White-Box Elliptic Curve Diffie-Hellman

Arnaud Casteigts

October 27, 2011

Abstract

This report describes the prototype implemented during August and September 2011, as part of a SME4SME Applied Research Project between Irdeto Canada and the University of Ottawa. The prototype follows a cryptographic scheme designed by Mizanur Rahman between April and July 2011 as part of the same project (see Mizan's report [1] for design details). The level of integration with respect to Irdeto's library, *Clockware Security* (*CS*), lies somewhere between "out of the box" and fully integrated. Precisely, the *runtime* parts at stub, smooth, and robust levels (without transcode pragmas) are merged with the library, while the *compile time* generation of glue code and data is simulated through hardcoded functions which fake the behavior of `codegen` and `datagen` for the prototype needs. Testing can be done through compiling and running a given test program in stub, smooth, or robust mode.

1 Introduction

This report being intended for internal use, the reader is referred to Irdeto's intranet or the web for basics on key agreement protocols and white box cryptography. Below is a minimal introduction lifted from the SME4SME project proposal:

> Elliptic Curve Diffie-Hellman (ECDH) is a key agreement protocol that allows two parties, each having an elliptic curve public-private key pair, to establish a shared secret over an insecure channel. This shared secret may be directly used as a key, or to derive another key which can then be used to encrypt subsequent communications using a symmetric key cipher. It is a variant of the Diffie-Hellman protocol using elliptic curve cryptography. ECDH is an important protocol, used for things like DTCP-IP (Digital Transmission Content Protection over Internet Protocol). [...] While ECDH is a great technique to establish protected communication between two trusted parties, it is from the era of the 'Black-Box' attack context, where attacks can only happen as a 'Man-in-the-Middle'. When we consider the case where there is distrust with a party or the 'White-Box' attack context, there are many more concerns and issues with the execution of the software for the 'Man-at-the-End'.

Among these issues is the fact that the adversary can watch the host memory while a program executes. If this program makes use of cryptographic primitives, such as key agreement primitives, the privates keys become exposed. Besides the generic obfuscation transforms already offered by the CS library, this project aimed to find a way to perform the three basic operations of ECDH key agreement – *Key Pair Generation*, *Shared Secret Computation*, and *Key Derivation* – without exposing the private keys. This was achieved in the design of [1], through breaking up critical computations into a sequence of indirect operations involving compile time random numbers.

2 Testing the prototype

You should first apply the patch from within your `cs/` directory (`cd cs; patch -p1 < ecdh-arnaud.patch`), then recompile the library using `make` at any level higher or equal to `modules/crypto/`. To run the prototype, you can go in `modules/crypto/asymmetric/ecc/test/e2e/ecdh`, and then:

For stub mode:	`$ make ka-stub.exe` *(compilation)*	`$ ./ka-stub.exe` *(execution)*
For smooth mode:	`$ make ka-smooth.exe`	`$ ./ka-smooth.exe`
For robust mode:	`$ make ka-robust.exe`	`$ ./ka-robust.exe`

The smooth and robust executions should output something like:

```
Key Pair Generation on A's side
A_pub->Qx: 8e9ab49e 06433f45 94c89aca 5bbfd4e7 35d956af 3f821f64 06a27b1f 0da81eaf
A_pub->Qy: 71ddd3a7 40657bd3 40035f79 0ef542b8 7df534ee 87f9f239 a014a9d8 9fa6fe5c
A_pri->d:  20e04c02 cd2efb24 a36e8e98 7e181c8d 96af0657 ec551a4d c1eb27f2 25ff49b2

Key Pair Generation on B's side
B_pub->Qx: 235af398 8cdc3310 50ba64e2 f9c3f23a 97ce34eb ac3f808d 2bcb3d6f 4db1252b
B_pub->Qy: 5b64f011 1281d92f 83da9065 3cea9533 ef0875ef 3e64499e 06a1a37b 152dc690
B_pri->d:  8c2d53ff 48de4340 4b09ac36 b3a95464 c9620b42 fcd66475 2690c66c 8e5a7119

Shared Secret Computation on A's side
A's Shared Secret: 4bb447a7 82336711 6bcb072f f07955dc 690b1f08 ae6bea66 0a1bb71d dfd0f038

Shared Secret Computation on B's side
B's Shared Secret: 4bb447a7 82336711 6bcb072f f07955dc 690b1f08 ae6bea66 0a1bb71d dfd0f038

Key Derivation Function on A's side
A's Derivated Key: 4bb447a7 82336711 6bcb072f f07955dc 690b1f08 ae6bea66 0a1bb71d dfd0f038

Key Derivation Function on B's side
B's Derivated Key: 4bb447a7 82336711 6bcb072f f07955dc 690b1f08 ae6bea66 0a1bb71d dfd0f038
```

Note that the content of `A_pri->d` in robust mode is not A's private key (which is never present in memory), but instead the value of $\hat{r}$, the so-called *transformed random number*. The struct field was so recycled to allow the sharing of $\hat{r}$ in between calls without modifying the internal representation of a private key in the CS library (`struct xc_wb_ECC_private_key`). Another possibility is to create a dedicated field in the structure `XC_KeyAgreementOptions`.

3 ECDH White-box design (in a nutshell)

The normal (*i.e.,* black-box) procedure described p30 of [2] specifies that the key pair (private key d, public key Q), where d is a large number and Q is an elliptic curve point, is to be generated by means of a random number r as follows: $d = r + 1$, and $Q = dG$, where G is a given *generator* of the considered field. Several white-box methodologies were proposed in [1], corresponding to various strengths and complexities. After discussion with James, it was decided to focus mainly on *Methodology 1* (henceforth abbreviated *M1*), which is believed to offer the best tradeoff.

The trick is to replace the use of r by another number $\hat{r}$ such that $\hat{r} = k_1\, r + k_2$, where k_1 and k_2 are two *instance-specific* (and thus compile time) random numbers k_1 and k_2 called the *magic* numbers. In fact, r is never drawn concretely; $\hat{r}$ is the drawn number (at runtime), and is manipulated *as if* it were determined by the above expression, *i.e.*, in such a way that:

1: $\hat{G} \leftarrow (k_1^{-1})G$
2: $U_1 \leftarrow \hat{r}\,\hat{G}$
3: $U_2 \leftarrow (k_1 - k_2)\hat{G}$
4: $Q \leftarrow U_1 + U_2$

leads the very same Q as in $Q = (r + 1)G$ without ever using r. Symmetrically, computation of the shared secret (the x-coordinate of the point $P = d_A Q_B = d_B Q_A$) can be reformulated through:

1: $\hat{Q}_1 \leftarrow (k_1^{-1})\ Q$
2: $\hat{Q}_2 \leftarrow \hat{r}\hat{Q}_1$
3: $\hat{Q}_3 \leftarrow (k_1 - k_2)\ \hat{Q}_1$
4: $P \leftarrow \hat{Q}_2 + \hat{Q}_3$

where Q is the *remote* public key. Algebraic proofs of these equivalences are given in [1]. As for the *Key Derivation* procedure, which simply copies the shared secret in our case, it needs no particular white-box adaptation.

3.1 Compile time versus Runtime

White-box computation of keys and shared secrets (as specified above) are not performed as a single block. Since $k1$, $k2$, and G are already known at compile time, some computations involving them can be performed by `datagen` at compile time (more security advantages). This is the case with $\hat{G}$ and U_2 (key pair generation), and k_1^{-1} and $k_1 - k_2$ (shared secret computation). These are depicted in red/gray on Figure 1. Note that, contrary to the

generator G, the remote public key Q is known only at runtime, which is why we cannot compute $\hat{Q}_1$ and $\hat{Q}_3$ at compile time (at least in the case of *ephemeral* key agreement, which is the one we consider).

1: $\hat{G} \leftarrow (k_1^{-1})G$
2: $U_1 \leftarrow \hat{r}\,\hat{G}$
3: $U_2 \leftarrow (k_1 - k_2)\hat{G}$
4: $Q \leftarrow U_1 + U_2$

(a) Key Pair Generation

1: $\hat{Q}_1 \leftarrow (k_1^{-1})\ Q$
2: $\hat{Q}_2 \leftarrow \hat{r}\hat{Q}_1$
3: $\hat{Q}_3 \leftarrow (k_1 - k_2)\ \hat{Q}_1$
4: $P \leftarrow \hat{Q}_2 + \hat{Q}_3$

(b) Shared Secret Computation

Figure 1: Computation by `datagen` at compile time (red/gray) *vs.* by CS library at runtime (black).

3.2 A slight variant

The computation of $\hat{Q}_1$ (*step 1 above*) involves at runtime the use of one magic number non-combined with the other (k_1, inverted), which may or may not be considered as a security problem (likely not serious if working at a transformed level). An computationally equivalent alternative is to pose $\hat{r} \leftarrow k_2(k_1 r + 1)$ instead of $\hat{r} = k_1\, r + k_2$, and consider instead these new versions of lines 1 and 3:

1: $\hat{G} \leftarrow (k_1^{-1}k_2^{-1})\ G$
3: $U_2 \leftarrow (k_2(k_1 - 1))\ \hat{G}$

(a) Key Pair Generation

1: $\hat{Q}_1 \leftarrow (k_1^{-1}k_2^{-1})\ Q$
3: $\hat{Q}_3 \leftarrow (k_2(k_1 - 1))\ \hat{Q}_1$

(b) Shared Secret Computation

Figure 2: A possible variant combining k_1 and k_2 in every step. (*Steps 2 and 4 are unchanged.*)

This fix was implemented in addition to the original version. Is not clear whether it adds more security or just little confusion to an attacker; the answer certainly depends on whether k_1 and k_2 are to be used in other parts of the code. In any case, the changes between both variants are confined to the level of `datagen`. Runtime computations remain unchanged, and to make it clear, the runtime code for *shared secret computation* refers to the red expressions usingthe generic names of `factorQ1` and `factorQ3`. The computations in *key pair generation* were also adapted to be consistent (the variant must indeed be the same for both operations, although it needs not be the same among hosts). The name of both variants in the prototype are `v1` and `v2`.

4 Implementation

4.1 User API

The CS Library offers generic cryptography APIs that encapsulte the concrete implementation of algorithms. The main sets of API are:

- Block-Ciphers
- Asymmetric-Ciphers
- Hash-Functions

The generic function names therein are to be called from the user program, and then be mapped at compile time (through a glue code generated by `codegen`) into real calls to the concrete algorithms. This allows the user to tune various parameters or to switch from an algorithm to another (*e.g.* `des` to `aes`) without modifying its code, by means of an external parameter file. Key Agreement protocols do not fall into any of these APIs, and the following generic function names were introduced as part of a new *Key-Agreement* family:

- `XC_Dynamic_Key_Asymmetric_Cipher_Key_Agreement_Key_Pair_Generation()`
- `XC_Dynamic_Key_Asymmetric_Cipher_Key_Agreement_Shared_Secret_Computation()`
- `XC_Dynamic_Key_Asymmetric_Cipher_Key_Agreement_Key_Derivation_Function()`

Key agreement is not strictly speaking an "Asymmetric Cipher", but it was decided to keep this radical to simplify future integration into `codegen` (for conventions related to the automatic contraction of function names in `codegen` templates). The file containing these APIs is `modules/crypto/include/xc/xc_wb.h`. It actually contains macros that expand these functions according to the mode considered (stub, smooth, or robust). In stub mode, the functions have their names appended with a `_stub()` suffixe, which makes them point to the generic stub implementations in the CS library (`modules/crypto/asymmetric/common/src/wb_ac_stub.c`). In smooth or

robust modes, the function names are appended with the id passed as their own first argument, which refers to a particular entry of the parameter file. The corresponding functions (that is, the ones being called for real) are functions generated by `codegen` into a glue code file at compile time. These functions encapsulate calls to appropriate functions in the CS library based on the parameter file again.

4.2 Architecture

The eventual architecture of key agreement features in the CS library will probably look like the diagram in Figure 3. The "additional parameters" in this diagram correspond to the red/gray elements in Figure 1, *i.e.*, those elements which can be computed (and "*diversified*") at compile time using `datagen`.

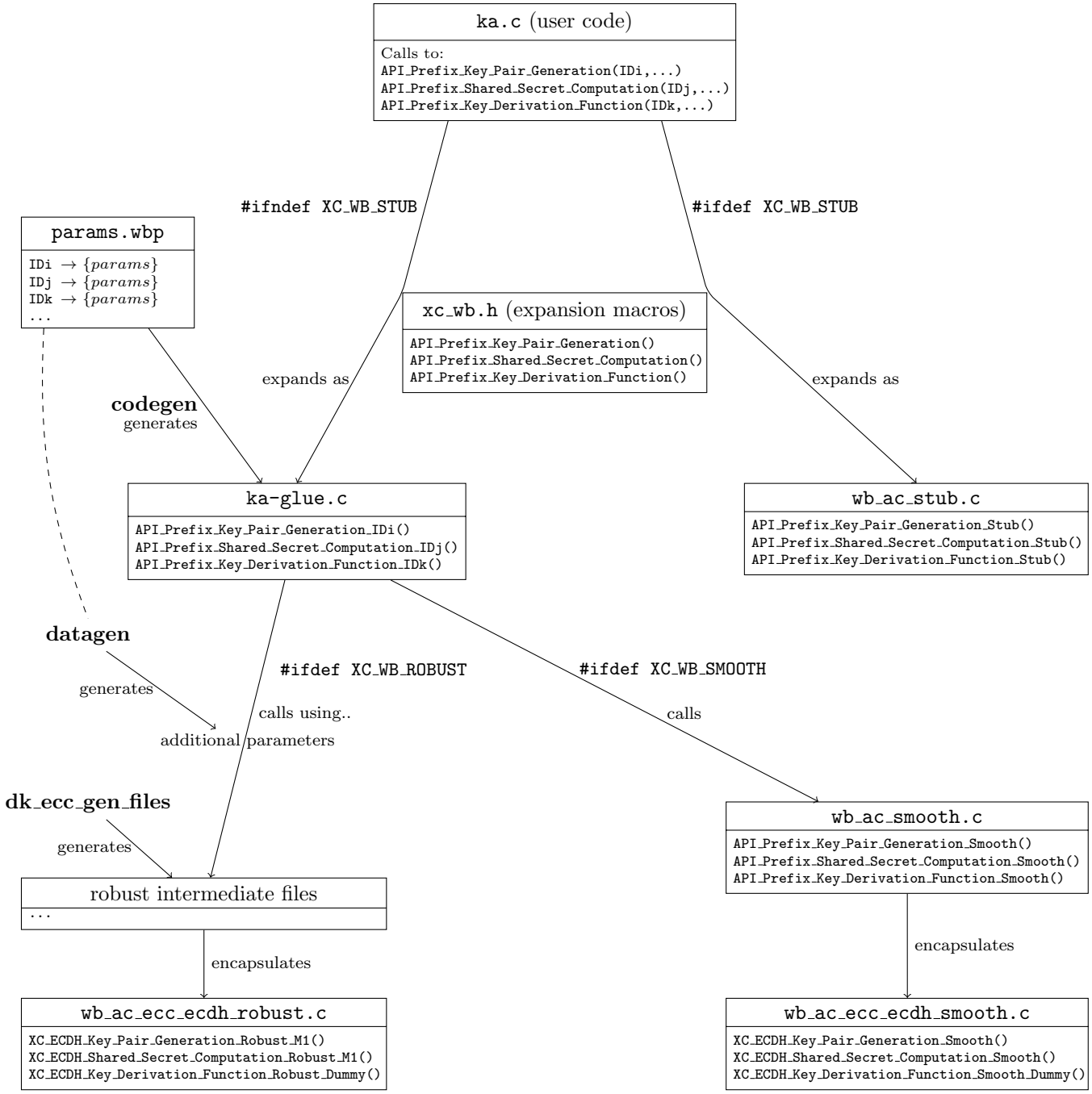

Figure 3: Eventual architecture (ignoring the transcoder). *In the function names,* `API_Prefix` *should be replaced by* `XC_Dynamic_Key_Asymmetric_Cipher_Key_Agreement`.

Given the complexity of this architecture, due mostly to the various levels of code generation involved, a number of simplifying assumption were made in order to obtain a runnable prototype in the given time frame. These assumptions are:

1. *No transforms:* the prototype code is not to be transcoded; it does not use any pragma like `_xc_transform`, `_xc_transformtype`, or `_xc_transformcast`. In a sense, it can be seen as a `noxcode`-level robust implementation on both the library and user sides.
2. *Hard glue code:* rather than modifying codegen (that is, creating new perl templates for the three key agreement functions in `xtools/whitebox/wbcodegen/WBTemplates/`), I settled down for writting a hardcoded glue file (`fake-glue.c`) which branches the key agreement calls to their appropriate implementation, depending on which of `XC_WB_STUB`, `XC_WB_SMOOTH`, or `XC_WB_ROBUST` is defined. Hence, the test program can be compiled in either mode without modifying the code (in the spirit of the other cryptographic tests).
3. *Runtime data generation:* rather than having the robust data (red/gray expressions in Figure 1) generated at compile time by `datagen`, they are generated at runtime by means of a fake datagen implemented in C in `fake-data.c`. These functions are called from within the glue code whenever `XC_WB_ROBUST` is defined, and their outputs are passed as additional parameters to the robust versions of ECDH key agreement functions (on the library side). Both variants described in Section 3 are implemented (`v1` and `v2`).

The resulting architecture in depicted on Figure 4

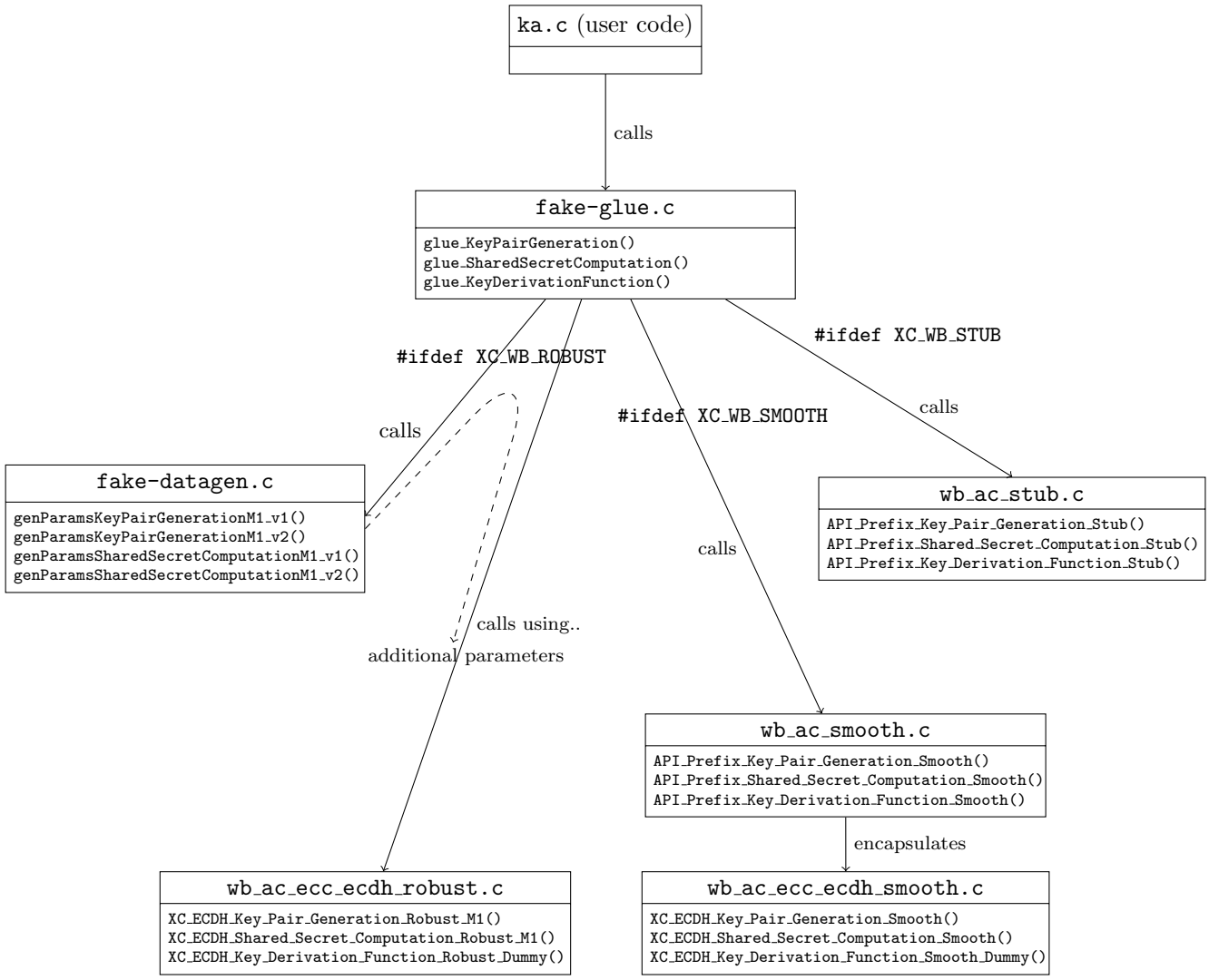

Figure 4: Architecture of the prototype

4.3 The three ECDH functions

Implementations of the three key agreement functions (key pair generation, shared secret computation, and key derivation function) constitute the heart of the prototype. They can be found in:

- `modules/crypto/asymmetric/common/src/wb_ac_stub.c` (stub implementation)
- `modules/crypto/asymmetric/ecc/src/wb_ac_ecc_ecdh_smooth.c` (smooth implementation)
- `modules/crypto/asymmetric/ecc/src/wb_ac_ecc_ecdh_robust.c` (robust implementation)

Stub implementation, which is little more than an empty shell, is intended to validate the key flow in between API calls from a user program. Smooth implementation does implement the ECDH features, but without doing the white-box computations explained in Section 3, that is, using the standard procedures [2]. The robust implementation is the one implementing white-box computations. Both smooth and robust implementations delegate elliptic curve operations to the CS elliptic curve library, which conveniently exist in both smooth and robust versions (the robust one is called `hydrogen`). Note that our robust implementation still relies on one preamble function from the smooth elliptic curve library: `XC_ECC_Get_Domain_Parameters_Smooth()`; this function having no equivalent in the `hydrogen` version.

As per the coding style, it strived to follow the examples of comparable source files in the library, *e.g.,* declaring all variables before any assignment, allocating memory at one time and testing whether each allocation was a success, providing debugging message, and using whichever naming conventions I identified in other parts of the code.

4.4 List of impacted files

Here is the list of files that were either created or modified in the framework of this prototype. All the corresponding changes are contained in the patch `ecdh-arnaud.patch`.

1. `ka.c` *(created)*: the test program, to reside anywhere in the filesystem (in my case, this was in a dedicated subfolder of CS library test directory, `modules/crypto/asymmetric/ecc/test/e2e/ecdh/`).
2. `Makefile` *(created)*: accompanying the test program, in the same directory. The recognized targets are `ka-stub.exe`, `ka-smooth.exe`, and `ka-robust.exe`.
3. `fake-glue.c` *(created)*: hard coded glue, same directory as the test program.
4. `fake-datagen.[c,h]` *(created)*: runtime data generation, same directory again. This file uses smooth elliptic curve operations, and therefore requires an additional linking to `xc_wb_smooth.xlib:eval` despite being in robust mode (taken into account in the Mafefile).
5. `modules/crypto/include/xc/xc_wb.h` *(modified)*: key agreement API and expansion macros.
6. `modules/crypto/include/xc/xc_wb_types.h` *(modified)*: API related typedefs: `XC_KeyAgreementSecret`, `XC_KeyAgreementDerivatedKey`, and `XC_KeyAgreementOptions`.
7. `modules/crypto/asymmetric/ecc/include/int_wb_ac_ecc.h` *(modified)*: other lower-level typedefs: `struct xc_wb_ECC_secret` and `struct xc_wb_ECC_derivated_key`.
8. `modules/crypto/include/xsys/xc_wb_funcs_ac.h` *(modified)*: stub and smooth declarations.
9. `modules/crypto/asymmetric/common/src/wb_ac_stub.c` *(modified)*: stub implementation.
10. `modules/crypto/asymmetric/common/src/wb_ac_smooth.c` *(modified)*: smooth encapsulation.
11. `modules/crypto/asymmetric/ecc/src/wb_ac_ecc_ecdh_smooth.c` *(created)*: smooth implementation.
12. `modules/crypto/asymmetric/ecc/src/wb_ac_ecc_ecdh_robust.c` *(created)*: robust implementation.
13. `modules/crypto/asymmetric/ecc/src/wb_ac_ecc_ecdh_util.c` *(created)*: random number conversion.
14. `modules/crypto/asymmetric/ecc/makefile.xc_wb_ecc` *(modified)*: having the latter three files included for compilation of the library.
15. `XPP/src/com/cloakware/whitebox/ecc/ECCKeyAgreementDraft.java` *(created)*: translation of the elliptic curve operations in `fake-datagen.c` into `Java` (see below).

5 What's next

The next steps come to relax the simplifying assumptions made in Section 4.2. In other words, to make the code transcodable; write `codegen` templates in `Perl` (one for each ECDH function); and write `datagen` classes in `Java` (one for *key pair generation*, and one for *shared secret computation*). To help in the latter direction, I am joining to the patch a `Java` class (`ECCKeyAgreementDraft.java`) which is a simple translation of the fake datagen in Java. It contains four functions, generating the red/gray expressions for both design variants on Figures 1 and 2. This does not resemble the structure of other `datagen` classes, but at least provides the elliptic curve computations to be used in the eventual `datagen` implementation. The class comes with a `main` function, and can be compiled and run from `XPP/src/` as follows:

- `javac com/cloakware/whitebox/ecc/ECCKeyAgreementDraft.java`
- `java com.cloakware.whitebox.ecc.ECCKeyAgreementDraft`

On a different topic, the expansion macros for *key agreement* in `modules/crypto/include/xc/xc_wb.h` are also ready to be used.

6 The learning path...

Working on the CS library seems to involve a variety of skills and knowledge whose acquisition does not follow an obvious learning path (there are various entry points, some of which cyclically depends on each other). Besides Mizan's report [1] and crucial guidance from James (many thanks!), I found that [3] offered the most appropriate overview of the CS library to a newcomer; I wish I had encountered this document earlier in the project and encourage the team to add it among reference documents in the intranet. Finally, sketching a diagram like the one of Figure 3 helped me quite a lot to understand the various levels of interaction between static and generated code, compile time and runtime, *etc.* Please feel free to reuse it (as well as any material in this report).

References

[1] Mizanur Raman. White-Box Elliptic Curve Diffie-Hellman Key Exchange Protocol Design. *Technical Report*, Aug. 2011.

[2] National Institute of Standards and Technology. Suite B Implementer's Guide to NIST SP 800-56A. *CryptoBytes, RSA Laboratories*, 4(1):610, July 2009.

[3] Cliffor Liem, Yuan Gu, Harold Johnson. A compiler-based infrastructure for software-protection. *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, p33–44, 2008.