needed for these attacks.
- Crypto bugs are becoming a trend for exploitation.

Thank You!

Questions?

PRNG: Pwning PseudoRandom number
Generators
(in PHP applications)

George Argyros

Aggelos Kiayias