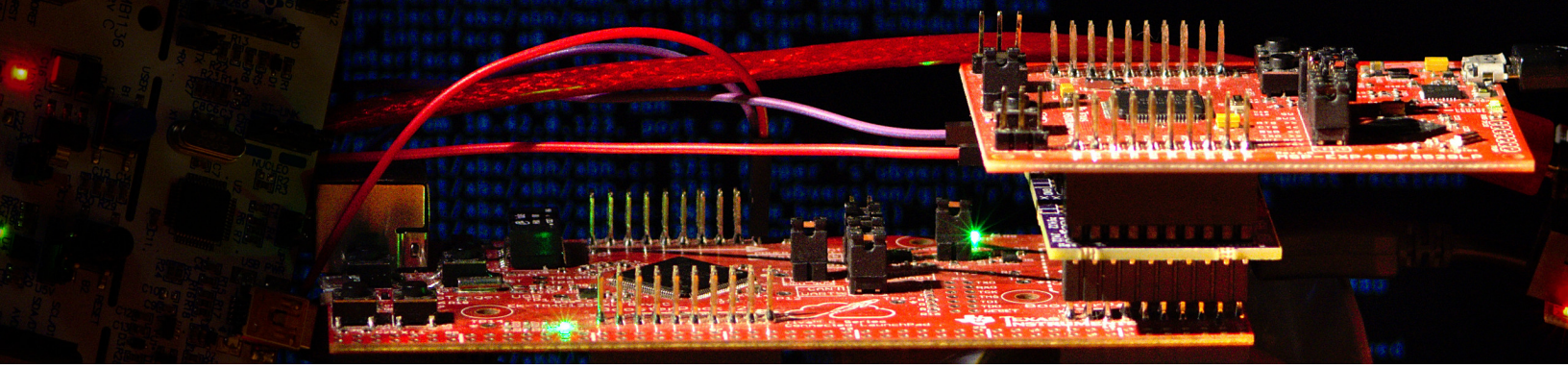

XXBX Software – XBS

XBS User Guide v1.0

Matthew R. Carter **Raghurama R. Velegala** **Jens-Peter Kaps**
jkaps@gmu.edu

April 9, 2018
George Mason University
Fairfax, Virginia

cryptography.gmu.edu

www.gmu.edu

Cryptographic Engineering Research Group
Department of Electrical and Computer Engineering George Mason University
3100 Engineering Building, 4400 University Drive, Fairfax, VA 22030-4444, USA
Voice: (703) 993-1561, Fax: (703) 993-1601

Contents

1 Introduction

The XBS was rewritten in Python 3, away from a mix of shell script and Perl that it was originally written in. The primary reason for this was because we wanted to add the ability to resume runs if a failure occurred, and we felt it was easier to work with a full object-oriented scripting language with libraries in order to add this functionality. We were also more familiar with Python then we were with Perl.

2 Build Environment

2.1 Build Process

The builds use GNU make to build the codebase. Multiple makefiles are simultaneously used to do separate parallel builds thus speeding up the build process and not blocking by stalled builds. Rebuilds do not require recompiling everything from scratch, only the code that changed or that was interrupted.

2.2 Toolchain Integration

2.2.1 TI MSP430 and MSP432

2.2.2 ST ARM

3 Benchmarking

Resuming interrupted runs was something we felt was desirable, as failure could interrupt a test run, and we did not want to have to rerun the entire test run from scratch. This was implemented but necessitated large rewrites in order to save state and prevent duplicate runs.

3.1 Setup

XXBX are mostly compatible with SUPERCOPs output and use its algopacks, which are collections of different crypto algorithms (e.g. hash, AEAD, one-time authentication, signature algorithms, etc.) that itself uses and are updated with its own development[6]. Algopacks is imported into the microcontrollers by ./import_algopack external/supercop command. Then the checksum of the crypto is acquired by using python/xbx/dircheсksum.py algobase/crypto_core/aes128encrypt/ in xbs_xbd directory. The file impl_conf.ini is updated with the checksum of the crypto acquired from the algopacks. Then config.ini is updated to select the desired platform and algorithms.

3.1.1 Algorithm Selection

The algorithm is selected in config.ini file. The file config.ini specifies whitelists and blacklists of implementations, the target operation, the target platform, paths, parameters, dependencies, etc. After this compile.py is ran using ./compile.py command on Linux platform. This creates the file data.db , used to store results, configuration, and other detailed information. If a whitelist exists in config.ini , only the implementations in this list will be built (and later run), otherwise all implementations available in the specified operation and primitives are built, excluding those specified in the blacklist. The compile script generates the makefiles and header files for each implementation and builds the code for the XBD for each implementation x compiler, flag. The compiler flag to use is defined per device, for example the mspexp430f5529lp board uses xbs_xbd/platforms/mspexp430f5529lp_16mhz/c_compilers file to update the compiler name and flags to test one line per configuration. Once a build succeeds, the execute.py script is run. The configuration used from the last build is loaded from data.db , if config.ini has not changed between the last build and calling execute.py. Behavior is undefined if config.ini or blacklists or code are modified between calls of compile.py and execute.py. For each successfully compiled implementation, the binary is loaded into the XBD and tests are run The test case configuration for this are ! Message (128 bytes) ! Associated Data (Associated Data+Message must be less than 2048 bytes) ! Ciphertext, which is the size of Message+Associated Data plus additional bytes for authentication. Additional bytes are limited to 128 bytes. ! Secret Bytes (64 bytes) ! Public Bytes (64 bytes) ! Key Bytes (64 bytes) This test case limits were configured only for encryption.

3.1.2 Target Selection

3.1.3 Compiler and Linker Options

Crosstool-NG: crosstool-NG is a set of scripts that can build a toolchain from scratch. GCC (GNU Compiler Collection) compiler was used because it supported the ARM chip on the EKTM4C1294XL. We decided to use crosstool-ng in order to build GCC, along with newlib, which provides an implementation of the standard C library.

TivaWare: The TivaWare software for C Series is an extensive suite of software tools designed to simplify and speed development of Tiva C Series-based MCU applications. It was used for the EK-TM4C1294XL board.

We support Link time Optimization (LTO), which looks at the entire generated binary when optimizing during the linking phase as opposed to a single file, This allows for more aggressive inlining and code-size reduction. However the aggressive optimizations may often break functionality and make debugging extremely difficult due to the aggressive inlining- stepping through the generated assembly is required.

3.2 Verification

As with the original XBX, primitive implementations are tested for correctness. We reused the SUPERCOP primitive implementation verification code. This code has options for large and small tests. As we are operating in a resource constrained environment, we only use the small tests, that handle lengths up to 128 bytes.

For hash functions, correct results are verified by mixing in the hash function output for various sizes into an instance of Salsa20 [**?**], instead of chaining the outputs as with original XBD. The output of this is later compared with a known value by the XBS. Deterministic results are also checked for, as well as if the implementation can handle the output being placed in the same buffer as the input and buffers remain within bounds.

For AEAD functions, the ciphertext is fed into the Salsa20 function, and is decrypted with the plaintext and associated data also fed into the Salsa20 function for checking by the XBS. In addition, the decrypted message and associated data is compared to the original message and associated data. In addition to the correctness verifications, the implementation is checked to see if the output is deterministic, if buffers remain within bounds, overlap of input and output buffers is handled if specified by the implementation, and if forgeries are detected.

The XBD to XBH (and vice-versa) communication code for sending and receiving large data over multiple I^2C packets was refactored to reuse the same code over multiple commands.

The XXBX reuses the SUPERCOP primitive implementation verification code to test primitive implementations for accuracy. Small tests, that handle lengths up to 128 bytes were used to test because of resource limitations. For hash functions, correct results are verified by mixing in the hash function output for various sizes into an instance of Salsa20 [7], instead of chaining the outputs as with original XBD. The output of this is later compared with a known value by the XBS. Deterministic results are also checked for, as well as if the implementation can handle the output being placed in the same buffer as the input and buffers remain within bounds. For AEAD functions, the ciphertext is fed into the Salsa20 function, and is decrypted with the plaintext and associated data also fed into the Salsa20 function for checking by the XBS. In addition, the decrypted message and associated data is compared to the original message and associated data.

In addition to the correctness verifications, the implementation is checked to see if the output is deterministic, if buffers remain within bounds, overlap of input and output buffers is handled if specified by the implementation, and if forgeries are detected. This was also verified by running the a select few of algorithms on a microcontroller using Code Composer. These algorithms were implemented, compiled and debugged in the Code Composer Studio IDE configured to work on MSP 430FF5529.

3.3 Results

Results can be examined by opening the generated SQLite database (`data.db`) in a SQLite browser application, such as DB Browser for SQLite [**?**]. For analysis, the SQLAlchemy objects can be manipulated directly in an IPython [**?**] notebook. IPython is an interactive python environment, with an interface similar to Maple or SageMath with "notebooks." The schema is listed in appendix **??**.

3.4 Persisting state

In order to persist the state of the runs and the actual results, we decided to store this information into a SQLite [**?**] database. We used SQLAlchemy [**?**] to simplify the task of persisting the state of the XBS. SQLAlchemy is an object-relational mapper (ORM) that maps Python objects to SQL tables and rows, without having to directly write SQL. We used this library as it is significantly simpler than using SQL directly.

4 Results Database

XXBX results can be analyzed by opening the generated SQLite database (data.db) in a SQLite browser application, such as DB Browser for SQLite [8]. For analysis, the SQLAlchemy objects can be manipulated directly in an IPython [11] notebook. IPython is an interactive python environment, with an interface similar to Maple or SageMath with notebooks. Information was saved into a SQLite [12] database. We used SQLAlchemy [13] to simplify the task of persisting the state of the XBS. SQLAlchemy is an object-relational mapper (ORM) that maps Python objects to SQL tables and rows, without having to directly write SQL. We used this library as it is significantly simpler than using SQL directly. Using the generated SQLite database (data.db) we can see the table values busing SQLbrowser called DB Browser. DB Browser shows the tables, and the column values. An example is provided below.

A Database Table Description