Secure Elliptic Curve Implementations: An Analysis of Resistance to Power-Attacks in a DSP Processor

Catherine H. Gebotys[1] and Robert J. Gebotys[2]

[1] Department of Electrical and Computer Engineering, University of Waterloo, Waterloo Ontario Canada N2L 3G1
cgebotys@optimal.vlsi.uwaterloo.ca

[2] Wilfrid Laurier University, Waterloo Ontario Canada

Abstract. With the popularity of wireless communication devices a growing new important dimension of embedded systems design is that of security. This paper presents exploration of power attack resistance, using a statistical approach for identifying regions of the power trace which pose a possible security threat. Unlike previous power analysis research, a new metric supporting small timing shifts and complex processor architectures is presented. This research helps to identify how to create secure implementations of software. Elliptic curve point multiplications using the Weierstrass curve and Jacobi form over 192-bit prime fields were implemented and analyzed. Over 60 real measured power traces of elliptic curve point multiplications running at 100MHz on a DSP VLIW processor core were analyzed. Modification of power traces through software design was performed to maximize resistance to power attacks in addition to improving energy dissipation and performance by 44% with a 31% increase in code size. This research is important for industry since efficient yet secure cryptography is crucial for wireless communication embedded system devices and future IP enabled smart cards.

1 Introduction

Security is increasingly becoming important in current design methodologies for embedded systems which concentrate on high performance, low cost, low power and low energy. Design for security involves secure protocol implementation and power analysis in addition to algorithm design. In fact power dissipation has a large impact on security as well as cost and reliability of an embedded system. Not only must cryptographic algorithms be high performance and low energy, but more importantly they must be secure or safe from side channel attacks. A side channel attack[1,11] involves obtaining useful information from the cryptographic application which may lead to the revelation of the secret key. Useful information includes the amount of time it takes to perform various operations or the variation of power dissipation during key computations. In the later case this is known as a power attack[1]. As an example, an attacker who has obtained the secret key is able to eavesdrop on a confidential wireless communication between two parties. Consider Mary, the victim, who wishes to encrypt a conversation with Bob. She first has to set up a session key. Mary computes the session key as: xP in elliptic curve cryptography (or y^x *mod n* in RSA technology) where x is the secret key and this computation is performed various times for different P's (or y's). The attacker may know P (or y) and in addition may obtain the computations times or may be able to monitor power dissipation. With this additional information the attacker may eventually be able to compute the secret key, x [1]. When the attacker has obtained the secret key, communication between Mary and Bob is not secure. Alternatively, obtaining the secret key in other applications, such as smart cards, allows one to illegally use phone or digital TV services. In power attacks, the dynamic power of the processor is measured over time and is called a power trace [1,11,12,13]. In elliptic curve cryptography (ECC), the analysis of the power trace may reveal when a point doubling occurs (or calculation of $2P$), and when two points are being added (such as $2P+P$) in the computation of xP, thus revealing the secret key.

In SOC (systems on a chip) platforms, multiple DSP processor cores on the same chip are common. Often these core processors run at different voltages, and use separate power pins, thus secure implementations of cryptography on SOCs is important. Cryptography on DSP processor cores is an important lower cost and lower power dissipation alternative to cryptographic processor cores and general purpose processor cores respectively. This paper for the first time presents a new metric for quantizing security and design exploration for ECC on a DSP processor core. Results are based upon over 60 real measured dynamic power traces performing point multiplication with different keys. Additionally tradeoffs in code size, performance, and energy dissipation for security are explored.

Currently research in power attacks of smart cards, have utilized general purpose processors [1,11,12,13]. Typically smart card applications are not time critical and energy dissipation is not a major concern since power is obtained from the card reader (or ATM machine, etc). Power attacks of more sophisticated processors with parallel instruction execution have not been reported in the literature. The measurement of power while a processor is executing an application (or a power trace) has been used in power-attacks of cryptographic devices, such as smart cards [1]. In particular the analysis of the variation of power, and computations on a number of power traces can be used to detect data and algorithmic dependencies [1]. This research studied the correlation of power variation with data values being manipulated and instruction sequencing. In the former case, known as differential power attacks, encryption applications were analyzed [1]. In the later case, known as a simple power attack (SPA), it was concluded that the correlation was significant and techniques such as random sequencing of instructions have since been investigated. Researchers have also investigated the use of DSP processors for encryption [2,5] as well as elliptic curve implementation [9], however their resistance to power attacks has not been addressed. Researchers addressing smart card application have suggested security against power attacks be achieved through 70% increase in computational cost [3], assuming a 192-bit prime number and signed window method with r=5[14], using 16 multiplications for both doubling and summing compared to 8 multiplications for a doubling and 16 for a summing. This is achieved through using different forms of the curve, such as the Jacobi form, where mathematically the sum and the double of a point use the same formula 3,8]. Other researchers have investigated the cubic form of the EC, known as the Hessian form of the curve [8,10]. In this research a 33% improvement in performance overheads is achieved, since only 12 multiplications are required for both summing and doubling.

This paper will present a new metric for analyzing implementation security against power attacks, the implementation security index (*ISI*), and design exploration of performance, energy dissipation, code size in addition to security. It will be demonstrated on a complex VLIW DSP processor core, the Star*core (SC140), developed by Motorola and Lucent[4]. An elliptic curve cryptographic application is analyzed for resistance to power-attacks trading off low energy dissipation, high performance, and small code size. The uncertainty of security from power-attacks is explored with real current measurements of the DSP hardware VLSI core in a chip. A previously suggested power-attack resistant technique, the Jacobi form of the EC is also implemented and analyzed for comparisons in implementation security.

2 Elliptic Curves and Software Implementation

This section will review the application, elliptic curve point multiplication in prime fields and introduce the methodology used to develop a security index for measuring resistance to power attacks. Prime field cryptography involves a significant number of integer multiplies which can be performed very efficiently on DSP cores. In addition to a chosen key length, there are many different fields, projective coordinates, and types of elliptic curves that can be implemented. For added security portable devices should be able to support numerous curves and fields. However it is important for the designer to be able to choose which sets to implement on an embedded device, to tradeoff performance, code size, energy dissipation, and security against power attacks. The application, point multiplication, will be introduced in this section, followed by a discussion of implementation methods.

Point multiplication was implemented using the Weierstrass equation of the elliptic curve and the Jacobi form of the elliptic curve. Implementation details are provided in appendix A, B respectively. All curves were implemented with 192-bit field operations, using prime polynomial $x^{192}-x^{64}-1$. The Weierstrass equation of the curve was $y^2 = x^3-3x^2+b$ over 192-bit prime fields[6] (see Appendix A for details). However to avoid the long latency of inversion in prime fields (Jacobi) projective coordinates [7,8] were chosen. In our DSP processor core the field multiplication to inversion ratio was 0.014 (330 cycles / 23146 cycles) using a worst case time for inversion. The equations for doubling and summing points in prime fields with Jacobi coordinates are given below[8]. These coordinates were used in the SC140 implementation and correspond to the cycle counts of the point double and sum for prime fields in table 1. Given point $P=(x,y,z)$, the point $2P = (x_{12},y_{12},z_{12})$ is given below (point doubling) and the point $(x_1,y_1,z_1)+(x_2,y_2,z_2) = (x_3,y_3,z_3)$ is also given below for point summing.

$$x_{12} = 9(x^2 - z^4)^2 - 8y^2x$$
$$y_{12} = 3(x^2 - z^4)(4y^2x - (3x^2 + z^4) + 8y^2x) - 8y^4$$
$$z_{12} = 2yz$$
$$x_3 = (y_2(z_1)^3 - y_1(z_2)^3)^2 - (x_2(z_1)^2 + x_1(z_2)^2)(x_2(z_1)^2 - x_1(z_2)^2)^2$$
$$y_3 = (y_2(z_1)^3 - y_1(z_2)^3)(x_1(z_2)^2(x_2(z_1)^2 - x_1(z_2)^2)^2 - x_3)$$
$$- y_1(z_2)^3(x_2(z_1)^2 - x_1(z_2)^2)^3$$
$$z_3 = z_1z_2(x_2(z_1)^2 - x_1(z_2)^2)$$

The clock cycle counts for the prime field EC codes running on SC140 along with the code size and average power measurements are given in table 1. The first row of table 1 provides clock cycle counts for point multiplication using key $13 (where $ indicates hexadecimal notation), for illustration purposes, with a signed NAF implementation (no windowing). Using these codes with a 192-bit key point multiplication can be performed in under 3ms running the DSP processor core at 300MHz (assuming a signed NAF sliding window with r=5 as in [3]).

Table 1. Original Prime Field Code Implementation Characteristics on SC140

192-bit Prime Fields	Clock Cycles	Code Size(bytes)	Average Power (mA)
Point Operations			
Point Multiplication	28,897	8,070	48.8
(k=$13)	3,177	5,008	47.9
Double	5,554	4,920	49.8
Sum			
Field Operations			
Multiplication	330	1,270	49.6
Squaring	213	2,212	51.6
Mod reduction	60	460	45.9
Addition	33	320	45
Subtraction	29	248	43.8

The modification of *sum* and *double* routines for security against power attacks was explored by inserting redundant operations into the *double* and *sum* routines so that the order and type of field operation were identical. Figure 1 illustrates the sum and double routine modifications. Since the point summation was almost double the execution time of the point double routine (see Table 1 cycle counts), it was split into two routines, *sum1* and *sum2* shown in each column in figure 1. Redundant Operations are identified by __ or underscores preceding their operation in the table. Table 2 illustrates the original and modified EC codes with respect to the field operation counts. The shifts are implemented for coefficient multiplications (ie. multiplies by 2,4,8). Approximately 25 clock cycles are required on average to implement a shift of x number of bits (where x = 2,4,8). The only field operations which had variable clock cycle counts were the modular reductions which may or may not be required after additions, subtractions, or shifts. Also the modular reductions for the result of squarings and

Double	**Sum1**	**Sum2**
b1 = y ^2	**z2s = z2 ^2**	**hs = h ^2**
e1 = z ^2	**z1s = z1 ^2**	**om = j ^2**
b2 = b1 * b1	**z2c = z2s * z2**	**al = th * hs**
b = b2<<3	**__al=y2<<3**	**__th=y2<<3**
__z2=y2 * x2	**f = z1s * x2**	**la = h * hs**
z31 = y * z	**g = z2s * x1**	**th = la * i**
e2 = x - e1	**___th=x1-z1s**	**x3 = om - al**
e3 = x + e1	**___ga=x1+z1s**	**___be=al + om**
e = e2 * e3	**z1c = z1s * z1**	**z3 = ga * h**
z12 = z31<<1	**___om=g<<1**	**___al=hs<<1**
c1 = e<<1	**___ga=z1c<<1**	**___be=z3<<1**
c = c1 + e	**th = f + g**	**___om=be+z3**
f1 = b1<<2	**___al=z2s<<2**	**___ga=hs<<2**
a = f1 * x	**ga = z1 * z2**	**la = hs * g**
f3 = a<<1	**___la=ga<<1**	**___al=la<<1**
d1 = c * c	**i = y1 * z2c**	**___be=om*om**
x12 = d1 - f3	**___al=i-la**	**om = la –x3**
y31 = a - x3	**h = f - g**	**___ga=om-al**
y32 = y31 * c	**om = y2 * z1c**	**al = om * j**
y12 = y32 - b	**j = om - i**	**y3 = al - th**

Fig. 1. PA-resistant code, *WR*, for point doubling(1st column)

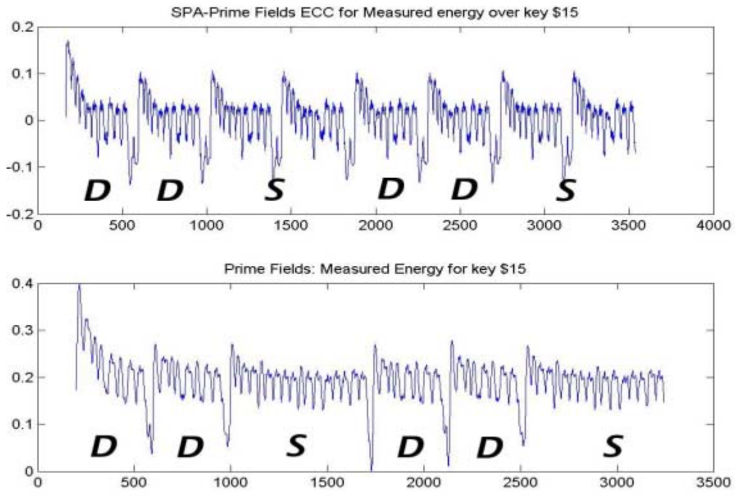

Fig. 2. Comparison of power traces of original code at bottom with power-attack resistant code WR1 for the same key at top

multiplications were variable (since the final summation may require one or two subtractions of the prime polynomial [7]). These caused minor delay differences in our codes. Additionally the signed NAF higher level algorithms were designed so that the routines in between the *sum1, sum2,* and *double* were also identical (ie. so one could not distinguish *sum* followed by *double* or *double* followed by *sum,* or *double* followed by *double*).

The direct implementation of the Weierstrass Curve with Jacobi projective coordinates has a very low value of implemented security (or low resistance to power attacks) on the DSP processor. This is illustrated in the power trace shown at the bottom (2nd,3rd column) of figure 2, where S is the *sum* routine and D is the *double* routine. The power trace at the bottom, or original point multiplication, shows 8 bumps for the double routine, where each bump is the power dissipation of a multiplication (field) operation or a squaring (field) operation (narrower than the multiplication). This rise in power dissipation is due to the higher instruction level parallelism in the code as well as the use of higher power dissipating types of instructions, such as a 32X32 bit integer multiplication. Since the point summation is almost double the execution time of a point double, the power traces can easily be used to obtain the secret key and thus are very vulnerable to power attacks on this VLIW DSP processor. The implementation with redundant operations is shown in the power trace at the top of figure 2. The top power trace used an extra field and point summing multiplication at the beginning of each double and sum's routines to prevent the compiler from eliminating dual input operands to the double routine (however a less costly shift field operations could have been used). The two power traces in figure 2 perform point multiplication on the same key but the top is far more resistant to simple power attacks.

3 Analysis of Uncertainty in Power-Attack Resistance

The power traces for several keys were obtained for the software implementations of the Weierstrass curve(*WR*) and the Jacobi curve (*JC*) running on the SC140 DSP processor. In *WR,* the code was further optimized replacing redundant operations with more efficient implementations and detailed assembly modification to ensure good cycle-accurate timings of the *sum* routines and *double* routine. All power traces were obtained by executing the cryptographic algorithms on the SC140 at 100MHz (for illustration purposes, though power traces at 300MHz were very similar), using a pattern generator and high speed oscilloscope to capture the power traces. In the power trace plots the y-axis represents the current variation (in Amps centered by the oscilloscope at zero, and amplified by the probe by a factor of 5) and the x-axis represents the time (currently sampled data points). Matlab was used for signal analysis of the power waveforms.

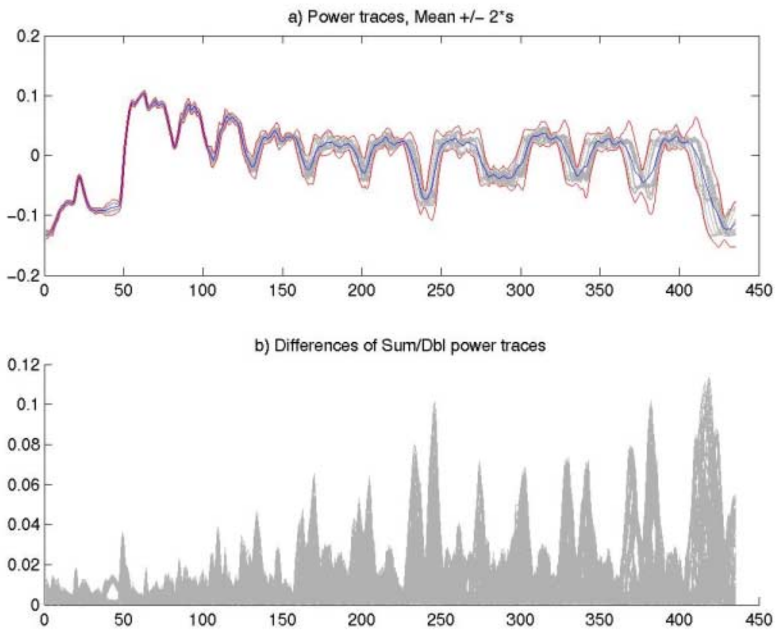

Fig. 3. Top: means, variances; Bottom: differences of *WR1* sum/dbl power traces

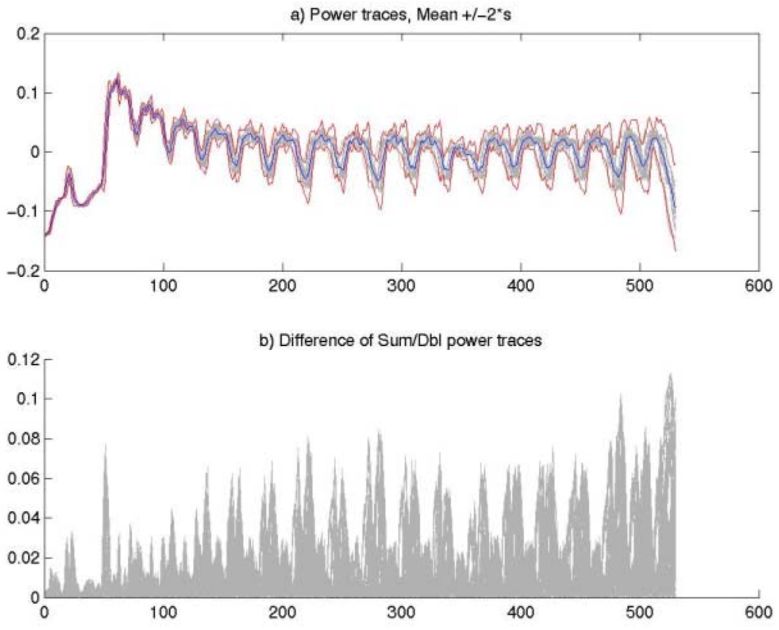

Fig. 4. Top: means, variances; Bottom: differences of *JC* sum/dbl power traces

Table 2. Field operation counts for original Weierstrass EC code and modified EC code for power-attack resistance

	Original	PA-resistant Code (*WR*)
Sum		
# of Multiplications	13	14
# of shifts	4	10
# of Squarings	0	4
# of Additions/subtractions	6	12
Double		
# of Multiplications	4	7
# of shifts	5	5
# of Squarings	4	2
# of Additions/subtractions	6	6
Cycles for key $0b	29,517	37,294
CodeSize	7,626	9,618

Initially code was implemented by modifying the code in figure 1, by specifically removing the first redundant shift by 3 bits (or <<3) in the sum routines and replacing b2<<3 by the first redundant multiplication (z2=y2*x2 becomes b=b2*8) in the double routine to reduce the latency. Power traces of this code, *WR1*, were extracted and the variances and the mean plus or minus two times the standard deviation were computed for *sum1, sum2, double* and plotted in figure 3. It was compared to the variance and standard deviation of power traces obtained from the Jacobi curve, *JC*, (which is mathematically power-attack resistant, since the point doubling and summing use the same routine) shown in figure 4. The superimposed Jacobian power traces in figure 4 each show 17 power bumps, all for multiplications except the last two which are squarings (see Appendix B). The average variances of both *double* and *sum* power traces were *3.12E-4* for the *Weierstrass curve, WR1,* and *1.911E-4* for the *Jacobi curve, JC.*

However it is difficult to use this average variance or absolute differences of double and sum power traces (as shown at the bottom of both figures) to determine which part of the code needs to be modified to increase security. Note also in these figures that the differences of the sum and double routines shown in the bottom plot of the figures are high where the power traces have the highest slopes (due to timing shift effects near the rise and fall of power dissipation during integer multiplication or squaring routines), thus not providing much information.

To further analyze the power traces and identify regions of software implementation which were possibly insecure, the variances and means were integrated into an implementation security index (*ISI*), shown in equation (1).

$$ISI_{1,2}(t) = \left(\frac{(\bar{x}_1(t) - \bar{x}_2(t))}{\sqrt{\frac{(s_1(t))^2}{n_1} + \frac{(s_2(t))^2}{n_2}}} \right)^{-1} \qquad (1)$$

Equation (1) utilizes the means, $\bar{x}_1(t), \bar{x}_2(t)$ and standard deviations $s_1(t), s_2(t)$ for each time t and the number of power subtrace samples n_1, n_2. This formula could be used to analyze differences in the power traces of the *sum* (sample set 1) and the *double* (sample set 2) routines representing $ISI_{S,D}(t)$, or it could be used to analyze larger portions of the power trace including software implementations in between the *sum* and *double* routines. For example, a subtrace sample could be any part of the power trace such as a *double* followed by a *double*, DD, (or *sum1* followed by a *sum2*, SS) representing the $ISI_{DD,SS}(t)$ formula. Using this formula, areas of the power trace where the variance was low and the means of the *double* and *sum* routines differed were identified as a low security measure. In other cases if the differences of means were small or variances were large then a high security measure is indicated by the statistic.

Figure 5 shows four plots of *WR1*, from top to bottom the variances (Vars), the $ISI_{S,D}(t)^{-1}$ variable values (or ISI where peaks or valley's indicate a security problem), the difference of means (or DPA), and the actual sum, double means in the last bottom plot (Means). The oval identifies a peak of $ISI_{S,D}(190)^{-1}$ (in the *x* value near 190). This peak identified a problem in the security of the software implementation of *WR1*, near the second multiply. This problem was verified (see bottom mean plot) as a power difference due to multiplying a number with a large number of zeros (192-bit number = $08) in the double routine and not in the sum routine (where a normal multiplication of two more random 192-bit numbers is performed). In the DSP processor, multiplication with zero's dissipates larger power than a random or all 1's number (due to precharged busses, etc). Note that the difference of means or DPA (plot below the $ISI_{S,D}(t)^{-1}$) did not pick up any significant difference relative to the other regions of the plot. The $ISI_{S,D}(t)^{-1}$ measure shows more variation over earlier t regions than the DPA due to extremely small variances, since these subtraces were aligned at the beginning.

The modified code, called *WR*, removed the **8* from the double routine and replaced it with a redundant multiply of two full 192-bit numbers (whose result is put in a temporary variable), and introduced shifts to accomplish the **8* functionality, as detailed in figure 1. The power trace of the modified code is shown in figure 6, where the mean and variance and differences are plotted. In this figure the *sum2-double* power subtraces are shown. A horizontal line is added for comparison of field operations in both routines. The $ISI_{SD,DS}(t)^{-1}$ for the resulting code, *WR*, is shown in figure 7 and it has an average *ISI* (or mean of $|ISI_{SD,DS}(t)|$) of *0.49*. The peak circled in figure 7 of $ISI_{SD,DS}(97)^{-1}$ indicates a security leak.

It is interesting to note that the difference of means (DPA) also has peaked indicating a possible security leak. However DPA continues to peak through the rest of the power traces in figure 7, whereas the $ISI_{SD,DS}(t)^{-1}$ flattens out. Peaking through the DPA indicates differences of means exist, however they are not significant according to $ISI_{SD,DS}(t)^{-1}$ since the variances are also large in these areas. Again these large variances and difference of means occur due to timing shifts in the power traces. For example consider the DPA peak and valley centered near t=150. These result from the timing shifts in the rise and fall of power dissipation of the 192-bit multiplication routine. In this example, since the rise or fall of power is large, any timing shift will

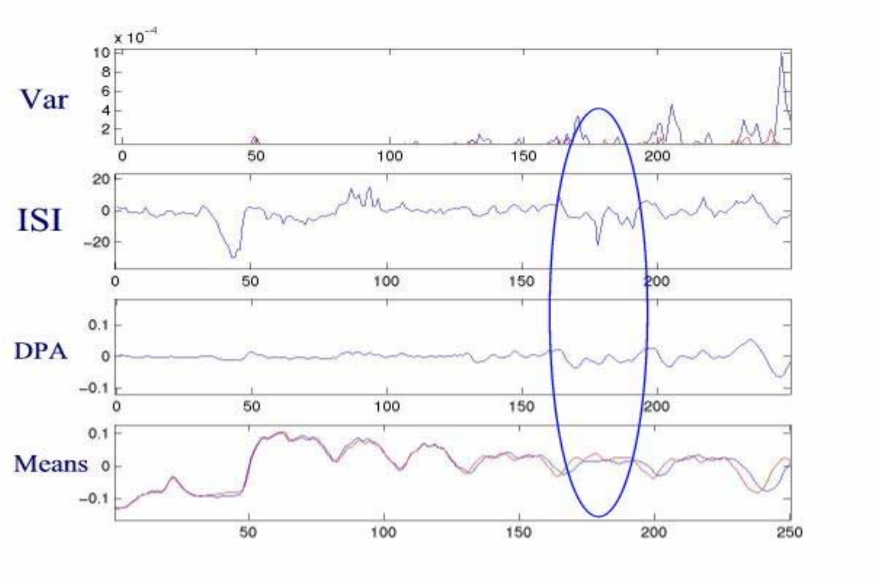

Fig. 5. Variances, $ISI_{S,D}(t)^{-1}$, DPA, and means of *WR1*

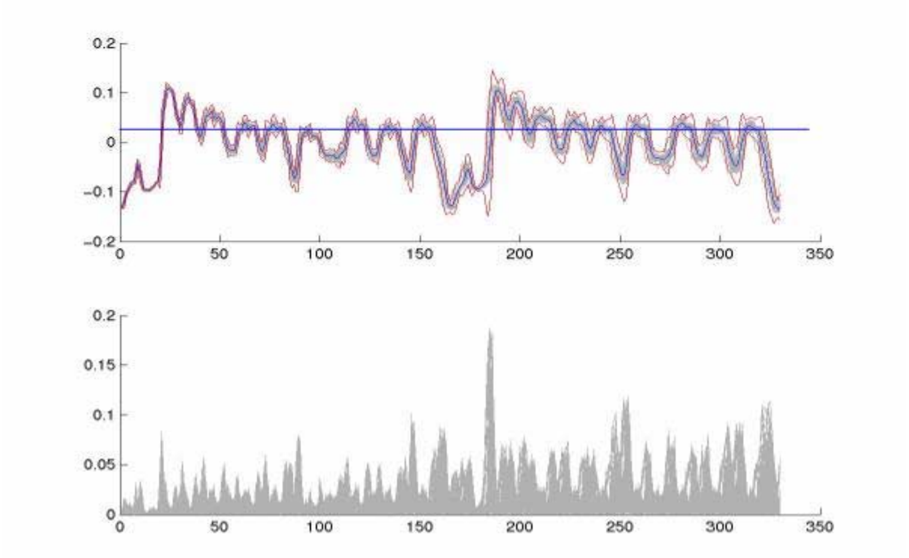

Fig. 6. Top: means, variances; Bottom: differences of sum/double subtraces of *WR*

create a large difference of means which is picked up by DPA methods. However in ISI since the timing shifts produce a large variance over the many power subtraces, no peak or security leak is correctly identified. Hence $ISI_{SD,DS}(t)^{-1}$ tends to identify which

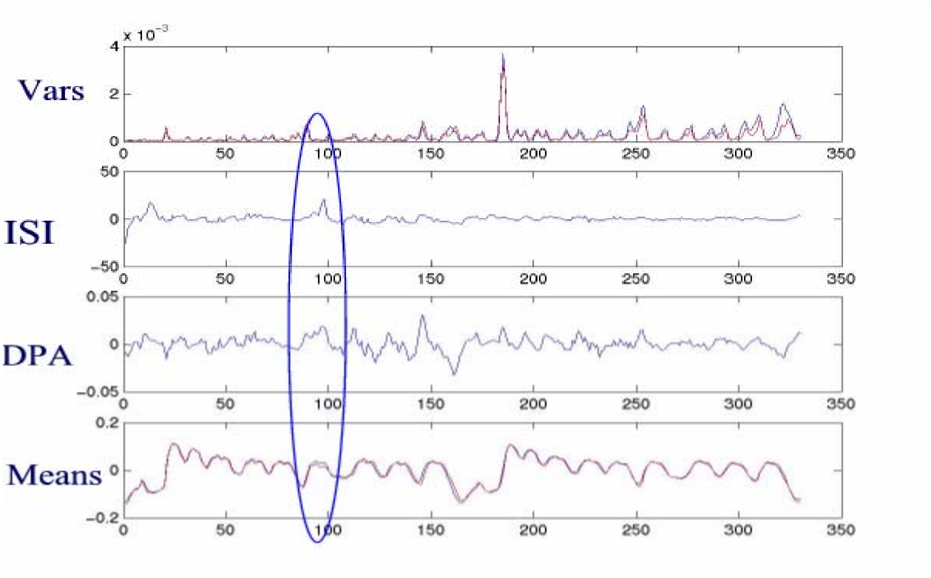

Fig. 7. Variances, $ISI_{SD,DS}(t)^{-1}$, DPA, and means of double-double/sum-sum power traces of *WR*

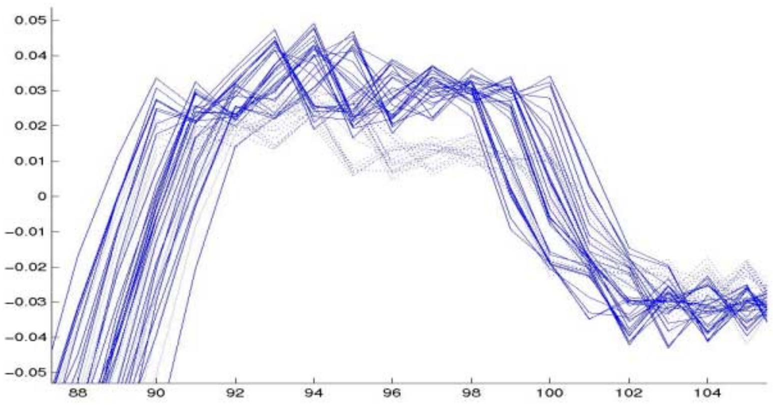

Fig. 8. Hole produced by memory stalls in highly parallel section of integer multiplication

difference of means are in fact significant (or exist with small variances). However the user must still verify that the $ISI_{SD,DS}(t)^{-1}$ peak is not a result of strictly very small variances alone. Looking closer at this $ISI_{SD,DS}(97)^{-1}$ peak, figure 8 shows superimposed power traces over this region and the *hole* or gap between the *sum2* and *double (and sum1)* routines (see x-axis between 96 and 98). The peak indicates a valid difference between the double and sum2 power traces due to memory stalls within the highly parallel integer multiplication routine.

In the SC140 a memory stall will cause an extra one cycle delay whose power characteristic is similar to a nop instruction (or no-operation instruction) which has the lowest power dissipation. In SC140, memory stalls will occur whenever more than one request to the same memory module but a different row is made in the same processor clock cycle. Since the DSP processor uses a unified memory space (where both program and data is stored in same memory), memory stall identification is more complex since it involves the program code as well. In this case, it was determined that in the second sum routine, memory stalls due to conflicts between the program and the loading of data in the middle of a loop with high instruction level parallelism caused the power not to peak as much as it did in the other *sum1* routine, or *double* routine. The lower dotted traces in figure 8 are the *sum2* routine, while the upper line traces are the *sum1* and *double* routines respectively.

The Jacobi form of the curve was also analyzed and the variances, $ISI_{S,D}(t)^{-1}$ variables, and means are shown in figure 9. The final *JC* code had an average *ISI* of *0.44*. Figures 7 and 9 both show high absolute values of $ISI_{S,D}(t)^{-1}$ near the beginning of the power traces and in some other areas. In these cases, the absolute peak occurs due to very low variances and does not represent a security problem.

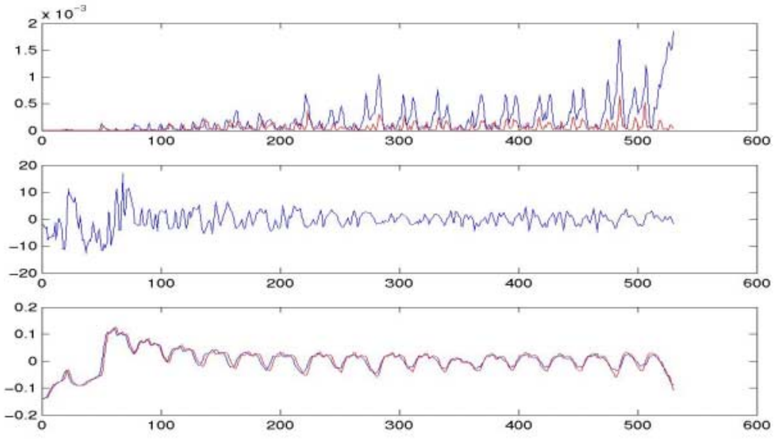

Fig. 9. Variances, $ISI_{S,D}(t)^{-1}$, means of Jacobi form of curve, *JC*

The bar chart in figure 10 from left to right shows the energy dissipation (mJ) of *WR, JC* and clock cycle counts for *WR, JC* for each key (ie. \$b, \$10, and a random 192-bit key). The number of clock cycles for key \$b and \$10 have been divided by 100 in the figure for scaling purposes. In figure 10, the cycle count of the 192-bit key is divided by 10,000 (1.81M cycles for *JC* and 1.25M cycles for *WR*, a 44.7% improvement) and the energy of the 192-bit key is divided by 100. The average *ISI* of *WR* and *JC* is respectively 0.49 and 0.44. The energy per bit (energy dissipation of 192-bit key divided by 192) of the *WR* and *JC* implementations is 10.37 and 14.96 respectively (a 44.2% improvement). The code size of *WR* is 9618 bytes compared to *JC* which only requires 7338 bytes (since the *double* and *sum* are the same routine).

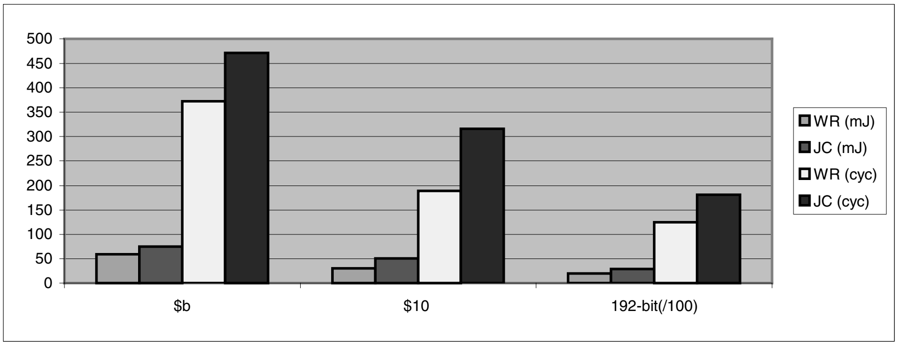

Fig. 10. Energy dissipation and cycle comparison of *WR* with *JC*

4 Discussions and Conclusions

The methodology presented in this paper, has shown that *ISI* can provide important information for cryptographic applications being implemented by embedded system designers. Previous methods suggested, such as simple power attacks, or differencing can be improved by exploring variances and *ISI*. This design exploration has been used to develop code which has improved security yet lower energy dissipation and higher performance compared to the Jacobi curve implementation. The lower energy dissipation will be important for secure implementations in portable devices.

Unlike previous research mathematical approaches to power-attack resistance, this research has examined techniques for ensuring the security of the software implementation through modification of power and energy dissipation. Design exploration of verified elliptic curve point multiplication routines running on a complex VLIW DSP processor core is presented. Previous methods suggested, such as SPA, or DPA (not easily extended for complex architectures, since there are multiple active busses each clock cycle) can be improved by exploring variances and *ISI* which handle small timing shifts. For the first time, a new metric, the implementation security index, *ISI*, has been introduced for quantizing security of implementations. Real power traces have been measured, and security from power-attacks verified with real hardware VLSI chip power measurements. This methodology for the design of secure software for the SC140 DSP processor can in general be applied to other processors.

Results show that *WR* code improves energy dissipation, performance, and implementation security index by 1.44 times, 1.44 times, and 1.11 times respectively compared to our implementation of previously research routines, *JC*, with a 31% increase in code size. This metric can be used for design exploration of security in addition to performance, code size and energy dissipation. This research is crucial for supporting a methodology for designing software that is not only optimized for performance, power and cost, but also for implementation security.

Acknowledgments. The author would like to thank NSERC, Motorola, and CITO for their support through funding of this research. The author also would like to thank R.Muresan for his acquisition of the power traces using [15] and A.Sathiananthan for coding the cryptographic algorithms and running them on the Star*Core board.

References

1. P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems", LNCS 1109, 1996.
2. S. Dusse, B. Kaliski, "A cryptographic library for the Motorola DSP56000", vol 473, LNCS, May 1990, pp. 230–244.
3. P. Liardet, N. Smart, "Preventing SPA/DPA in ECC systems using the Jacobi Form", LNCS 2162, May 2001, pp 391–401.
4. "Star*Core 140 DSP Core Reference Manual", Motorola/Lucent, Sept 1999.
5. T. Wollinger, M. Wang, J. Guajardo, C. Paar, "How well areHigh End DSPs Suited for the AES Algorithms?"
6. IEEE Std 1363-2000, IEEE Standard specifications for public-key cryptography, IEEE computer Soc. 2000.
7. D. Hankerson, J. Hernandez, A. Menezes, "Software Implementation of NIST Elliptic Curves over Prime Fields", White Paper, www.certicom.com, 2000
8. Chudnovsky, D.V., G.V. Chudnovsky, "Sequences of Numbers generated by addition in formal groups and new primality and factorization tests", Applied Mathematics, Vol.7, pp 385–434, 1986.
9. K. Itoh, M. Takenaka, N. Torii, S. Temma, Y. Kurihara, "Fast implementation of public-key cryptography on a DSP TMS320C6201", CHES '99, vol 1717, LNCS, 1999, pp. 61–72.
10. M. Joye, J. Quisquater, "Hessian Elliptic Curves and Side-Channel Attacks" LNCS 2162, May 2001, pp 402–410.
11. P. Kocher, J. Jaffe, B. Jun, "Differential Power Analysis" Crypto'99, LNCS 1666, 1999.
12. T. Messerges, E. Dabbish, R. Sloan, "Investigations of Power analysis attacks on Smartcards" USENIX workshop on Smartcard Technology, 1999.
13. O. Kommerling, M. Kuhn, "Design principles for Tamper-resistant Smartcard Processors", USENIX workshop on Smartcard Technology, 1999.
14. I. Blake, G. Seroussi, N. Smart, "Elliptic Curves in Cryptography" , LMS 265, Cambridge Univ. Press, 2000
15. R. Muresan, C. Gebotys, "Current consumption dynamics at instruction and program level for a VLIW DSP Processor", ACM Proc. of ISSS, Oct 2001, pp. 130–135.

Appendix: A

The Weierstrass model [8,15] used was $\mathbf{E : y^2 \equiv x^3 - 3x + b \ (mod\ p)}$ where (hexadecimal notation is denoted by 0x)
b=0x64210519e59c80e7 0fa7e9ab 72243049 feb8deec c146b9b1

p= 0xfffffffffffffffffffffffffffffffeffffffffffffffff (or

6277101735386680763835789423207666416083908700390324961279 or

x^{192}-x^{64}-1) using (Jacobian) projective coordinates (where $x=x/z^2$,$y=y/z^3$) and the following NIST recommended x,y starting points and:

x = 0x188da80e,0xb03090f6,0x7cbf20eb,0x43a18800,0xf4ff0afd,0x82ff1012
y = 0x07192b95,0xffc8da78,0x631011ed,0x6b24cdd5,0x73f977a1,0x1e794811
z = 0x00000000,0x00000000,0x00000000,0x00000000,0x00000000,0x00000001
Other starting points were also investigated where z>1.
Example: the output values for key $0b were:
Value of x: 3e677863 ed84f02a 514987dd f5ec9fee 26cbc7bf 8794ca26
Value of y: 75bed8f8 327b78cd eb1d339e d6e9d58d 856922e5 6c3ca607
Value of z: 8c83fb04 a32bc227 9e07c3d0 6bfad1e1 ae9357aa 99a48ae5

Appendix: B

Jacobi form of curve, represented as an intersection of two quadratics[8,3] **E: $w^2 + x^2 - z^2 = 0$, $k^2w^2 + y^2 - z^2 = 0$** *, $p = x^{192}$-x^{64}-1, with* k^2 [2] and starting points given as:
w={0x7a73b10f, 0xd4201d0c, 0xf0a56204, 0xba70362f, 0x2471ac47, 0x067277d1 };
x={0x12712cc2, 0xcbe55812, 0x2bcb2aaa, 0x00a9e313, 0xc75c9c34, 0x15d2b44a };
y={0x47fc02ce, 0xa38ea373, 0x2eae6122, 0xb9d9f5e6, 0xab9dd76a, 0x300be399 };
z={0x01379630, 0x88fd6a29, 0x50f0f425, 0xa78b7b28, 0x98fd71c7, 0xa23f074d };
k^2={0x33148392, 0xa8a1abb4, 0xd16e45ba, 0xa2451dbb, 0x983a69d4, 0x286eca33 };
Example: the output points for key $0b were
Value of w: 98d7bb57 b34b19d3 399b7ed 2371a568 4274c9aa 38297506
Value of x: 45cfa54b ee52c9a0 ca3b06bb c2c9641f 6634e224 465267dc
Value of y: d2c1c135 e88469b2 f695c8c3 6362c15d 816fc025 dc1c8ba8
Value of z: a1b9de72 e60f5d59 e5b92102 e1937046 b28420a3 1db8b731
Implemented Code: (c0,c1,c2,c3)= (a0,a1,a2,a3)+(b0,b1,b2,b3)
a3b1=a3*b1 , a0b2=a0*b2 , a2b0=a2*b0 , a1b3=a1*b3 , c01= a3b1*a0b2 , c0=c01+ a2b0*a1b3 , c11=a3b1*a1b3 , c1=c11-a2b0*a0b2 , a3a2=a3*a2 , b3b2=b3*b2 , a3a2b3b2=a3a2*b3b2 , a0a1=a0*a1 , b0b1=b0*b1 , a0a1b0b1=a0a1*b0b1 , k^2a0a1b0b1= k^2*a0a1b0b1 , c2= a3a2b3b2 - k^2a0a1b0b1 , a3b1_s= (a3b1) ^2 , a2b0_s= (a2b0) ^2 , c3= a3b1_s + a2b0_s