Speculative Execution Resilient Cryptography

Rui Fernandes

Supervisors: Bernardo Portela, José Bacelar Almeida, Tiago Oliveira

July 2023

Master's degree in Information Security
Faculty of Sciences of the University of Porto

Context

- **Speculative Execution:** Modern CPUs improve performance by predicting and executing instructions ahead of time;
- **Speculative Execution Attacks:** Exploit speculative execution to execute instructions that would not be executed otherwise. Examples:
  - **Spook.js [1]:** Read sensitive information such as passwords. Affects all Chromium-based browsers;
  - **Foreshadow [2]:** Extract cryptographic keys from Intel SGX enclaves;
  - **Meltdown [3]:** Read kernel memory from the user-space.

Motivation

- We can block speculative execution by inserting `LFENCE` instructions at specific points in the program;
  - The excessive use of these instructions results in a considerable performance penalty.
- **Alternative:** Speculative Load Hardening (SLH) $\Rightarrow$ harden speculative loads from memory;
- Jasmin is a framework for writing efficient and verified cryptographic software:
  - Cryptographic software is security critical;
  - In Jasmin, we can protect programs against Spectre v1 attacks at the source level by implementing SLH using a type system [4].

Objectives

- Protect `libjbn`, a Jasmin big number library, against speculative execution attacks;
- Extend `libjbn` with a generic implementation of arithmetic operations over elliptic curves.

Background

Side-channel Attacks

Side-channel attacks exploit information leaked through the implementation rather than targeting the underlying cryptographic algorithms:

- **Timing side-channel attacks:** Exploit differences in the execution time of a program:
  - **Constant-Time (CT) Programming:** Control-flow and memory accesses should be independent of secret data;
- **Cache side-channel attacks:** Exploit the fact that accessing data in the cache is faster than accessing data from memory;
- Other side-channels can also be exploited: e.g. power consumption.

Spectre v1 – Bounds Check Bypass

Exploits conditional branch misprediction;

```
if (x < array1_size) { // x is unstrusted input
    index = array1[x] * 4096;
    y = array2[index];
}
```

1: Conditional branch misprediction [5]

Speculative Load Hardening

- **Speculative Load Hardening (SLH):** Maintain a predicate indicating whether the execution is misspeculating or not. If it is, this value is then used to "poison" both values and memory addresses of load instructions.
- **Selective Speculative Load Hardening (selSLH):** Not all speculative loads need to be hardened.

Jasmin

Jasmin

Framework for writing cryptographic implementations:

- Verfication-friendly;
- Low-level;
- High-assurance and high-speed;

- The compiler is proved to be functionally correct in the Coq proof assistant;
- Automatic safety checker:
  - Memory-Safety;
  - Termination.

Jasmin: Example

```
fn sum(reg ptr u64[100] p) -> reg u64 {
    reg bool cond;
    reg u64 sum i;

    sum = 0; i = 0;

    while { cond = (i < 100); } (cond) {
        sum += p[(int) i];
        i += 1;
    }

   return sum;
}
```

2: Jasmin local function [6]

Control-Flow:

- For loops are fully unrolled:
  - The compiled assembly does not contain any branching instructions;
  - The number of iterations must be known at compile-time;
- While loops/if statements compile to branching instructions.

Functions:

- **Inline functions:** Function calls are replaced by the function body;
- **Local functions:** Compile to `call`/`ret` instructions;
- **Export functions:** Can be called from other programs – e.g. C programs.

Speculative Type System

Security Levels & Security Types

- Two security levels L and H:
  - L denotes a low security level.
  - H a high security level.

- A security type is a pair of security levels (τ_n, τ_s):
  - τ_n denotes the security level under normal (i.e. sequential) execution.
  - τ_s denotes the security level of all executions (including misspeculation).

  - (L, L) denotes public data;
  - (H, H) denotes secret data;
  - (L, H) denotes transient data.

Speculative Constant-Time (SCT) & Speculative Type System

- A program is SCT if, for every possible choice of speculation, it does not leak anything beyond what is leaked during sequential execution;
- The type system provides primitives to implement selSLH, ensuring that code is SCT.

Jasmin	Semantics	Compiled to
`ms = #init_msf();`	`ms = 0`	`lfence; ms = 0;`
`ms = #update_msf(e, ms);`	`assert(e)`	`ms = -1 if !e;`
`x = #protect(x, ms);`	`assert(ms == 0)`	`x |= ms;`

Table 1: Type system primitives

Speculative Type System – Example

```
fn sum(#msf reg u64 ms, reg ptr u64[100] p)
    -> #msf reg u64, #public reg u64 {
    #public reg bool cond;
    #public reg u64 sum i;

    sum = 0; i = 0;

    while { cond = (i < 100); } (cond) {
        ms = #update_msf(cond, ms);
        sum += p[(int) i];
        i += 1;
    }

    ms = #update_msf(!cond, ms);
    sum = #protect(sum, ms);

   return ms, sum;
}
```

3: Jasmin local function protected against Spectre-v1 [6]

Speculative Type System – Example

```
fn sum(#msf reg u64 ms, reg ptr u64[100] p)
    -> #msf reg u64, #public reg u64 {
    #public reg bool cond;
    #public reg u64 sum i;

    sum = 0; i = 0;

    while { cond = (i < 100); } (cond) {
        ms = #update_msf(cond, ms);
        sum += p[(int) i];
        i += 1;
    }

    ms = #update_msf(!cond, ms);
    sum = #protect(sum, ms);

   return ms, sum;
}
```

4: Jasmin local function protected against Spectre-v1 [6]

Speculative Type System – Example

```
fn sum(#msf reg u64 ms, reg ptr u64[100] p)
    -> #msf reg u64, #public reg u64 {
    #public reg bool cond;
    #public reg u64 sum i;

    sum = 0; i = 0;

    while { cond = (i < 100); } (cond) {
        ms = #update_msf(cond, ms);
        sum += p[(int) i];
        i += 1;
    }

    ms = #update_msf(!cond, ms);
    sum = #protect(sum, ms);

   return ms, sum;
}
```

5: Jasmin local function protected against Spectre-v1 [6]

Speculative Type System – Example

```
fn sum(#msf reg u64 ms, reg ptr u64[100] p)
    -> #msf reg u64, #public reg u64 {
    #public reg bool cond;
    #public reg u64 sum i;

    sum = 0; i = 0;

    while { cond = (i < 100); } (cond) {
        ms = #update_msf(cond, ms);
        sum += p[(int) i];
        i += 1;
    }

    ms = #update_msf(!cond, ms);
    sum = #protect(sum, ms);

   return ms, sum;
}
```

6: Jasmin local function protected against Spectre-v1 [6]

Implementation

Libjbn

Jasmin big number library;

- Defines a set of basic arithmetic operations for big integers and finite field arithmetic;
- Each number is represented an array of 64-bit unsigned integers;
- Functions are generic on the number of limbs of the numbers;
- The programmer must provide a set of parameters: e.g. number of limbs, size of the finite field.

Source Code Modifications

- If needed, free one register for the misspeculation flag;
- If needed, change function signature to return the misspeculation flag;
- Add security type annotations:
  - Arguments of export functions are transient;
  - Memory addresses are public;
  - Big numbers are secret.
- If needed, protect loads from memory;
- Declassify public values loaded from memory.

Source Code Modifications – Example

```
export fn bn_set0(reg u64 rp) {
    inline int i;

    for i = 0 to NLIMBS {
        [rp + 8*i] = 0;
    }
}
```

```
export fn bn_set0(#transient reg u64 rp) {
    _ = #init_msf();

    inline int i;

    for i = 0 to NLIMBS {
        [rp + 8*i] = 0;
    }
}
```

Source Code Modifications – Example

```
// ... implementation omitted for brevity
for i = 0 to NLIMBS {
    t = scalar[(int) i];

    k = 64;
    while { cond = (k > 0); } (cond) {
        ms = #update_msf(cond, ms);
        sk = k; st = t;
        // ... implementation omitted for brevity
        t = st;
        // ... implementation omitted for brevity
        k = sk;
        k = #protect(k, ms);
        k -= 1;
    }
    ms = #update_msf(!cond, ms);
}
```

Performance Evaluation – Number of Cycles (Integer Arithmetic)

Function	CT (Clock Cycles)	SCT (Clock Cycles)	Overhead (%)
bn_addn	60	100	66.67
bn_copy	60	100	66.67
bn_eq	60	100	66.67
bn_muln	100	140	40
bn_set0	60	100	66.67
bn_sqrn	100	140	40
bn_subn	60	100	66.67
bn_test0	60	100	66.67

Table 2: Performance comparison of integer arithmetic functions for 4 limbs

Performance Evaluation – Number of Cycles (Field Arithmetic)

Function	CT (Clock Cycles)	SCT (Clock Cycles)	Overhead (%)
fp_add	80	160	100
fp_expm_noct	35760	35560	-0.56
fp_fromM	120	180	50
fp_inv	63180	63520	0.54
fp_mul	220	280	27.27
fp_sqr	220	280	27.27
fp_sub	80	140	75
fp_toM	140	200	42.86

Table 3: Performance comparison of finite field arithmetic functions for 4 limbs

Performance Evaluation – Number of Cycles (Field Arithmetic)

Function	CT (Clock Cycles)	SCT (Clock Cycles)	Overhead (%)
`fp_add`	80	160	100
`fp_expm_noct`	35760	35560	-0.56
`fp_fromM`	120	180	50
`fp_inv`	63180	63520	0.54
`fp_mul`	220	280	27.27
`fp_sqr`	220	280	27.27
`fp_sub`	80	140	75
`fp_toM`	140	200	42.86

Table 3: Performance comparison of finite field arithmetic functions for 4 limbs

Elliptic Curve Cryptography

Elliptic Curve Cryptography: Algebraic Addition

Incomplete addition formulas:

- If $P_1 = \mathcal{O}$, then $P_1 \oplus P_2 = P_2$.
- If $P_2 = \mathcal{O}, P_1 \oplus P_2 = P_1$.
- If $P_2 = -P_1$, i.e. if $P_2 = (x_1, -y_1)$, then $P_1 \oplus P_2 = \mathcal{O}$.
- If $P_1 = P_2$, then $P_1 \oplus P_2 = 2 \cdot P_1 = (x_3, y_3)$, where:

$$x_3 = \lambda^2 - 2x_1, \quad y_3 = \lambda(x_1 - x_3) - y_1, \qquad \text{where } \lambda = \frac{3x_1^2 + a}{2y_1}$$

- Otherwise, $P_1 \oplus P_2 = P_3 = (x_3, y_3)$, where:

$$x_3 = \lambda^2 - x_1 - x_2, \quad y_3 = \lambda(x_1 - x_3) - y_1, \qquad \text{where } \lambda = \frac{y_1 - y_2}{x_1 - x_2}$$

Scalar Multiplication

Given a point $P \in E(\mathbb{F}_p)$ and a scalar $k \in \mathbb{Z}$, we compute the point $Q = k \cdot P$ by repeatedly adding P with itself:

$$Q = k \cdot P = \underbrace{P \oplus P \oplus \cdots \oplus P}_{k \text{ terms in the sum}}$$

Scalar Multiplication – Implementation

Require: t-bit scalar $k = (k_{t-1}, \ldots, k_1, k_0)_2$, Point $P \in E(\mathbb{F}_p)$
Ensure: $Q = k \cdot P$

```
Q ← 𝒪
for i from 0 to t − 1 do
    if k_i = 1 then
        Q ← Q ⊕ P
    end if
    P ← 2P
end for
return Q
```

Algorithm 1: Right-to-left binary method for point multiplication

Scalar Multiplication – Implementation

- Vulnerable to side-channel attacks – leaks the binary representation of the scalar k.

- In ECDSA, the public key Q is computed as $Q = k \cdot G$, where k is the private key and G is the base point of the curve.

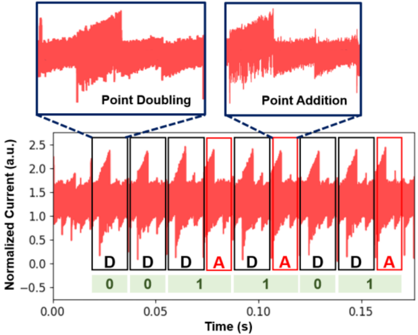

Figure 1: Power trace of the double-and-add algorithm on a RISC-V CPU [7]

Scalar Multiplication – Montgomery Ladder

Require: t-bit scalar $k = (k_{t-1}, \ldots, k_1, k_0)_2$, Point $P \in E(\mathbb{F}_p)$
Ensure: $Q = k \cdot P$

$R_0 \leftarrow \mathcal{O}$
$R_1 \leftarrow P$
for i from 0 to $t-1$ **do**
    if $\mathrm{k}_i = 0$ **then**
        $R_1 \leftarrow R_0 \oplus R_1$
        $R_0 \leftarrow 2 \cdot R_0$
    else
        $R_0 \leftarrow R_0 \oplus R_1$
        $R_1 \leftarrow 2 \cdot R_1$
    end if
end for
return R_0

Algorithm 2: Montgomery Ladder for point multiplication

Arithmetic Operations Over Elliptic Curves

- Incomplete addition formulas are vulnerable to side-channel attacks and require more effort to protect against Spectre v1 attacks;
- **Complete Addition Formulas**: Compute the sum of *any* two points of $E(\mathbb{F}_p)$:
  - Slower than incomplete addition formulas;
  - Protection against side-channel attacks.

Performance Evaluation – Number of Cycles (Elliptic Curve Arithmetic)

Function	CT (Clock Cycles)	SCT (Clock Cycles)	Overhead (%)
`ecc_add`	2480	2560	3.23
`ecc_branchless_scalar_mul`	1732380	1741960	0.55
`ecc_double`	2200	2280	3.64
`ecc_mixed_add`	2180	2220	1.83
`ecc_normalize`	63980	63380	-0.94
`ecc_scalar_mul`	1178220	1187860	0.82

Table 4: Performance comparison of elliptic curve arithmetic functions for 4 limbs

Performance Evaluation – Number of Cycles (Elliptic Curve Arithmetic)

Function	CT (Clock Cycles)	SCT (Clock Cycles)	Overhead (%)
`ecc_add`	2480	2560	3.23
`ecc_branchless_scalar_mul`	1732380	1741960	0.55
`ecc_double`	2200	2280	3.64
`ecc_mixed_add`	2180	2220	1.83
`ecc_normalize`	63980	63380	-0.94
`ecc_scalar_mul`	1178220	1187860	0.82

Table 4: Performance comparison of elliptic curve arithmetic functions for 4 limbs

Performance Evaluation – Number of Cycles

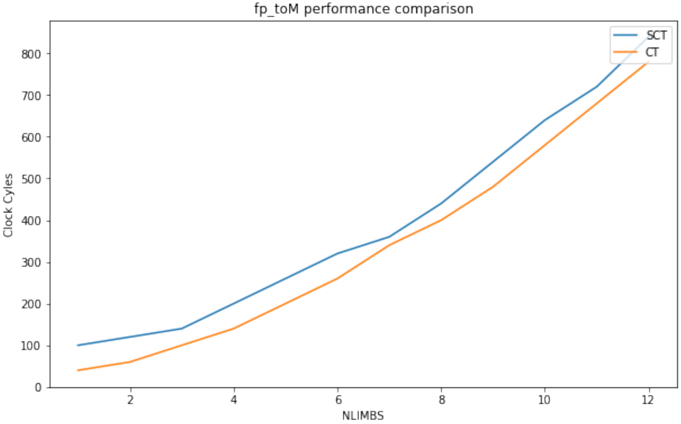

Figure 2: Cycle Count in terms of the number of limbs for the `fp_toM` function

Conclusion & Future Work

Conclusion

It is possible to protect cryptographic implementations against Spectre v1 with minimal overhead.

Limitations

- The Jasmin compiler is not proved to preserve the protections enforced by the type system $\Rightarrow$ we do not have the guarantee that the compiled assembly is SCT.
- Other side-channels can still be exploited (e.g. Differential Power Analysis attacks).

Future Work

- Optimized implementations leveraging AVX/AVX2 instructions.
- Formal proofs of correctness in EasyCrypt.

Questions?

References i

[1] A. Agarwal, S. O'Connell, J. Kim, S. Yehezkel, D. Genkin, E. Ronen, and Y. Yarom, "Spook.js: Attacking Chrome Strict Site Isolation via Speculative Execution," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 699–715.

[2] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

References ii

[4] B. A. Shivakumar, G. Barthe, B. Grégoire, V. Laporte, T. Oliveira, S. Priya, P. Schwabe, and L. Tabary-Maujean, "Typing high-speed cryptography against spectre v1," Cryptology ePrint Archive, Paper 2022/1270, 2022, https://eprint.iacr.org/2022/1270. [Online]. Available: https://eprint.iacr.org/2022/1270

[5] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *2019 IEEE Symposium on Security and Privacy (S&P)*, 2019, pp. 1–19.

[6] "`jasmin-lang/jasmin`: Language for high-assurance and high-speed cryptography," https://github.com/jasmin-lang/jasmin, (Accessed on 14/02/2023).

[7] U. Banerjee, "Efficient Algorithms, Protocols and Hardware Architectures for Next-Generation Cryptography in Embedded Systems," Thesis, Massachusetts Institute of Technology, Jun. 2021. [Online]. Available: https://dspace.mit.edu/handle/1721.1/139330