Secure Silicon: Towards Virtual Prototyping

Laurent Sauvage*†, Sofiane Takarabt†, Youssef Souissi† and Naofumi Homma‡
*LTCI, CNRS, Télécom ParisTech, Université Paris-Saclay
75013 Paris, France
Email: name.forname@telecom-paristech.fr
†Secure-IC
15 rue Claude Chappe
35510 Cesson-Sévigné, France
Email: name.forname@secure-ic.com
‡Graduate School of Information Sciences, Tohoku University
6-6-05, Aramaki Aza Aoba, Aoba-ku
Sendai-shi 980-8579, Japan
Email: name.forname@aoki.ecei.tohoku.ac.jp

***Abstract*—Evaluating security vulnerabilities of software implementations at design step is of primary importance for applications developers, while it has received litte attention from scientific community. In this paper, we describe virtual prototyping of an implementation of scalar multiplication aiming to make it secure against simple side-channel attacks. Reproducing information leakage as close to reality as possible requires bit- and clock-cycle accuracy, we got with Mentor Graphics Modelsim tool, simulating the execution of the software implementations on PULPino, an open-source 32-bit microcontroller based on the recently released RISC-V instruction set architecture. For each clock cycle, we compute the number of bit toggles into microcontroller, an image of the power consumption, and watch the program counter to identify the assembly instruction executed, then the corresponding C function. We first start with a naive double-and-add implementation relying on cryptographic primitives of the mbed TLS library, formerly PolarSSL before acquisition by ARM. The virtual analysis pinpoints differences in the way the double function on one side and the add function on the other side manage variables and internal operations, which can be used to extract the private key. We propose some modifications of the C code, hence independent of the considered microcontroller, then we compare the impact on performances with other solutions such as Montgomery ladder, most used in practice as more efficient.**

I. Introduction

Today the human being is becoming more and more intimately connected to technologies. Technologies are surrounding us everywhere: smart-phones, Internet-of-Things (IoT) devices, payment and banking cards, laptops, smart home devices, cars, health-care tracking engines, etc. Basically, these technologies are sharing two points: The common first point is that they are all based on small circuits, known as embedded systems, perfectly and assembled and packaged for the end-user product. Those embedded systems can be defined as small computers subject to hardware and software designing requirements. The second common point is that all those technologies are permanently holding manipulating sensitive data. The manipulated data can by any private information from personal photos and videos, e-mails, contact directory to classified and governmental documents. Everyday, new critical security issues are identified. The last one (october 2016) is no doubt the spectacular *DDOS attack on Dyn DNS*. This was as a big alert all over the world. *The Guardian*, famous UK newspaper, has recently warned people that IoT might be the next target. One may argues that the problem of data protection and privacy preserving is already resolved as the solution is simply the cryptography: This is not totally true! In fact, sure modern cryptography with its theoretical principles is robust. As a matter of fact, no one attacker would try to exhaustively predict a 128-bit secret key. This means he should try all possible combinations (2^{128}), which would take hundreds of years even with sophisticated computers. The real problem is coming from the integration of cryptography into embedded systems. The system will leak sensitive information that can be exploited easily by the so-called physical attacks. Basically, those recent attacks are classified into two classes:

1) **Passive analysis** that aims at exploiting a physical property like the power, electromagnetic emanations, time calculations,the sounds, etc to retrieve the secret. The commonly known passive analysis are Side-channel analysis (SCA.)
2) **Active analysis** that aims at interacting directly with the system by perturbing it. Such perturbation impact the systems which lead to an abnormal behaviour. Such abnormal behaviour is exploited to retrieve the secret. The most commonly known analyses are Fault injection analysis (FIA)

Obviously, assessing the robustness of embedded systems against those attacks is vital. A good evaluation of the cryptographic implementation is the key to the success for certification. The security certification is not just a marketing label, more importantly it is an image of the user confidence level. Generally, the evaluation is performed at the post-silicon stage just before assembling the end-user product. Real physical analysis requires a real target and acquisition equipment, namely an oscilloscope and probes to extract the so-called

978-1-5386-0689-6/17/$31.00

consumption traces. Unfortunately, when dealing with a real analysis, results are always severely impacted by measurement condition like the noise or traces misalignment. Now, how about assessing the robustness of a chip before producing its post-silicon version. The answer is the core idea behind the so-called virtual analysis which is entirely based on a simulated image of the cryptographic embedded design. Moreover, evaluating such at an early stage allows you considering the best ever attacker with free-noise models. Additionally, virtual prototyping allows you saving a lot of effort, time and money.

II. Mixed Software and Hardware Simulation

In our process, we provide the hardware description (HDL) to the simulator to synthesise a netlist which serve us to virtualize the design in one hand, and in the other hand we compiled the software sources to generate the binary code that will be converted to instruction and data memory files, and then, will be loaded to the hardware memory. The testbench is used to have this connection between the software and the hardware, particularly, to load data and instructions into memory and start the execution of the program. At the end of the simulation, a VCD (Value Change Dump) is exported, and it contains the transitions of each signal of the selected region in the design. We have used a model based on Hamming distance to estimate the power consumption, and generate the trace of the Program counter (PC).

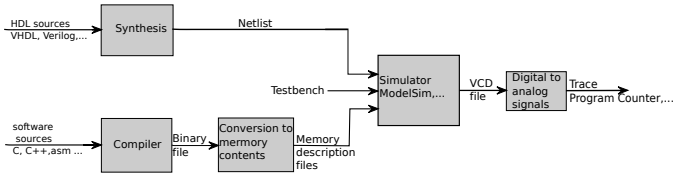

Figure 1. Workflow for the mixed software and hardware simulation.

III. Virtual Prototyping for Countermeasure Design

This section presents an example of virtual prototyping, that is to say, an example of iterating a process which consists in searching for security flaws using simulated observations, then correcting them. The source code in C language of the initial implementation is reproduced in listing 1. It performs scalar multiplication between a point `P` in the group `grp` of an elliptic curve and a private key `k`, according to the left-to-right binary algorithm: Point doubling ($2P$) is systematically performed, then the addition to itself ($P+P$) if the current key bit (k_i) is set. For these two functions, we used `ecp_double_jac` (line 13) and `ecp_add_mixed` (line 14) primitives of mbed TLS [1] library. As the left-to-right binary algorithm is particularly sensitive to Simple side-channel attacks (SSCAs), it is the perfect candidate to illustrate in the simplest way possible our example of virtual prototyping.

Figure 2 represents the simulated observation obtained when the implementation of listing 1 is executed by PULPino [2], an open-source microcontroller based on a small 32-bit RISC-V core, and designed for Internet of things (IoT) applications.

```
1  static int l2r_jac(
2     mbedtls_ecp_group *grp,
3     mbedtls_ecp_point *R,
4     const mbedtls_mpi *k,
5     const mbedtls_ecp_point *P,
6     size_t l)
7  {
8     // Initialization
9     mbedtls_ecp_point_init(R);
10    mbedtls_ecp_set_zero(R);
11    // Left-to-right binary algorithm
12    for (int i=l-1; i>=0;i--) {
13       ecp_double_jac(grp, R, R);
14       if (mbedtls_mpi_get_bit( k, i )==1)
             ecp_add_mixed(grp, R, R, P);
15    }
16    return OK;
17 }
```

Listing 1. Left-to-right binary algorithm using mbed TLS primitives.

The clock cycle number is plotted on the abscissa. In total, 376 771 cycles have been simulated. The first 28 027 cycles correspond to the boot of the microcontroller followed by the execution of the main program until the initialization and the zeroing of the result point `R` (line 10 of listing 1). The next cycles correspond to the execution of the left-to-right binary algorithm for the first four most significant bits of the key (`0b1101`). Simulation results contain the successive values taken by each register of the processor, in particular those of the register PC. Therefore, we know exactly for each clock cycle which address of the assembly code is running, and can deduce the corresponding function of the C source code. Moreover, by tracking the evolution of PC, we can determinate the calling function. We used this technique to identify the instants during which `ecp_double_jac` (line 13 of listing 1)) and `ecp_add_mixed` (line 14 of listing 1) primitives are executed. In fig. 2, these instants are indicated in green for the first primitive, in purple for the second one. Their succession is consistent with the value of the private key. The first execution of the primitives `ecp_double_jac` then `ecp_add_mixed`, from cycle 28 027 to cycle 52 192, is much shorter than the following executions. Indeed, as the result point `R` is set to zero (line 10 of listing 1), `ecp_double_jac`[1]makes only a copy of `R` into `R`, and `ecp_add_mixed` a copy of `P` into `P` (see lines 1006 to 1013 of [3]).

The ordinate is the number of RISC-V core's nets toggling, that is to say whose state has changed. It is an image of the global activity of the microcontroller, as would be in the real world a measure of its power consumption or of its electro-magnetic radiation with a large probe. We distinguish very clearly at the end of the execution of each primitive some quiet instants, whose duration is doubled for `ecp_add_mixed` (in purple). This difference makes it possible to distinguish the two primitives, and thus ultimately to reconstruct the complete sequence, which reveals the private key with a single

[1]More exactly, primitives `mbedtls_mpi_mul_mpi`, `mbedtls_mpi_add_mpi` and `mbedtls_mpi_sub_mpi` (see listing 3 in appendix.

observation. A zoom on these quiet instants is given at the bottom of figs. 3 and 4. At the top of these figures, the evolution of the PC's value shows four executions of the primitive `mbedtls_mpi_free` in the first case, seven in the second. They come from sequences of memory cleaning, line 973 of listing 3 (in appendix) for `ecp_double_jac` primitive, and lines 1065 and 1066 of listing 2 (or [3]) for `ecp_add_mixed`.

```
1063 cleanup:
1064
1065     mbedtls_mpi_free( &T1 ); mbedtls_mpi_free( &
             T2 ); mbedtls_mpi_free( &T3 );
             mbedtls_mpi_free( &T4 );
1066     mbedtls_mpi_free( &X ); mbedtls_mpi_free( &Y
         ); mbedtls_mpi_free( &Z );
```

Listing 2. Cleanup of `ecp_add_mixed` primitive [3].

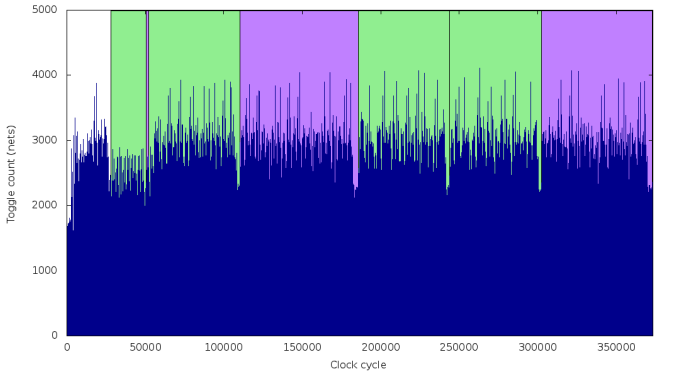

Figure 2. Simulated observation of a RISC-V core running the implementation of listing 1, for the first four most significant key bits (`0b1101`).

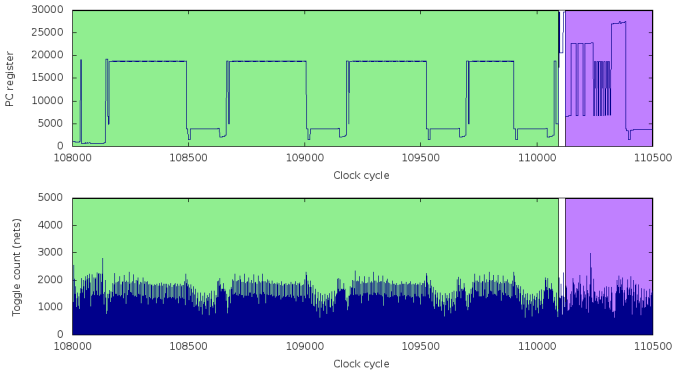

Figure 3. Program counter (*top*) and toogle count (*bottom*) at the end of `ecp_double_jac` mbed TLS primitive.

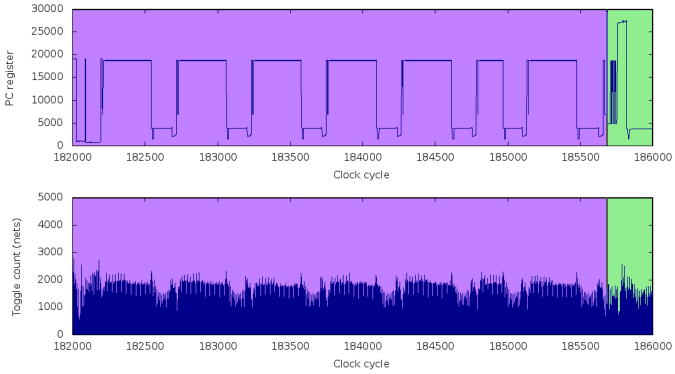

Figure 4. Program counter (*top*) and toogle count (*bottom*) at the end of `ecp_add_mixed` mbed TLS primitive.

To make this security vulnerability unexploitable, the number of `mbedtls_mpi_free` in `ecp_double_jac` primitive has to be balanced, but also their duration, which depends on the size of the variable to be unallocated. Primitive `ecp_add_mixed` (fig. 4) uses seven variables: `T1`, `T2`, `T3`, `T4`, `X` and `Z` of a size of 512 bits, and `Y` of a size of 256 bits. Primitive `ecp_double_jac` uses four variables: `S`, `T` and `U` of a size of 512 bits, and `M` of a size of 256 bits. In the end, three dummy 512-bit variables have to be declared at the beginning of `ecp_double_jac` primitive, along with three `mbedtls_mpi_free` to unallocate them. The impact of this modification on the initial implementation is 6690 cycles, so an overhead of 6.99 %, as indicated in the second line of table I.

In previous figures, simulated observations were represented as a sequence of pulses, one per clock cycle, each having an amplitude equal to the number of nets having toggled at the beginning of the cycle. To get an observation closer to reality, each individual pulse must be convolved with the temporal response of the transmission channel. This channel extends from the internal gates of the integrated circuit to the analog-to-digital converter of the oscilloscope, including measurement probes. A common technique for locating patterns potentially revealing confidential information is to extract the maximum value of the observation in each clock cycle, then to display a trace composed of the sequence of the maximums connected to each other. Applying it to the observation of fig. 2, as well as a zoom on the end, we get fig. 5.

In the zones identified in red, the bottom of the simulated observation takes the form of a solid rectangle, because of a number of commutations oscillating to low values, around 250 nets. There are eight such areas in the part corresponding to `ecp_double_jac` primitive (in green), and eleven in that of `ecp_add_mixed` (in purple). As before, the knowledge of the PC value makes it possible to identify the corresponding primitive. It is `mbedtls_mpi_mul_mpi`, and there are indeed eleven systematic calls to this primitive from `ecp_add_mixed`, at lines 1024, 1025, 1026, 1027, 1046, 1047, 1048, 1049, 1051, 1055 and 1056 of [3], and six systematic calls from `ecp_double_jac`, at lines 945, 947, 951, 955, 961 and 965 of listing 3 (in appendix). The two remaining calls are these at lines numlist921;924. Indeed, since we use the elliptic curve `secp256r1` ($y^2 = x^3 + 7$) whose parameter a is null, the condition of line 918 is valid. Note that for other curves, previous analyses remain valid. More precisely, the particular pattern which makes the identification of `mbedtls_mpi_mul_mpi` possible is due to eight successive executions of `mpi_mul_hlp` primitive,

Table I
AVERAGE DURATION OF DOUBLE-AND-ADD IMPLEMENTATIONS.

Version	#Clock cycles/key bit	Overhead
Initial (listing 1)	95 669	-
Idem w/ #`mbedtls_mpi_free` balanced	102 359	6.99 %
Idem w/ #`mpi_mul_mpi` balanced	119 377	24.78 %
double-and-add-always	133 447	39.48 %
`ecp_mul_mxz` (Montgomery ladder)	80 702	−15.64 %

generated by the loop of lines 1190 and 1191 of [4], to calculate a 256-bit multiplication by piece of 32-bit width. This pattern is identified in red in fig. 6 which represents a complete execution of `mbedtls_mpi_mul_mpi` primitive.

As before, it is possible to add three dummy calls to `mbedtls_mpi_mul_mpi` in `ecp_double_jac` primitive to in order to balance their number with those of `ecp_add_mixed`, thus removing this second security vulnerability. Multiplications should involve 256-bit variables, possibly already existing. The impact of this modification is 17 018 cycles. Added to the balancing in `mbedtls_mpi_free`, the total overhead on the initial implementation reaches 24.78 %, as reported in the third line of table I.

The classic countermeasure against the analyses we have just carried out is to always perform the addition operation (*double-and-add-always*). Its performances are reported in the penultimate line of table I. They are 15 points under the performances of the implementation with both balancing. But in reality, to be complete, it would also be necessary to balance the other functions like `mbedtls_mpi_add_mpi`. In the end, their performance should be similar. However, in practice, it is rather the Montgomery multiplier that is used, much more efficient (see last line of table I). But we recall that the aim of this article is not to design a strong counter-measure, but to illustrate the principles of virtual prototyping.

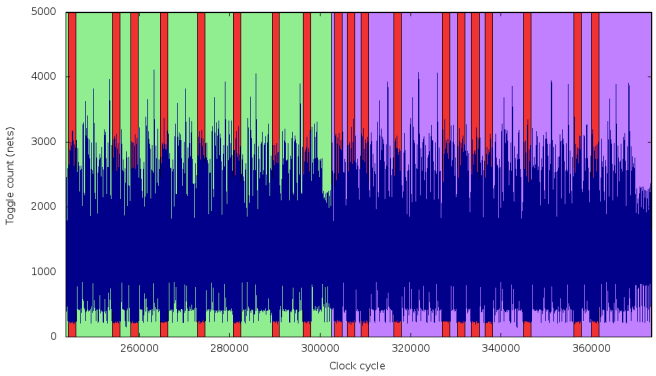

Figure 5. Simulated observation during the last execution of `ecp_double_jac` and `ecp_add_mixed` mbed TLS primitives.

IV. CONCLUSION & PERSPECTIVES

During this work we were able to conceive a simulation method of a soft program running on a SoC, and to propose a way of a cryptographic system evaluation. Our next purpose would at first be to improve the performance of the simulations and the trace acquisition and secondly, to improve the consumption model to be more realistic. The advanced evaluation as leaks detection at the hardware level needs more means for data management issued from the simulation which has to target high levels of analysis as post synthesis, place & route and post layout, which require more important material resources. Soon, we are going to consider a generic solution for any SoC, and to analyse other standard implementation and propose a mechanism more automated for the virtualisation.

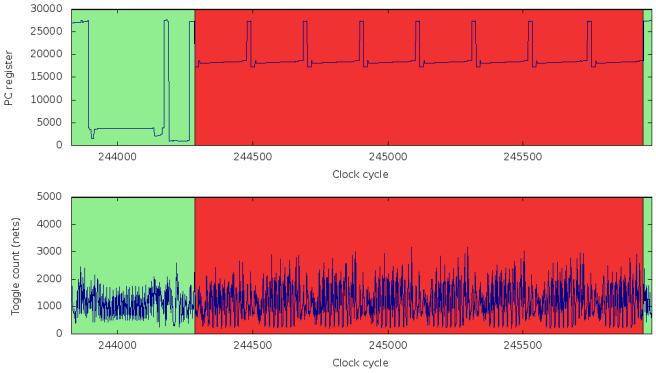

Figure 6. Program counter (*top*) and toogle count (*bottom*) during `mbedtls_mpi_mul_mpi` mbed TLS primitive.

REFERENCES

[1] ARM. mbed TLS. [Online]. Available: https://tls.mbed.org
[2] PULP – Parallel Ultra Low Power Platform. [Online]. Available: www.pulp-platform.org
[3] ARM. Elliptic curves over GF(p): generic function (ecp.c). [Online]. Available: https://tls.mbed.org/api/ecp_8c_source.html
[4] ——. Multi-precision integer library (bignum.c). [Online]. Available: https://tls.mbed.org/api/bignum_8c_source.html

APPENDIX

```
905 int ecp_double_jac( const mbedtls_ecp_group *grp, mbedtls_ecp_point *R,
906                       const mbedtls_ecp_point *P )
907 {
908     int ret;
909     mbedtls_mpi M, S, T, U;
910
911 #if defined(MBEDTLS_SELF_TEST)
912     dbl_count++;
913 #endif
914
915     mbedtls_mpi_init( &M ); mbedtls_mpi_init( &S ); mbedtls_mpi_init( &T ); mbedtls_mpi_init( &U );
916
917     /* Special case for A = -3 */
918     if( grp->A.p == NULL )
919     {
920         /* M = 3(X + Z^2)(X - Z^2) */
921         MBEDTLS_MPI_CHK( mbedtls_mpi_mul_mpi( &S,  &P->Z,  &P->Z   ) ); MOD_MUL( S );
922         MBEDTLS_MPI_CHK( mbedtls_mpi_add_mpi( &T,  &P->X,  &S      ) ); MOD_ADD( T );
923         MBEDTLS_MPI_CHK( mbedtls_mpi_sub_mpi( &U,  &P->X,  &S      ) ); MOD_SUB( U );
924         MBEDTLS_MPI_CHK( mbedtls_mpi_mul_mpi( &S,  &T,     &U      ) ); MOD_MUL( S );
925         MBEDTLS_MPI_CHK( mbedtls_mpi_mul_int( &M,  &S,     3       ) ); MOD_ADD( M );
926     }
927     else
928     {
929         /* M = 3.X^2 */
930         MBEDTLS_MPI_CHK( mbedtls_mpi_mul_mpi( &S,  &P->X,  &P->X   ) ); MOD_MUL( S );
931         MBEDTLS_MPI_CHK( mbedtls_mpi_mul_int( &M,  &S,     3       ) ); MOD_ADD( M );
932
933         /* Optimize away for "koblitz" curves with A = 0 */
934         if( mbedtls_mpi_cmp_int( &grp->A, 0 ) != 0 )
935         {
936             /* M += A.Z^4 */
937             MBEDTLS_MPI_CHK( mbedtls_mpi_mul_mpi( &S,  &P->Z,  &P->Z   ) ); MOD_MUL( S );
938             MBEDTLS_MPI_CHK( mbedtls_mpi_mul_mpi( &T,  &S,     &S      ) ); MOD_MUL( T );
939             MBEDTLS_MPI_CHK( mbedtls_mpi_mul_mpi( &S,  &T,     &grp->A ) ); MOD_MUL( S );
940             MBEDTLS_MPI_CHK( mbedtls_mpi_add_mpi( &M,  &M,     &S      ) ); MOD_ADD( M );
941         }
942     }
943
944     /* S = 4.X.Y^2 */
945     MBEDTLS_MPI_CHK( mbedtls_mpi_mul_mpi( &T,  &P->Y,  &P->Y   ) ); MOD_MUL( T );
946     MBEDTLS_MPI_CHK( mbedtls_mpi_shift_l( &T,  1               ) ); MOD_ADD( T );
947     MBEDTLS_MPI_CHK( mbedtls_mpi_mul_mpi( &S,  &P->X,  &T      ) ); MOD_MUL( S );
948     MBEDTLS_MPI_CHK( mbedtls_mpi_shift_l( &S,  1               ) ); MOD_ADD( S );
949
950     /* U = 8.Y^4 */
951     MBEDTLS_MPI_CHK( mbedtls_mpi_mul_mpi( &U,  &T,     &T      ) ); MOD_MUL( U );
952     MBEDTLS_MPI_CHK( mbedtls_mpi_shift_l( &U,  1               ) ); MOD_ADD( U );
953
954     /* T = M^2 - 2.S */
955     MBEDTLS_MPI_CHK( mbedtls_mpi_mul_mpi( &T,  &M,     &M      ) ); MOD_MUL( T );
956     MBEDTLS_MPI_CHK( mbedtls_mpi_sub_mpi( &T,  &T,     &S      ) ); MOD_SUB( T );
957     MBEDTLS_MPI_CHK( mbedtls_mpi_sub_mpi( &T,  &T,     &S      ) ); MOD_SUB( T );
958
959     /* S = M(S - T) - U */
960     MBEDTLS_MPI_CHK( mbedtls_mpi_sub_mpi( &S,  &S,     &T      ) ); MOD_SUB( S );
961     MBEDTLS_MPI_CHK( mbedtls_mpi_mul_mpi( &S,  &S,     &M      ) ); MOD_MUL( S );
962     MBEDTLS_MPI_CHK( mbedtls_mpi_sub_mpi( &S,  &S,     &U      ) ); MOD_SUB( S );
963
964     /* U = 2.Y.Z */
965     MBEDTLS_MPI_CHK( mbedtls_mpi_mul_mpi( &U,  &P->Y,  &P->Z   ) ); MOD_MUL( U );
966     MBEDTLS_MPI_CHK( mbedtls_mpi_shift_l( &U,  1               ) ); MOD_ADD( U );
967
968     MBEDTLS_MPI_CHK( mbedtls_mpi_copy( &R->X, &T ) );
969     MBEDTLS_MPI_CHK( mbedtls_mpi_copy( &R->Y, &S ) );
970     MBEDTLS_MPI_CHK( mbedtls_mpi_copy( &R->Z, &U ) );
971
972 cleanup:
973     mbedtls_mpi_free( &M ); mbedtls_mpi_free( &S ); mbedtls_mpi_free( &T ); mbedtls_mpi_free( &U );
974
975     return( ret );
976 }
```

Listing 3. `ecp_double_jac` primitive [3].