Analyzing Cryptographic Functions in Java for JIT-Based Sidechannels

William Eiers

June 20, 2018

Abstract

In the context of runtime systems, security is of utmost importance, especially for cryptographic functions. One particular class of vulnerabilities, side-channel vulnerabilities, have seen increasing popularity hackers due to both its subtlety and complexity. Most side-channel attacks take advantage of the fact that to compute anything, it takes both time and space to do so. Programs containing side-channel vulnerabilities leak information through which a potential attacker can learn secret information simply by observing the programs execution. Many side-channels are simply present because of bad-code practices, but others are much more subtle. In this project, we aim to investigate side-channel vulnerabilities arising from Just-In-Time (JIT) compilation of Java cryptographic functions (such as ModPow), where potential vulnerabilities not originally present in the source code may arise from JIT optimizations.

1 Introduction and Motivation

Side-channel attacks against software have become more popular in recent years. Optimizations within core algorithms used in software are implemented by developers, causing different execution paths to be taken during program execution. Such optimizations can cause the program to spend longer amounts of time or use different amounts of memory, depending on the inputs to the program. This creates a side-channel, a way for an outside agent to possibly learn secret information using observable differences in resource usage [2, 5, 6]. Developer-introduced optimizations are just one way a side-channel can be realized. However, optimizations can be implemented automatically, possibly by a compiler during code generation. Runtime systems, such as the Java Virtual Machine (JVM), operate on architecture-independent bytecode by interpreting it into architecture-dependent code on execution. This causes a huge performance hit, however, as interpretation is orders-of-magnitute slower than compiled code. The JVM gets some of this performance back by introducing runtime optimizations, such as method compilation or compiling paths which are taken multiple times. These optimizations create a case much similar to the one above, where developers inadverdently induce a side-channel. In cryptographic functions, such as the Java BigInteger method modPow, a single invocation causes specific paths to be "heated up" allowing the JVM to optimize these paths due to properties of secret values, such as the private exponent. By gaining some performance back, it is possible that the JVM introduces a side-channel, possibly leaking information about secret values.

1.1 Outline

In this report, we aim to investigate how the Java HotSpot Virtual Machine affects the presence of side-channels, mainly in the context of cryptographic functions. Specifically, we explore the effect of Just-In-Time compiler optimizations with the following questions in mind:

- What is Just-In-Time compilation?
- In what ways can Just-In-Time compilation impact the presence or strength of side-channels?
- Can cryptographic functions leak information due to JIT optimizations?

All experiments were run on an Intel i5-6600K CPU at 3.5 GHz and 32GB of RAM, running the KDE Neon Linux 16.04 distribution. We used Java 8 SE, version 1.8.0_171, Java HotSpot(TM) 64-bit server Virtual Machine.

2 Just-In-Time (JIT) Compilation

The JIT compiler is a runtime component of the Java Virutal Machine which aims to recover some of the performance lost due to interpretation of architecture-independent bytecode. Normally, at runtime Java class files are loaded into the JVM and interpreted line-by-line, causing a huge overhead for each instruction. When JIT is enabled, the JIT compiler interacts with the JVM and attempts to compile code it sees as being importance to performance. For example, the first time a method is invoked, the JIT compiler will compile the method, introducing a small overhead during the initial invocation. Subsequent calls to the method will use the compiled bytecode, rather than intepreting the code line-by-line, boosting performance tremendously. If the method is taken multiple times, the JIT compiler will attempt to optimize the method even further. The possible levels of compilation include five tiers, from purely interpreted (L0) to purely optimized (L4). In this report, we specifically look at the effects of L4 compilation in the context of the modular exponentiation function within the Java library. Other kinds of optimizations include (but are not limited to):

- *Method compilation.* Depending on how often a method is called, the method can be continually recompiled to the next optimization tier.

- *Branch prediction.* Similar to branch-prediction at the micro-architectural level, the HotSpot virtual machine keeps track of how often different conditional branches are taken, and uses this information to generate more efficient native code where the more frequent branch appears first.
- *Optimistic compilation.* As a method gets more rigorously compiled (towards the L4 level), if a particular branch is taken nearly all the time, HotSpot will optimize away the rarely taken branch, leaving an exception-like *trap* in its place.

3 Side-channel Attacks

Side-channels are indirect information channels in which an attacker can learn secret information just by observing differences in resource consumption through said channel. Essentially, side-channel attacks take advantage of the fact that for a program to compute anything useful, it must use some kind of resource to do so. This can be time, memory, or even power usage. This resource usage is secondary to the program's goal: that is, the resource usage is not the end goal of the program. Additionally, such usage tends to be observable: programs generally take varying amounts of time to perform a task based on the inputs received. For example, factoring a small number is computationally less expensive than factoring large numbers. A side-channel attack occurs when an attacker can use the observable differences leaked through the side-channel to learn secret information within the program. We consider one such case in the next section.

3.1 Insecure Password Checker

Consider the password checking method in Figure 1. This function checks whether a user-input string $guess$ matches the stored password $password$, which we refer to as the secret value. This particular implementation walks character by character and returns as soon as it finds a mismatch. This optimization, however, introduces a side-channel in the code, causing the function to be insecure. Particularly, with no prior knowledge of the secret value, a potential attacker can leverage a side-channl attack by trying different passwords and measuring the time taken for the function to return.

To illustrate this timing difference, we assume the secret value (stored word) is $LEET$ and measure the time it takes for the `passwordCheckerInsecure` to return on five different inputs to the JVM. The length of each input is 4, the same as the secret value. For zero matched characters, time taken was 200ns. Each matching character increased this value by 200ns, taking 800ns when all characers matched. While the timing difference is small, it is nonetheless observable, allowing an attacker to learn the secret password. When JIT was enabled, no characters matched took 45ns, one character 51ns, two characters 50ns, three characters 60ns, four characters 13617ns. The extreme timing of the last case is an example of a complex JIT compilation sequence, where either

```java
public static boolean passwordCheckerInsecure(final String
     guess) {
    if (guess.length() != password.length())
        return false;
    for (int i = 0; i < guess.length(); ++i) {
        if (guess.charAt(i) != password.charAt(i)) {
            return false;
        }
    }
    return true;
}
```

Figure 1: Insecure Password checker Code.

the last invocation caused an uncommon trap to be triggered, or a new level of optimization to be induced.

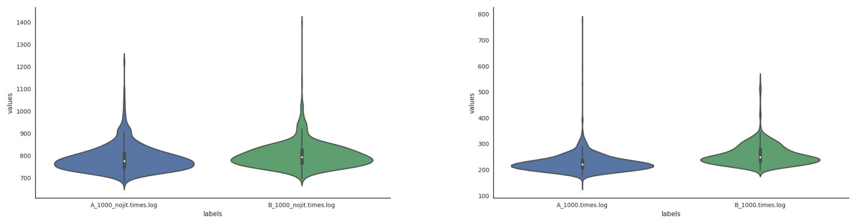
3.2 Secure Password Checker

Ideally, if the `passwordCheckerInsecure` function took the same amount of time regardless of the input, then there would be no observable timing differences. To locate such imbalances, developers can use static analysis tenchniques [4, 7, 1, 3] to analyze program code and fix possible side-channels. This is the case for the `passwordCheckerSecure` function in Figure 2. The function is made secure by making sure all inputs take an equal amount of time to check the password. However, Just-In-Time compilers introduce various optimization techniques in the runtime dynamically which may introduce a timing imbalance, depending on the inputs.

```java
public static boolean passwordCheckerSecure(final String
     password, final String guess) {
    boolean matched = true;
    for (int i = 0; i < password.length(); ++i) {
        if (guess.charAt(i) != password.charAt(i)) {
            matched = matched & false;
        } else {
            matched = matched & true;
        }
    }
    return matched;
}
```

Figure 2: Secure Password checker Code.

Our goal is to show that `passwordCheckerSecure` function may contain



(a) JIT disabled (b) JIT enabled

Figure 3: Timing difference for two different heated paths. Time is in nanoseconds.

a JIT-induced side-channel. Our experiment setup considered two executions paths, one where no characters match, and one where only the first characters match. We consider the following approach:

- Prime the JVM environment a large number of times with a single input string which matches none of the characters of password
- Time a single invocation of secure password checker on a different value for both cases and compare the results

We assume both inputs are of length 4, and the secret password is again $LEET$. We performed 1000 trials of this process and averaged the result. The number of priming iterations was 40000. The results are shown in Figure 3. Clearly, when JIT is disabled (Figure 3a) we see no side-channel. When JIT is enabled (Figure 3b) we see a slight difference in the timing values. Again, the observable difference is small, but noticeable.

4 Case Study: Java ModPow

Side-channels, whether induced by the JIT compiler or by developers themselves, are much easier to detect when encompassed code is relatively small. Modern cryptographic functions contain much more code than the password checker example mentioned in Section 3.2. We consider the Java function `BigInteger.modPow`, for which the source code can be found in the Appendix A. The core of `BigInteger.modPow` function does only a small amount of work, relegating much of the computation to the functions `BigInteger.oddModPow` and `BigInteger.montgomeryMultiply`. The algorithm used is a variant of the sliding window algorithm equipped with montgomery multiplication. Given a *base*, *exponent*, and *modulus* the algorithm works as follows. Initially, if *modulus* is even, the algorithm separates it into an odd part (m_1) and a power of two (m_2), exponentiates mod m_1 through `BigInteger.oddModPow`, manually exponentiates mod m_2, and uses the Chinese Remainder Theorem to combine the results. If *modulus* is odd, the algorithm simply calls `BigInteger.oddModPow` to perform the modular exponentiation. Regardless of whether or not *modulus* is even, the majority of the work is done in the `BigInteger.montgomeryMultiply` function and its derivatives (implMontgomeryMultiply and montReduce).

Due to the large amount of code involved in even a single `BigInteger.modPow` invocation, and the limited time available for experimentation, we were unable to successfully induce a JIT-based side-channel in the code. Instead, we investigated the different levels of optimization induced by JIT compilation, exploring the possibility that a side-channel might be induced by carefully crafting input values which trigger the diffent levels of optimization. In particular, we focused on the number of times `BigInteger.montgomeryMultiply` was compiled to the L4 level, and the effect on timing the compilation induced. We considered ten randomly generated exponents, each with 20 digits and roughly equal in magnitude, and used a similar priming scheme as the one mentioned in Section 3.2. We used the following values for the base and modulus, as well as the exponent value we used for the single timing of `BigInteger.modPow`:

- base $b = 5973054346545950711$
- even modulus $m = 16427260672836758 22$
- odd modulus $m = 16427260672836758 21$
- text exponent $e = 16768880304745619701$

We experimented with priming values $p = \{21, 22\}$ and considered cases where the modulus was even and where the modulus was odd. For space reasons, the results for `BigInteger.montgomeryMultiply` are shown in Table 1, while the full results are shown in Appendix B. The second column shows how many times montgomeryMultiplication was invoked in a single `modPow` invocation. The interesting case is highlighted in Table 1, for the fifth exponent value $e = 21581371295657932221$. In both cases, `montgomeryMultiply` was not compiled to the L4 level in any of the 100 trials when primed 21 times. When we primed for 22 times, montgomeryMultiply was compiled to L4 in most of the cases, resulting in a performance overhead in the timing of the last invocation. This occurred for both even and odd modulus, though the timing is more prevalent when the modulus was odd. Interestingly, in the timings for the even case for all the exponents are higher when the modulus was even. We believe this is due to the increased amount of work done when the modulus is even. While this clearly isn't a side-channel in and of itself, it gives credence to the idea that JIT compilation may be able to induce side-channels in rather complex blocks of code.

5 Conclusion

In this report we explored the possibility that Just-In-Time compilation optimizations within the context of runtime systems can inadverdently induce side-channels in previously secure code. Specifically, we investigated the JIT

(a) Odd modulus

Exponent	# Calls	nP = 21		nP = 22	
		# L4	Time	# L4	Time
25730899574802462604	436	4	80941 (98%)	1	69815 (88%)
24660523187409145475	436	9	81926 (99%)	2	69605 (87%)
33649042240140657826	458	0	71057 (86%)	0	67897 (85%)
28324482328545617634	436	2	81665 (98%)	4	65914 (83%)
21581371295657932221	392	0	69231 (83%)	81	77826 (98%)
25652608396773858801	436	5	80565 (97%)	3	68651 (86%)
24341655831718219991	414	33	69864 (84%)	10	79685 (100%)
23536477189379635045	436	0	78880 (95%)	3	69932 (88%)
24020887706891028596	436	4	82994 (100%)	6	69765 (88%)
25608332753867915599	436	1	80707 (97%)	3	69593 (87%)

(b) Even modulus

Exponent	# Calls	nP = 21		nP = 22	
		# L4	Time	# L4	Time
25730899574802462604	436	99	96145 (90%)	99	85493 (87%)
24660523187409145475	436	100	95067 (89%)	100	87333 (89%)
33649042240140657826	458	98	98635 (93%)	96	87591 (89%)
28324482328545617634	436	99	98545 (92%)	98	89117 (90%)
21581371295657932221	392	0	106599 (100%)	94	98678 (100%)
25652608396773858801	436	98	96233 (90%)	99	85908 (87%)
24341655831718219991	414	97	101209 (95%)	97	88342 (90%)
23536477189379635045	436	99	95723 (90%)	99	86699 (88%)
24020887706891028596	436	98	99169 (93%)	100	90106 (91%)
25608332753867915599	436	100	95377 (89%)	98	84983 (86%)

Table 1: The results for montgomeryMultiplication during 100 modPow computations. Time is in nanoseconds, with the percentage this exponent took with respect to the mean timing of all exponents in parenthesis.

component of the Java Hotspot Virtual Machine on a toy password checker example, and the real world implementation of the cryptographic function modPow within the Java library. We were able to induce a side-channel in the password checker example, and showed promising results for the case of modPow. Possible future work includes a deeper investigation of the effects of the JIT compiler on complex code blocks such as those found in modPow, and other cryptographic functions within the Java library. Our work shows that JIT-like optimizations in runtime systems can potentially induce a side-channel, or at least affect the strength of existing side-channels.

References

[1] Timos Antopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for k-safety. 2017.

[2] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[3] Jia Chen, Yu Feng, and Isil Dillig. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 875–890. ACM, 2017.

[4] Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 286–296. ACM, 2007.

[5] Pedro Malagón, Juan-Mariano de Goyeneche, Marina Zapater, Zorana Bankovic, José M Moya, and David Fraga. Effects of compiler optimizations on side-channel attacks.

[6] Dan Page. A note on side-channels resulting from dynamic compilation. *IACR Cryptology ePrint Archive*, 2006:349, 2006.

[7] Quoc-Sang Phan, Lucas Bang, Corina S Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of adaptive side-channel attacks. In *Computer Security Foundations Symposium (CSF), 2017 IEEE 30th*, pages 328–342. IEEE, 2017.

Appendices

Appendix A Java BigInteger modPow Source code

Listing 1: Java listing

```java
/**
 * Returns a BigInteger whose value is
 * <tt>(this<sup>exponent</sup> mod m)</tt>. (Unlike {@code pow}, this
 * method permits negative exponents.)
 *
 * @param exponent the exponent.
 * @param m the modulus.
 * @return <tt>this<sup>exponent</sup> mod m</tt>
 * @throws ArithmeticException {@code m} &le; 0 or the exponent is
 *         negative and this BigInteger is not <i>relatively
 *         prime</i> to {@code m}.
 * @see    #modInverse
 */
public BigInteger modPow(BigInteger exponent, BigInteger m) {
    if (m.signum <= 0)
        throw new ArithmeticException("BigInteger: modulus not positive");

    // Trivial cases
    if (exponent.signum == 0)
        return (m.equals(ONE) ? ZERO : ONE);

    if (this.equals(ONE))
        return (m.equals(ONE) ? ZERO : ONE);

    if (this.equals(ZERO) && exponent.signum >= 0)
        return ZERO;

    if (this.equals(negConst[1]) && (!exponent.testBit(0)))
        return (m.equals(ONE) ? ZERO : ONE);

    boolean invertResult;
    if ((invertResult = (exponent.signum < 0)))
        exponent = exponent.negate();

    BigInteger base = (this.signum < 0 || this.compareTo(m) >= 0
                       ? this.mod(m) : this);
```

Listing 1 (Cont.): Java listing

```java
        BigInteger result;
        if (m.testBit(0)) { // odd modulus
            result = base.oddModPow(exponent, m);
        } else {
            /*
             * Even modulus. Tear it into an "odd part" (m1) and power of two
             * (m2), exponentiate mod m1, manually exponentiate mod m2, and
             * use Chinese Remainder Theorem to combine results.
             */

            // Tear m apart into odd part (m1) and power of 2 (m2)
            int p = m.getLowestSetBit(); // Max pow of 2 that divides m

            BigInteger m1 = m.shiftRight(p); // m/2**p
            BigInteger m2 = ONE.shiftLeft(p); // 2**p

            // Calculate new base from m1
            BigInteger base2 = (this.signum < 0 || this.compareTo(m1) >= 0
                                ? this.mod(m1) : this);

            // Caculate (base ** exponent) mod m1.
            BigInteger a1 = (m1.equals(ONE) ? ZERO :
                             base2.oddModPow(exponent, m1));

            // Calculate (this ** exponent) mod m2
            BigInteger a2 = base.modPow2(exponent, p);

            // Combine results using Chinese Remainder Theorem
            BigInteger y1 = m2.modInverse(m1);
            BigInteger y2 = m1.modInverse(m2);

            if (m.mag.length < MAX_MAG_LENGTH / 2) {
                result = a1.multiply(m2).multiply(y1).add(a2.multiply(m1).multiply(y2)).mod(m);
            } else {
                MutableBigInteger t1 = new MutableBigInteger();
                new MutableBigInteger(a1.multiply(m2)).multiply(new MutableBigInteger(y1), t1);
                MutableBigInteger t2 = new MutableBigInteger();
                new MutableBigInteger(a2.multiply(m1)).multiply(new MutableBigInteger(y2), t2);
                t1.add(t2);
                MutableBigInteger q = new MutableBigInteger();
```

Listing 1 (Cont.): Java listing

```java
                result = t1.divide(new MutableBigInteger(m), q).
                    toBigInteger();
            }
        }

        return (invertResult ? result.modInverse(m) : result);
    }

    /**
     * Returns a BigInteger whose value is x to the power of y mod
         z.
     * Assumes: z is odd && x < z.
     */
    private BigInteger oddModPow(BigInteger y, BigInteger z) {
    /*
     * The algorithm is adapted from Colin Plumb's C library.
     *
     * The window algorithm:
     * The idea is to keep a running product of b1 = n^(high-order
         bits of exp)
     * and then keep appending exponent bits to it. The following
         patterns
     * apply to a 3-bit window (k = 3):
     * To append  0: square
     * To append  1: square, multiply by n^1
     * To append 10: square, multiply by n^1, square
     * To append 11: square, square, multiply by n^3
     * To append 100: square, multiply by n^1, square, square
     * To append 101: square, square, square, multiply by n^5
     * To append 110: square, square, multiply by n^3, square
     * To append 111: square, square, square, multiply by n^7
     *
     * Since each pattern involves only one multiply, the longer
         the pattern
     * the better, except that a 0 (no multiplies) can be appended
         directly.
     * We precompute a table of odd powers of n, up to 2^k, and can
          then
     * multiply k bits of exponent at a time. Actually, assuming
         random
     * exponents, there is on average one zero bit between needs to
     * multiply (1/2 of the time there's none, 1/4 of the time
         there's 1,
     * 1/8 of the time, there's 2, 1/32 of the time, there's 3, etc
         .), so
     * you have to do one multiply per k+1 bits of exponent.
     *
```

Listing 1 (Cont.): Java listing

```java
     * The loop walks down the exponent, squaring the result buffer as
     * it goes. There is a wbits+1 bit lookahead buffer, buf, that is
     * filled with the upcoming exponent bits. (What is read after the
     * end of the exponent is unimportant, but it is filled with zero here.)
     * When the most-significant bit of this buffer becomes set, i.e.
     * (buf & tblmask) != 0, we have to decide what pattern to multiply
     * by, and when to do it. We decide, remember to do it in future
     * after a suitable number of squarings have passed (e.g. a pattern
     * of "100" in the buffer requires that we multiply by n^1 immediately;
     * a pattern of "110" calls for multiplying by n^3 after one more
     * squaring), clear the buffer, and continue.
     *
     * When we start, there is one more optimization: the result buffer
     * is implcitly one, so squaring it or multiplying by it can be
     * optimized away. Further, if we start with a pattern like "100"
     * in the lookahead window, rather than placing n into the buffer
     * and then starting to square it, we have already computed n^2
     * to compute the odd-powers table, so we can place that into
     * the buffer and save a squaring.
     *
     * This means that if you have a k-bit window, to compute n^z,
     * where z is the high k bits of the exponent, 1/2 of the time
     * it requires no squarings. 1/4 of the time, it requires 1
     * squaring, ... 1/2^(k-1) of the time, it reqires k-2 squarings.
     * And the remaining 1/2^(k-1) of the time, the top k bits are a
     * 1 followed by k-1 0 bits, so it again only requires k-2
     * squarings, not k-1. The average of these is 1. Add that
     * to the one squaring we have to do to compute the table,
     * and you'll see that a k-bit window saves k-2 squarings
     * as well as reducing the multiplies. (It actually doesn't
     * hurt in the case k = 1, either.)
     */
```

Listing 1 (Cont.): Java listing

```java
        // Special case for exponent of one
        if (y.equals(ONE))
            return this;

        // Special case for base of zero
        if (signum == 0)
            return ZERO;

        int[] base = mag.clone();
        int[] exp = y.mag;
        int[] mod = z.mag;
        int modLen = mod.length;

        // Make modLen even. It is conventional to use a cryptographic
        // modulus that is 512, 768, 1024, or 2048 bits, so this code
        // will not normally be executed. However, it is necessary for
        // the correct functioning of the HotSpot intrinsics.
        if ((modLen & 1) != 0) {
            int[] x = new int[modLen + 1];
            System.arraycopy(mod, 0, x, 1, modLen);
            mod = x;
            modLen++;
        }

        // Select an appropriate window size
        int wbits = 0;
        int ebits = bitLength(exp, exp.length);
        // if exponent is 65537 (0x10001), use minimum window size
        if ((ebits != 17) || (exp[0] != 65537)) {
            while (ebits > bnExpModThreshTable[wbits]) {
                wbits++;
            }
        }

        // Calculate appropriate table size
        int tblmask = 1 << wbits;

        // Allocate table for precomputed odd powers of base in Montgomery form
        int[][] table = new int[tblmask][];
        for (int i=0; i < tblmask; i++)
            table[i] = new int[modLen];

        // Compute the modular inverse of the least significant 64-
```

Listing 1 (Cont.): Java listing

```
            bit
        // digit of the modulus
        long n0 = (mod[modLen-1] & LONG_MASK) + ((mod[modLen-2] &
            LONG_MASK) << 32);
        long inv = -MutableBigInteger.inverseMod64(n0);

        // Convert base to Montgomery form
        int[] a = leftShift(base, base.length, modLen << 5);

        MutableBigInteger q = new MutableBigInteger(),
                          a2 = new MutableBigInteger(a),
                          b2 = new MutableBigInteger(mod);
        b2.normalize(); // MutableBigInteger.divide() assumes that
            its
                        // divisor is in normal form.

        MutableBigInteger r= a2.divide(b2, q);
        table[0] = r.toIntArray();

        // Pad table[0] with leading zeros so its length is at
            least modLen
        if (table[0].length < modLen) {
           int offset = modLen - table[0].length;
           int[] t2 = new int[modLen];
           System.arraycopy(table[0], 0, t2, offset, table[0].
               length);
           table[0] = t2;
        }

        // Set b to the square of the base
        int[] b = montgomerySquare(table[0], mod, modLen, inv, null
            );

        // Set t to high half of b
        int[] t = Arrays.copyOf(b, modLen);

        // Fill in the table with odd powers of the base
        for (int i=1; i < tblmask; i++) {
            table[i] = montgomeryMultiply(t, table[i-1], mod,
                modLen, inv, null);
        }

        // Pre load the window that slides over the exponent
        int bitpos = 1 << ((ebits-1) & (32-1));

        int buf = 0;
        int elen = exp.length;
```

Listing 1 (Cont.): Java listing

```
        int eIndex = 0;
        for (int i = 0; i <= wbits; i++) {
            buf = (buf << 1) | (((exp[eIndex] & bitpos) != 0)?1:0);
            bitpos >>>= 1;
            if (bitpos == 0) {
                eIndex++;
                bitpos = 1 << (32-1);
                elen--;
            }
        }

        int multpos = ebits;

        // The first iteration, which is hoisted out of the main
            loop
        ebits--;
        boolean isone = true;

        multpos = ebits - wbits;
        while ((buf & 1) == 0) {
            buf >>>= 1;
            multpos++;
        }

        int[] mult = table[buf >>> 1];

        buf = 0;
        if (multpos == ebits)
            isone = false;

        // The main loop
        while (true) {
            ebits--;
            // Advance the window
            buf <<= 1;

            if (elen != 0) {
                buf |= ((exp[eIndex] & bitpos) != 0) ? 1 : 0;
                bitpos >>>= 1;
                if (bitpos == 0) {
                    eIndex++;
                    bitpos = 1 << (32-1);
                    elen--;
                }
            }

            // Examine the window for pending multiplies
```

Listing 1 (Cont.): Java listing

```java
            if ((buf & tblmask) != 0) {
                multpos = ebits - wbits;
                while ((buf & 1) == 0) {
                    buf >>>= 1;
                    multpos++;
                }
                mult = table[buf >>> 1];
                buf = 0;
            }

            // Perform multiply
            if (ebits == multpos) {
                if (isone) {
                    b = mult.clone();
                    isone = false;
                } else {
                    t = b;
                    a = montgomeryMultiply(t, mult, mod, modLen, inv
                        , a);
                    t = a; a = b; b = t;
                }
            }

            // Check if done
            if (ebits == 0)
                break;

            // Square the input
            if (!isone) {
                t = b;
                a = montgomerySquare(t, mod, modLen, inv, a);
                t = a; a = b; b = t;
            }
        }

        // Convert result out of Montgomery form and return
        int[] t2 = new int[2*modLen];
        System.arraycopy(b, 0, t2, modLen, modLen);

        b = montReduce(t2, mod, modLen, (int)inv);

        t2 = Arrays.copyOf(b, modLen);

        return new BigInteger(1, t2);
    }

    /**
```

Listing 1 (Cont.): Java listing

```java
     * Returns a BigInteger whose value is (this ** exponent) mod
        (2**p)
     */
    private BigInteger modPow2(BigInteger exponent, int p) {
        /*
         * Perform exponentiation using repeated squaring trick,
             chopping off
         * high order bits as indicated by modulus.
         */
        BigInteger result = ONE;
        BigInteger baseToPow2 = this.mod2(p);
        int expOffset = 0;

        int limit = exponent.bitLength();

        if (this.testBit(0))
           limit = (p-1) < limit ? (p-1) : limit;

        while (expOffset < limit) {
            if (exponent.testBit(expOffset))
                result = result.multiply(baseToPow2).mod2(p);
            expOffset++;
            if (expOffset < limit)
                baseToPow2 = baseToPow2.square().mod2(p);
        }

        return result;
    }
```

```java
   // Montgomery multiplication. These are wrappers for
    // implMontgomeryXX routines which are expected to be replaced
        by
    // virtual machine intrinsics. We don't use the intrinsics for
    // very large operands: MONTGOMERY_INTRINSIC_THRESHOLD should
        be
    // larger than any reasonable crypto key.
    private static int[] montgomeryMultiply(int[] a, int[] b, int[]
         n, int len, long inv,
                                            int[] product) {
        implMontgomeryMultiplyChecks(a, b, n, len, product);
        if (len > MONTGOMERY_INTRINSIC_THRESHOLD) {
            // Very long argument: do not use an intrinsic
            product = multiplyToLen(a, len, b, len, product);
```

Listing 1 (Cont.): Java listing

```
        return montReduce(product, n, len, (int)inv);
    } else {
        return implMontgomeryMultiply(a, b, n, len, inv,
            materialize(product, len));
    }
}
private static int[] montgomerySquare(int[] a, int[] n, int len
     , long inv,
                                int[] product) {
    implMontgomeryMultiplyChecks(a, a, n, len, product);
    if (len > MONTGOMERY_INTRINSIC_THRESHOLD) {
        // Very long argument: do not use an intrinsic
        product = squareToLen(a, len, product);
        return montReduce(product, n, len, (int)inv);
    } else {
        return implMontgomerySquare(a, n, len, inv, materialize
            (product, len));
    }
}

// Range-check everything.
private static void implMontgomeryMultiplyChecks
    (int[] a, int[] b, int[] n, int len, int[] product) throws
      RuntimeException {
    if (len % 2 != 0) {
        throw new IllegalArgumentException("input array length
            must be even: " + len);
    }

    if (len < 1) {
        throw new IllegalArgumentException("invalid input
            length: " + len);
    }

    if (len > a.length ||
        len > b.length ||
        len > n.length ||
        (product != null && len > product.length)) {
        throw new IllegalArgumentException("input array length
            out of bound: " + len);
    }
}

// Make sure that the int array z (which is expected to contain
// the result of a Montgomery multiplication) is present and
// sufficiently large.
private static int[] materialize(int[] z, int len) {
```

Listing 1 (Cont.): Java listing

```
        if (z == null || z.length < len)
            z = new int[len];
        return z;
    }

    // These methods are intended to be be replaced by virtual
         machine
    // intrinsics.
    private static int[] implMontgomeryMultiply(int[] a, int[] b,
         int[] n, int len,
                                         long inv, int[] product) {
        product = multiplyToLen(a, len, b, len, product);
        return montReduce(product, n, len, (int)inv);
    }
    private static int[] implMontgomerySquare(int[] a, int[] n, int
          len,
                                       long inv, int[] product) {
        product = squareToLen(a, len, product);
        return montReduce(product, n, len, (int)inv);
    }



   /**
     * Montgomery reduce n, modulo mod. This reduces modulo mod and
           divides
     * by 2^(32*mlen). Adapted from Colin Plumb's C library.
     */
    private static int[] montReduce(int[] n, int[] mod, int mlen,
         int inv) {
        int c=0;
        int len = mlen;
        int offset=0;

        do {
            int nEnd = n[n.length-1-offset];
            int carry = mulAdd(n, mod, offset, mlen, inv * nEnd);
            c += addOne(n, offset, mlen, carry);
            offset++;
        } while (--len > 0);

        while (c > 0)
            c += subN(n, mod, mlen);

        while (intArrayCmpToLen(n, mod, mlen) >= 0)
            subN(n, mod, mlen);
```

Listing 1 (Cont.): Java listing

```
        return n;
    }
```

Appendix B BigInteger modPow Results

The following exponent values correspond to the respective columns from left to right:

- 25730899574802462604
- 24660523187409145475
- 33649042240140657826
- 28324482328545617634
- 21581371295657932221
- 25652608396773858801
- 24341655831718219991
- 23536477189379635045
- 24020887706891028596
- 25608332753867915599

java.lang.AbstractStringBuilder::append	99	100	100	100	99	98	99	100	99	100
java.lang.AbstractStringBuilder::ensureCapacityInternal	99	100	100	100	99	98	99	100	99	100
java.lang.Integer::numberOfLeadingZeros	42	49	53	47	100	42	99	47	36	42
java.lang.Math::min	100	100	100	100	100	100	100	100	100	100
java.lang.Number::<init>	99	100	99	100	100	99	100	100	98	100
java.lang.Object::<init>	100	100	100	100	100	100	100	100	100	100
java.lang.String::<init>	99	100	100	100	99	98	99	100	99	100
java.lang.String::charAt	100	100	100	100	100	100	100	100	100	100
java.lang.String::equals	100	100	100	100	100	100	100	100	100	100
java.lang.String::getChars	99	100	100	100	99	98	99	100	99	100
java.lang.String::hashCode	100	100	100	100	100	100	100	100	100	100
java.lang.String::indexOf	99	100	100	100	100	99	100	100	100	100
java.lang.String::length	100	100	100	100	100	100	100	100	100	100
java.lang.System::getSecurityManager	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::<init>	0	0	0	0	0	1	0	0	0	0
java.math.BigInteger::addOne	99	100	99	100	100	99	99	99	98	100
java.math.BigInteger::implMontgomeryMultiply	91	88	46	80	0	85	81	82	90	91
java.math.BigInteger::implMontgomeryMultiplyChecks	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::implMontgomerySquare	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::implMulAdd	99	100	99	100	100	99	100	100	99	100
java.math.BigInteger::implMulAddCheck	99	100	99	100	100	99	100	99	98	100
java.math.BigInteger::implSquareToLen	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::implSquareToLenChecks	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::intArrayCmpToLen	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::materialize	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::montReduce	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::montgomeryMultiply	99	100	98	99	0	98	97	99	98	100
java.math.BigInteger::montgomerySquare	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::mulAdd	99	100	99	100	100	99	100	99	98	100
java.math.BigInteger::multiplyToLen	99	100	99	100	100	98	99	99	98	100
java.math.BigInteger::primitiveLeftShift	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::squareToLen	100	100	100	100	100	100	100	100	100	100
java.util.Arrays::copyOfRange	99	100	100	100	99	98	99	100	99	100
Mean timing (ns)	96145	95067	98635	98545	106599	96233	101209	95723	99169	95377
Pct of max mean (%)	90	89	93	92	100	90	95	90	93	89

Figure 4: Even modulus, priming value = 21

java.lang.AbstractStringBuilder::append	100	99	99	100	98	100	99	99	100	100
java.lang.AbstractStringBuilder::ensureCapacityInternal	100	99	100	100	99	100	99	98	100	100
java.lang.Integer::numberOfLeadingZeros	40	40	40	53	98	40	99	43	36	48
java.lang.Math::min	100	100	100	100	100	100	100	100	100	100
java.lang.Number::<init>	99	100	99	100	99	99	100	99	100	100
java.lang.Object::<init>	100	100	100	100	100	100	100	100	100	100
java.lang.String::<init>	99	100	99	99	98	100	100	99	100	98
java.lang.String::charAt	100	100	100	100	100	100	100	100	100	100
java.lang.String::equals	100	100	100	100	100	100	100	100	100	100
java.lang.String::getChars	99	99	99	99	98	100	99	99	100	99
java.lang.String::hashCode	100	100	100	100	100	100	100	100	100	100
java.lang.String::indexOf	100	100	100	100	99	100	100	99	100	100
java.lang.String::length	100	100	100	100	100	100	100	100	100	100
java.lang.System::getSecurityManager	100	99	100	100	100	100	100	100	100	100
java.math.BigInteger::addOne	99	100	99	99	99	99	100	99	100	100
java.math.BigInteger::implMontgomeryMultiply	91	94	44	83	83	88	67	84	94	86
java.math.BigInteger::implMontgomeryMultiplyChecks	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::implMontgomerySquare	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::implMulAdd	99	100	99	100	99	99	100	99	100	100
java.math.BigInteger::implMulAddCheck	99	100	99	100	99	99	100	99	100	100
java.math.BigInteger::implSquareToLen	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::implSquareToLenChecks	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::intArrayCmpToLen	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::materialize	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::montReduce	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::montgomeryMultiply	99	100	96	98	94	99	97	99	100	98
java.math.BigInteger::montgomerySquare	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::mulAdd	99	100	99	100	99	99	100	99	100	100
java.math.BigInteger::multiplyToLen	99	100	99	98	98	99	99	99	100	98
java.math.BigInteger::primitiveLeftShift	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::squareToLen	100	100	100	100	100	100	100	100	100	100
java.util.Arrays::copyOfRange	99	100	99	99	98	100	100	99	100	98
Mean timing (ns)	85493	87333	87591	89117	98678	85908	88342	86699	90106	84983
Pct of max mean (%)	87	89	89	90	100	87	90	88	91	86

Figure 5: Even modulus, priming value = 22

Figure 6: Method compilation results for `BigInteger.modPow` in the case of an even modulus over 100 runs. The running time on the bottom is in nanoseconds.

java.lang.AbstractStringBuilder::append	99	100	99	100	100	98	100	99	100	100
java.lang.AbstractStringBuilder::ensureCapacityInternal	99	100	99	100	100	98	100	99	100	100
java.lang.Math::min	100	100	100	100	100	100	100	100	100	100
java.lang.Number::<init>	0	0	0	0	2	0	0	0	0	0
java.lang.Object::<init>	100	100	100	100	100	100	100	100	100	100
java.lang.String::<init>	99	100	99	100	99	98	100	99	100	100
java.lang.String::charAt	100	100	100	100	100	100	100	100	100	100
java.lang.String::equals	100	100	100	100	100	100	100	100	100	100
java.lang.String::getChars	99	100	99	100	99	97	100	99	100	100
java.lang.String::hashCode	100	100	100	100	100	100	100	100	100	100
java.lang.String::indexOf	100	100	100	100	100	99	100	100	100	100
java.lang.String::length	100	100	100	100	100	100	100	100	100	100
java.lang.System::getSecurityManager	99	100	100	99	100	100	100	100	99	99
java.math.BigInteger::addOne	100	99	100	100	100	99	99	100	100	100
java.math.BigInteger::implMontgomeryMultiply	1	3	0	1	0	2	4	0	1	0
java.math.BigInteger::implMontgomeryMultiplyChecks	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::implMontgomerySquare	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::implMulAdd	100	100	100	100	100	99	100	100	100	100
java.math.BigInteger::implMulAddCheck	100	99	100	100	100	99	99	100	100	100
java.math.BigInteger::implSquareToLen	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::implSquareToLenChecks	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::intArrayCmpToLen	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::materialize	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::montReduce	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::montgomeryMultiply	4	9	0	2	0	5	33	0	4	1
java.math.BigInteger::montgomerySquare	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::mulAdd	100	99	100	100	100	99	99	100	100	100
java.math.BigInteger::multiplyToLen	100	98	89	98	0	97	2	99	100	100
java.math.BigInteger::primitiveLeftShift	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::squareToLen	100	100	100	100	100	100	100	100	100	100
java.util.Arrays::copyOfRange	99	100	99	100	99	97	100	99	100	100
Mean timing (ns)	80941	81926	71057	81665	69231	80565	69864	78880	82994	80707
Pct of max mean (%)	98	99	86	98	83	97	84	95	100	97

Figure 7: Even odd, priming value = 21

java.lang.AbstractStringBuilder::append	100	100	100	100	98	100	100	100	100	98
java.lang.AbstractStringBuilder::ensureCapacityInternal	99	100	100	100	99	99	100	100	100	98
java.lang.Math::min	100	100	100	100	100	100	100	100	100	100
java.lang.Number::<init>	0	0	0	0	1	0	0	0	0	0
java.lang.Object::<init>	100	100	100	100	100	100	100	100	100	100
java.lang.String::<init>	99	98	100	100	99	100	100	100	100	99
java.lang.String::charAt	100	100	100	100	100	100	100	100	100	100
java.lang.String::equals	100	100	100	100	100	100	100	100	100	100
java.lang.String::getChars	100	98	100	100	98	100	100	100	100	98
java.lang.String::hashCode	100	100	100	100	100	100	100	100	100	100
java.lang.String::indexOf	100	100	100	100	100	100	100	100	100	100
java.lang.String::length	100	100	100	100	100	100	100	100	100	100
java.lang.System::getSecurityManager	100	99	100	100	99	100	99	100	100	100
java.math.BigInteger::addOne	100	98	98	99	99	98	100	100	100	99
java.math.BigInteger::implMontgomeryMultiply	0	0	0	0	26	1	0	0	3	2
java.math.BigInteger::implMontgomeryMultiplyChecks	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::implMontgomerySquare	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::implMulAdd	100	99	100	99	100	99	100	100	100	99
java.math.BigInteger::implMulAddCheck	100	99	99	99	100	99	100	100	100	99
java.math.BigInteger::implSquareToLen	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::implSquareToLenChecks	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::intArrayCmpToLen	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::materialize	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::montReduce	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::montgomeryMultiply	1	2	0	4	81	3	10	3	6	3
java.math.BigInteger::montgomerySquare	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::mulAdd	100	99	100	99	100	99	100	100	100	99
java.math.BigInteger::multiplyToLen	99	97	90	96	3	98	100	99	99	99
java.math.BigInteger::primitiveLeftShift	100	100	100	100	100	100	100	100	100	100
java.math.BigInteger::squareToLen	100	100	100	100	100	100	100	100	100	100
java.util.Arrays::copyOfRange	99	98	100	100	99	100	100	100	100	99
Mean timing (ns)	69815	69605	67897	65914	77826	68651	79685	69932	69765	69593
Pct of max mean (%)	88	87	85	83	98	86	100	88	88	87

Figure 8: Even odd, priming value = 22

Figure 9: Method compilation results for `BigInteger.modPow` in the case of an odd modulus over 100 runs. The running time on the bottom is in nanoseconds.