Reinterpreting and Improving the Cryptanalysis of the Flash Player PRNG

George Teşeleanu[1,2][0000−0003−3953−2744]

[1] Department of Computer Science
“Al.I.Cuza” University of Iaşi 700506 Iaşi, Romania,
george.teseleanu@info.uaic.ro
[2] Advanced Technologies Institute
10 Dinu Vintilă, Bucharest, Romania
tgeorge@dcti.ro

Abstract. Constant blinding is an efficient countermeasure against just-in-time (JIT) spraying attacks. Unfortunately, this mitigation mechanism is not always implemented correctly. One such example is the constant blinding mechanism found in the Adobe Flash Player. Instead of choosing a strong mainstream pseudo-random number generator (PRNG), the Flash Player designers chose to implement a proprietary one. This led to the discovery of a vulnerability that can be exploited to recover the initial seed used by the PRNG and thus, to bypass the constant blinding mechanism. Using this vulnerability as a starting point, we show that no matter the parameters used by the previously mentioned PRNG it still remains a weak construction. A consequence of this study is an improvement of the seed recovering mechanism from previously known complexity of $\mathcal{O}(2^{21})$ to one of $\mathcal{O}(2^{11})$.

1 Introduction

JIT compilers (*e.g.* JavaScript and ActionScript) translate source code or bytecode into machine code at runtime for faster execution. Due to the fact that the purpose of JIT compilers is to produce executable data, they are normally exempt from data execution prevention (DEP[3]). Thus, a vulnerability in a JIT compiler might lead to an exploit undetectable by DEP. One such attack, called JIT spraying, was proposed in [7]. By coercing the ActionScript JIT engine, Blazakis shows how to write shellcode into the executable memory and thus, bypass DEP. The key insight is that the JIT compiler is predictable and must copy some constants to the executable page. Hence, these constants can encode small instructions and then control flow to the next constant’s location.

To defend against JIT spraying attacks, Adobe employs a technique called *constant blinding*. This method prevents an attacker from loading his instructions into constants and thus, blocks the delivery of his malicious script. The idea behind constant blinding is to avoid storing constants in memory in their original form. Instead, they are first XORed with some randomly generated secret cookie and then stored inside the memory. If the secret cookie is generated by means of a weak PRNG[4], the attacker regains his ability to inject malicious instructions.

Instead of using an already proven secure PRNG, the Flash Player designers tried to implement their own PRNG. Unfortunately, in [1, 9] it is shown that the design of the generator is flawed. In [1] a brute force attack is implemented, while in [9] a refined brute force attack is presented. These results have been reported to Adobe under the code CVE-2017-3000 [5] and the vulnerability has been patched in version 25.0.0.127.

In this paper, we refine the attack presented in [9] from a time complexity of $\mathcal{O}(2^{21})$ to one of $\mathcal{O}(2^{11})$. We also show that no matter the parameters used by the PRNG, the flaw remains. More precisely we show that for any parameters the worst brute force attack takes $\mathcal{O}(2^{21})$ operations. In [9] the authors do not present the full algorithm for reversing the PRNG, while in [1] we found the full algorithm, but it was not optimized.

[3] The DEP mechanism performs additional checks on memory to help prevent malicious code from running on a system.

[4] *i.e.*, the seed used to generate the cookie can be recovered in reasonable time

For completeness, in Appendix A we also present an optimized version of the full algorithm. Note that in this paper we only focus on the Flash Player PRNG. For more details about JIT spraying attacks and constant blinding we refer the reader to [6–9].

Structure of the paper. Notations and definitions are presented in Section 2. The core of the paper consists of Sections 3 and 4 and contains a series of algorithms for inverting a generalized version of the hash function used by the Flash Player. Experimental result are given in Section 5. We conclude in Section 6. Supplementary algorithms may be found in Appendix A.

2 Preliminaries

Notations. In this paper we use C language operators (*i*.e. $|/\&$ for bitwise or/and, $\ll$ for left shift and $==$ for equality testing) as well as other widely adopted notations (*i*.e. $\oplus$ for the bitwise xor, $\leftarrow$ for assignment and $\gg_s$ for the right shift of a signed integer). Hexadecimal numbers will always contain the prefix `0x`, while binary ones the prefix `0b`. The subset $\{0, \ldots, q\} \in \mathbb{N}$ will be referred to as $[0, q]$.

By $0^\alpha 1^\beta 0^\gamma$ we will denote an $(\alpha + \beta + \gamma)$-bit word that has α bits of 0, followed by β bits of 1 and γ trailing zeros.

2.1 Constant Blinding in Flash Player

In this subsection we describe the implementation of the Flash Player PRNG, as presented in [3]. The generator has four components (described in Listing 1.1): a seed initialization function (*RandomFastInit*), a seed update function (*RandomFastInit*), a hash function (*RandomPureHasher*) and a cookie generation function (*GenerateRandomNumber*). According to the source code, the hash function is adapted from [10]. Note that the variable uValue is initialized by a function found in the Windows API (*VMPI_getPerformanceCounter*).

The role of the hash function is to make attackers unable to retrieve the seed value (uValue) in reasonable time. Note that the default timeout in Flash Player is $15s$. Thus, an attacker must succeed in finding the seed, predicate the secret value into the next round and embed the desired value in the executable heap in $15s$.

2.2 Shifting Signed Integers

According to [2], if we left shift a signed integer (*e.g.* iSeed) the result is unpredictable and if we right shift a signed negative integer the result is implementation dependent. Thus, we will make a clear distinction between implementation independent or dependent attack strategies against the Flash Player PRNG. In some cases, the attacks devised for a particular implementation are faster than the corresponding implementation independent strategy (see Section 4).

For simplicity, when talking about targeted attacks we consider the behavior of shifts implemented in Microsoft Visual Studio [2] and GCC [4] on x86 and x64 architectures. Thus, left shifts are sign independent (*e.g.* `0b11000000` $\ll 1 =$ `0b10000000`) and right shifts of signed integers use the sign bit to fill vacated bit positions (*e.g.* `0b11000000` $\gg_s 1 =$ `0b11100000` and `0b01000000` $\gg_s 1 =$ `0b00100000`).

2.3 Previous Cryptanalysis Results

By abstracting the code described in Listing 1.1, we identify the three main components of the cookie generation function, *i.e.*:

$$\begin{aligned}
f(x) &= (x \ll 13) \oplus x - (x \gg_s 21), \\
g(x) &= (c_3 \cdot x^3 + c_2 \cdot x + c_1)\ \&\ \texttt{0x7fffffff} + x, \\
h(x) &= 71 \cdot x \bmod 2^{32}.
\end{aligned}$$

If these functions are reversed, then the PRNG is broken. In [9], the authors propose an algorithm for reversing f (Algorithm 1) and a backtracking algorithm for reversing g (the complete description is presented in Algorithm 7). For completeness, we provide in Appendix A the full algorithm (Algorithm 8) for reversing the PRNG (which includes the inverse of h). Note that Algorithm 1 has a time complexity of $\mathcal{O}(2^{21})$ and is implementation independent.

```
#define c3   15731L
#define c2   789221L
#define c1   1376312589L
#define kRandomPureMax 0x7fffffffL

void RandomFastInit(pTRandomFast pRandomFast)
{
    int32_t n = 31;
    pRandomFast->uValue = (uint32_t)(VMPI_getPerformanceCounter());
    pRandomFast->uSequenceLength = (1L << n) - 1L;
    pRandomFast->uXorMask = 0x14000000L;
}

#define RandomFastNext(_pRandomFast)\
(\
    ((_pRandomFast)->uValue & 1L)\
    ? ((_pRandomFast)->uValue = ((_pRandomFast)->uValue >> 1) ^ (_pRandomFast)->
    uXorMask)\
    : ((_pRandomFast)->uValue = ((_pRandomFast)->uValue >> 1))\
)

int32_t RandomPureHasher(int32_t iSeed)
{
    int32_t    iResult;

    iSeed = ((iSeed << 13) ^ iSeed) - (iSeed >> 21);

    iResult = (iSeed*(iSeed*iSeed*c3 + c2) + c1) & kRandomPureMax;
    iResult += iSeed;
    iResult = ((iResult << 13) ^ iResult) - (iResult >> 21);

    return iResult;
}

int32_t GenerateRandomNumber(pTRandomFast pRandomFast)
{
    if (pRandomFast->uValue == 0)
    {
        RandomFastInit(pRandomFast);
    }
    long aNum = RandomFastNext(pRandomFast);
    aNum = RandomPureHasher(aNum * 71L);

    return aNum & kRandomPureMax;
}
```

Listing 1.1. ActionScript PRNG implementation.

Algorithm 1. The algorithm for reversing f

```
Input: The value to reverse $v$.
Output: The set of possible solutions $S$.
$S \leftarrow \varnothing$;
for $low \in [0, \texttt{0x7ff}]$ do
    $temp \leftarrow v \ \& \ \texttt{0x7ff}$;
    if $temp > low$ then
        $high = (1 \ll 11) + low - temp$;
    end
    else
        $high = low - temp$;
    end
    for $mid \in [0, \texttt{0x3ff}]$ do
        $s \leftarrow (high \ll 21) \mid (mid \ll 11) \mid low$;
        if $f(s) == v$ then
            $S \leftarrow S \cup s$;
        end
    end
end
return $S$;
```

3 Reinterpreting

Let n be the word size in bits. As Algorithm 7 can be used to reverse any generic polynomial g and the linear function h can be easily reversed, we only focus on reversing the generic function

$$f(x) = (x \ll \ell) \oplus x - (x \gg_s r),$$

where $1 \leq \ell, r \leq n$ are integers. We further denote by v the output of $f(x)$.

Degenerate Cases. Let $ct = 10^{n-1} \gg_s n$. We consider the cases $\ell, r \in \{0, n\}$ as degenerate due to different inherent weakness induced by these choices. Thus, in our study we do not take in consideration degenerate cases. We further present the weakness associated with the degenerate cases:

- when $r = n$ and $0 < \ell \leq n$, the function $f(x) = x \oplus (x \ll \ell) + ct$ leaks ℓ bits of its seed;
- when $\ell = 0$ and $0 \leq r \leq n$, the function $f(x) = -(x \gg_s r)$ leaks $n - r$ bits of its seed and v has the rest of the bits constant;
- when $r = 0$ and $0 < \ell \leq n$, the function $f(x) = x \oplus (x \ll \ell) - x$ always outputs a v with ℓ trailing zeros;
- when $\ell = n$ and $0 < r \leq n$, the function $f(x) = x - (x \gg_s r)$ leaks r bits of its seed.

We further present a series of attacks that are implementation independent. In the case $n - r \leq \ell, n \leq 2r$ we generalized a different algorithm than Algorithm 1, due to a more direct adaptation to an implementation dependent version.

Lemma 1. *Let $c_{24} = \lfloor n/r \rfloor + 1$. For each (set of) condition(s) presented in Column 2 of Table 1 there exists an attack whose corresponding time complexity is presented in Column 3 of Table 1.*

	Conditions	Time complexity
1	$n-r \le \ell$ and $n \le 2r$	$\mathcal{O}(2^r)$
2	$n-r \le \ell$ and $n \ge 2r$	$\mathcal{O}(c_{24}2^r)$
3	$n-r \ge \ell$ and $n \le 2r$	$\mathcal{O}(2^r)$
4	$n-r \ge \ell$ and $n \ge 2r$	$\mathcal{O}(c_{24}2^r)$

Table 1. Attack Parameters for Lemma 1

Proof. When $n-r \le \ell$, we can explicitly write the function f as shown in Figure 1. Note that the bits used to fill vacated positions are represented as question marks. As we want a compiler independent attack we consider the ? bits as unknown and tailor our attacks accordingly.

In the first case, we first recover the most significant $n-r$ bits (*high*) and then extract the least significant $n-r$ bits (*low*) from $v + high$. For the rest of $2r-n$ bits (*mid*) we do an exhaustive search. This leads to a time complexity of $\mathcal{O}(2^{n-r}2^{2r-n}) = \mathcal{O}(2^r)$.

$$\begin{array}{cccccccccc}
 & a_1 & \dots & a_{n-\ell} & a_{n-\ell+1} & \dots & a_r & a_{r+1} & \dots & a_n \\
\oplus & a_{\ell+1} & \dots & a_n & 0 & \dots & 0 & 0 & \dots & 0 \\
= & t_1 & \dots & t_{n-\ell} & t_{n-\ell+1} & \dots & t_r & t_{r+1} & \dots & t_n \\
+ & ? & \dots & ? & ? & \dots & ? & a_1 & \dots & a_{n-r}
\end{array}$$

Fig. 1. Bit representation of $f(x)$

Algorithm 2. The algorithm for reversing f (Case 1)

```
Input: The value to reverse $v$.
Output: The set of possible solutions $S$.
$S \leftarrow \varnothing$;
for $high \in [0, 1^{n-r}]$ do
    $temp \leftarrow v + high$;
    $low \leftarrow temp \ \& \ 1^{n-r}$;
    for $mid \in [0, 1^{2r-n}]$ do
        $s \leftarrow (high \ll r) \mid (mid \ll (n-r)) \mid low$;
        if $f(s) == v$ then
            $S \leftarrow S \cup s$;
        end
    end
end
return $S$;
```

In the second case, we can do better than simply using Algorithm 2. We first recover the least significant r bits (*low*) and then use *low* to gradually recover the rest of the bits (*mid*). This leads to the complexity $\mathcal{O}(2^r(q+1))$.

When $n-r \ge \ell$, we can explicitly write the function f as depicted in Figure 2. Note that some of the bits resulted from the left shift overlap with some from the right shift. Thus, in the third case we recover the least significant $n-r$ bits (*low*), add the overlapping bits, and then recover the most significant $n-r$ bits (*high*) from v. For the rest of $2r-n$ bits (*mid*) we do an exhaustive search. So, similarly to the first case, we obtain a complexity of $\mathcal{O}(2^r)$.

Algorithm 3. The algorithm for reversing f (Case 2 and 4)

```
Input: The value to reverse $v$.
Output: The set of possible solutions $S$.
Function $Minus(temp_1, temp_2, size)$:
    if $temp_2 > temp_1$ then
        $high = (1 \ll (size + 1)) + temp_1 - temp_2$;
    end
    else
        $high = temp_1 - temp_2$;
    end
    return $high \ \& \ 1^{size}$;
Function $ComputeMid(low)$:
    $q \leftarrow \lfloor (n - r)/r \rfloor$;
    $m \leftarrow n - r \bmod r$;
    $mid \leftarrow 0$;
    for $i \in [1, q]$ do
        $temp_1 \leftarrow (mid \ll r) \mid low$;
        $temp_1 \leftarrow (temp_1 \oplus (temp_1 \ll \ell)) \ \& \ 1^{ir}$; //only for Case 4
        $temp_2 \leftarrow v \ \& \ 1^{ir}$;
        $mid \leftarrow Minus(temp_1, temp_2, ir)$;
    end
    if $m \neq 0$ then
        $temp_1 \leftarrow (mid \ll r) \mid low$;
        $temp_1 \leftarrow (temp_1 \oplus (temp_1 \ll \ell)) \ \& \ 1^{n-r}$; //only for Case 4
        $temp_2 \leftarrow v \ \& \ 1^{n-r}$;
        $mid \leftarrow Minus(temp_1, temp_2, n - r)$;
    end
    return $mid$
Function $Main(v)$:
    $S \leftarrow \varnothing$;
    for $low \in [0, 1^r]$ do
        $mid \leftarrow ComputeMid(low)$;
        $s \leftarrow (mid \ll r) \mid low$;
        if $f(s) == v$ then
            $S \leftarrow S \cup s$;
        end
    end
    return $S$;
```

In the last case, we slightly modify the algorithm used in the second case to take into account the overlapping bits. Thus, the resulting attack has the same complexity $\mathcal{O}(2^r(q+1))$.

$$\begin{array}{ccccccccc}
 & a_1 & \dots & a_r & a_{r+1} & \dots & a_{n-\ell} & a_{n-\ell+1} & \dots & a_n \\
\oplus & a_{\ell+1} & \dots & a_{\ell+r} & a_{\ell+r+1} & \dots & a_n & 0 & \dots & 0 \\
= & t_1 & \dots & t_r & t_{r+1} & \dots & t_{n-\ell} & t_{n-\ell+1} & \dots & t_n \\
+ & ? & \dots & ? & a_1 & \dots & a_{n-r-\ell} & a_{n-r-\ell+1} & \dots & a_{n-r}
\end{array}$$

Fig. 2. Bit representation of $f(x)$

Corollary 1. *There exists an attack on the Flash Player PRNG with time complexity $\mathcal{O}(2^{21})$.*

Algorithm 4. The algorithm for reversing f (Case 3)

```
Input: The value to reverse $v$.
Output: The set of possible solutions $S$.
$S \leftarrow \varnothing$;
for $low \in [0, 1^{n-r}]$ do
    $temp_1 \leftarrow (low \oplus (low \ll \ell))$ & $1^{n-r}$;
    $temp_2 \leftarrow v$ & $1^{n-r}$;
    $high \leftarrow Minus(temp_1, temp_2, n-r)$;
    for $mid \in [0, 1^{2r-n}]$ do
        $s \leftarrow (high \ll r) \mid (mid \ll (n-r)) \mid low$;
        if $f(s) == v$ then
            $S \leftarrow S \cup s$;
        end
    end
end
return $S$;
```

4 Improving

In this section we consider implementation dependent attacks. For simplicity we assume the behavior of the Microsoft Visual Studio and GCC compilers on x86 and x64 architectures. Other compilers' behaviors can be modeled similarly.

Lemma 2. *Let $c_{13} = \lfloor r/\ell \rfloor + 1$. For each (set of) condition(s) presented in Column 2 of Table 1 there exists an attack whose corresponding time complexity is presented in Column 3 of Table 1.*

	Conditions	Time complexity
1	$n-r \leq \ell$ and $n \leq 2r$	$\mathcal{O}(c_{13}2^{n-r})$
3	$n-r \geq \ell$ and $n \leq 2r$	$\mathcal{O}(c_{13}2^{n-r})$

Table 2. Attack Parameters for Lemma 2

Proof. When $n - r \leq \ell$, we can explicit the function f as shown in Figure 3. Note that a_1 is the sign bit used to fill the gaps. With this in mind, we use the existing knowledge (low) to gradually recover the $2r - n$ bits (mid). Thus, we improve the exhaustive search of the mid part from Algorithm 2. This leads to a time complexity of $\mathcal{O}(2^r(q + 1))$.

$$\begin{array}{cccccccccc}
 & a_1 & \dots & a_{n-\ell} & a_{n-\ell+1} & \dots & a_r & a_{r+1} & \dots & a_n \\
\oplus & a_{\ell+1} & \dots & a_n & 0 & \dots & 0 & 0 & \dots & 0 \\
= & t_1 & \dots & t_{n-\ell} & t_{n-\ell+1} & \dots & t_r & t_{r+1} & \dots & t_n \\
+ & a_1 & \dots & a_1 & a_1 & \dots & a_1 & a_1 & \dots & a_{n-r}
\end{array}$$

Fig. 3. Bit representation of $f(x)$

When $n - r \geq \ell$, Figure 2 becomes Figure 4. In the third case, we adapt the algorithm used in Case 1 to take into account overlapping bits. Thus, we obtain the same time complexity.

Algorithm 5. The improved algorithm for reversing f (Case 1)

```
Input: The value to reverse $v$.
Output: The set of possible solutions $S$.
Function $Add(v, high)$:
    if $high \in [0, 1^{n-r-1}]$ then
        $temp \leftarrow v + high$;
    end
    else
        $temp \leftarrow v + high \oplus 1^r 0^{n-r}$;
    end
    return $temp$;
Function $SpeedMid(low, temp, size, step)$:
    $q \leftarrow \lfloor size/step \rfloor$;
    $m \leftarrow size \bmod step$;
    $temp_1 \leftarrow low$;
    for $i \in [0, q-1]$ do
        $offset \leftarrow (i+1) \cdot step$;
        $temp_2 \leftarrow (temp_1 \oplus (temp \gg offset))\ \&\ 1^{n-r}$;
        $mid \leftarrow mid \mid (temp_1 \ll (i \cdot step))$;
        $temp_1 \leftarrow temp_2$
    end
    $offset \leftarrow (q+1) \cdot step$;
    $temp_2 \leftarrow (temp_1 \oplus (temp \gg offset))\ \&\ 1^m$;
    $mid \leftarrow mid \mid (temp_1 \ll (q \cdot step))$;
    return $mid$;
Function $Main(v)$:
    $S \leftarrow \varnothing$;
    for $high \in [0, 1^{n-r}]$ do
        $temp \leftarrow Add(v, high)$;
        $low \leftarrow temp\ \&\ 1^{\ell}$;
        $mid \leftarrow SpeedMid(low, temp, r-\ell, \ell)$;
        $s \leftarrow (high \ll r) \mid (mid \ll \ell) \mid low$;
        if $f(s) == v$ then
            $S \leftarrow S \cup s$;
        end
    end
    return $S$;
```

$$\begin{array}{cccccccccc}
 & a_1 & \dots & a_r & a_{r+1} & \dots & a_{n-\ell} & a_{n-\ell+1} & \dots & a_n \\
\oplus & a_{\ell+1} & \dots & a_{\ell+r} & a_{\ell+r+1} & \dots & a_n & 0 & \dots & 0 \\
= & t_1 & \dots & t_r & t_{r+1} & \dots & t_{n-\ell} & t_{n-\ell+1} & \dots & t_n \\
+ & a_1 & \dots & a_1 & a_1 & \dots & a_{n-r-\ell} & a_{n-r-\ell+1} & \dots & a_{n-r}
\end{array}$$

Fig. 4. Bit representation of $f(x)$

Corollary 2. *There exist an attack on the Flash Player PRNG with time complexity* $\mathcal{O}(2^{11})$.

Corollary 3. *For any choice of* ℓ *and* r *there exists an attack whose time complexity is at most* $\mathcal{O}(n2^{n/2})$.

Proof. According to Lemma 1, Cases 2 and 4 there exists an attack with complexity $\mathcal{O}(2^r) \le \mathcal{O}(2^{n/2})$. In Cases 1 and 3 we make use of the attacks presented in Lemma 2. Thus, there exists an attack with complexity $\mathcal{O}(c_{13}2^{n-r}) \le \mathcal{O}(c_{13}2^{n/2}) \le \mathcal{O}(n2^{n/2})$. As a result, in the general case we obtain our statement.

Algorithm 6. The improved algorithm for reversing f (Case 3)

```
Input: The value to reverse $v$.
Output: The set of possible solutions $S$.
$S \leftarrow \varnothing$; $e \leftarrow n - r - \ell$;
for $low \in [0, 1^{n-r}]$ do
    $temp_1 \leftarrow (low \oplus (low \ll \ell))$ & $1^{n-r}$;
    $temp_2 \leftarrow v$ & $1^{n-r}$;
    $high \leftarrow Minus(temp_1, temp_2, n-r)$;
    $temp \leftarrow Add(v, high)$;
    $mid \leftarrow SpeedMid(low, temp, 2r - n + e, \ell)$;
    $mid \leftarrow mid \gg e$;
    $s \leftarrow (high \ll r) \mid (mid \ll (n-r)) \mid low$;
    if $f(s) == v$ then
        $S \leftarrow S \cup s$;
    end
end
return $S$;
```

Corollary 4. *There exists an attack on the Flash Player PRNG with time complexity at most $\mathcal{O}(2^{21})$ independent of ℓ and r.*

5 Experimental Results

We implemented Algorithms 2 to 6 and used 32 random seed values to test if our algorithms succeed in recovering the seed for all $1 \leq r < 32$ and $1 \leq l < 32$. The compilers we worked with are Microsoft Visual Studio 2017 version 15.7.5 with the C++14 extension activated and GCC version 5.4.0 with the C++11 extensions activated. The tests were a success.

In another experiment we run Algorithms 2 to 6 and Algorithm 8 with 2000 random seed values and used the function *omp_get_wtime()* to compute the running time necessary to invert the function f and the corresponding PRNG. The programs were run on a CPU Intel i7-4790 4.00 GHz and compiled with GCC with the O3 flag activated. The results for the 2000 iterations can be found in Table 3. Note that the average time for brute forcing one value is 2.88861s for f and 13.2578s for PRNG.

	Case 1 ($l = 13, r = 21$)			Case 2 ($l = 23, r = 11$)	Case 3 ($l = 9, r = 21$)		Case 4 ($l = 19, r = 11$)
	Algorithm 1	Algorithm 2	Algorithm 5	Algorithm 3	Algorithm 4	Algorithm 6	Algorithm 3
$f(x)$	2.16055s	2.82102s	0.00717478s	0.00608366s	2.77496s	0.00708749s	0.00809854s
PRNG	10.6442s	13.8981s	0.036917s	0.0334592s	14.6386s	0.0432187s	0.0437757s

Table 3. Running times for reversing the function f and the PRNG.

6 Conclusions

In this paper we improved the results from [9] and shown that no matter the parameters used by the Flash Player PRNG, there exists always a brute force attack with complexity at most $\mathcal{O}(n2^{n/2})$. As a consequence, we prove that the secret cookie used for constant blinding can always be recovered due to the weak design of the PRNG. Note that the results presented in this paper might be further improved if one uses other cryptanalytic methods, besides brute force. We leave this research direction as an open problem.

References

1. A Full Exploit of CVE-2017-3000 on Flash Player Constant Blinding PRNG. https://github.com/dangokyo/CVE-2017-3000/blob/master/Exploiter.as.
2. Left Shift and Right Shift Operators. https://docs.microsoft.com/en-us/cpp/cpp/left-shift-and-right-shift-operators-input-and-output?view=vs-2017.
3. Source Code for the Actionscript Virtual Machine. https://github.com/adobe-flash/avmplus/tree/master/core/MathUtils.cpp.
4. Using the GNU Compiler Collection. https://gcc.gnu.org/onlinedocs/gcc/Integers-implementation.html.
5. Vulnerability Details: CVE-2017-3000. https://www.cvedetails.com/cve/CVE-2017-3000/.
6. Michalis Athanasakis, Elias Athanasopoulos, Michalis Polychronakis, Georgios Portokalidis, and Sotiris Ioannidis. The Devil is in the Constants: Bypassing Defences in Browser JIT Engines. In *NDSS 2015*. The Internet Society, 2015.
7. Dionysus Blazakis. Interpreter Exploitation. In *WOOT 2010*. USENIX Association, 2010.
8. Elena Reshetova, Filippo Bonazzi, and N. Asokan. Randomization Can't Stop BPF JIT spray. In *NSS 2017*, volume 10394 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2017.
9. Chenyu Wang, Tao Huang, and Hongjun Wu. On the Weakness of Constant Blinding PRNG in Flash Player. In *ICICS 2018*, volume 11149 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 2018.
10. Greg Ward. A Recursive Implementation of the Perlin Noise Function. In *Graphics Gems II*, pages 396–401. Elsevier, 1991.

A Additional Algorithms

In [9] the algorithm used to invert g is not presented in full. Based on the descriptions found in [1, 9] we present the full algorithm in Algorithm 7. Note that the algorithm works for any generic polynomial g, not only for the one used in the Flash Player PRNG. Note that $\&S$ means that we pass S by reference.

Algorithm 7. Backtracking algorithm for reversing g

```
Input: The value to reverse $v$
Output: The set of possible solutions $S$
Function $Verify\_ith\_bit(v, i, sol)$:
    $b_1 \leftarrow g(sol)\ \&\ (1 \ll i)$;
    $b_2 \leftarrow v\ \&\ (1 \ll i)$;
    return $bit_1 == bit_2$;
Function $Add\_ith\_bit(v, i, sol, \&S, b)$:
    $sol \leftarrow sol \mid (b \ll i)$;
    if $Verify\_ith\_bit(v, i, sol) ==$ true then
        $i \leftarrow i + 1$;
        $Reverse\_bit(v, i, sol, S)$;
    end
Function $Reverse\_bit(v, i, sol, \&S)$:
    if $i == n$ then
        $S \leftarrow S \cup sol$;
        return;
    end
    $add\_ith\_bit(v, i, sol, S, 0)$;
    $add\_ith\_bit(v, i, sol, S, 1)$;
Function $Reverse\_polynomial(v)$:
    $S \leftarrow \varnothing$; //the set of possible solutions
    $i \leftarrow 0$; //the target bit
    $sol \leftarrow 0$; //the current solution
    $reverse\_bit(v, i, sol, S)$;
    return $S$;
```

The only algorithm we found for reversing the Flash Player PRNG is described in [1]. We improve their attack in Algorithm 8. To reverse the bit manipulation function f and the polynomial g we use the abstract functions *Reverse_bit_manipulation* and *Reverse_polynomial*, respectively. Remark that Algorithm 8 works for any generic polynomial g and any generic function $h(x) = p \cdot x \bmod 2^n$ with p odd. In the Flash Player case we have $p^{-1} \equiv 3811027319 \bmod 2^{32}$.

Algorithm 8. The algorithm for reversing the PRNG

```
Input: The value to reverse $v$
Output: The set of possible solutions $S$
1  $v' \leftarrow v \mid (1 \ll (n-1))$;
2  $S_{bit} \leftarrow Reverse\_bit\_manipulation(v) \cup Reverse\_bit\_manipulation(v')$;
3  $S_{pol}, S_{hash}, S \leftarrow \varnothing$;
4  for $s_{bit} \in S_{bit}$ do
5      $S_{pol} \leftarrow S_{pol} \cup Reverse\_polynomial(s_{bit})$;
6  end
7  for $s_{pol} \in S_{pol}$ do
8      $S_{hash} \leftarrow S_{hash} \cup Reverse\_bit\_manipulation(s_{pol})$;
9  end
10 for $s_{hash} \in S_{hash}$ do
11     $s \leftarrow s_{hash} \cdot p^{-1} \bmod 2^n$;
12     $S \leftarrow S \cup s$;
13 end
14 return $S$;
```