CTL: A Platform-Independent Crypto Tools Library Based on Dataflow Programming Paradigm (Extended Edition)*

Junaid Jameel Ahmad[1], Shujun Li[1,2], Ahmad-Reza Sadeghi[3,4], and Thomas Schneider[3]

[1] University of Konstanz, Germany
[2] University of Surrey, UK
[3] TU Darmstadt, Germany
[4] Fraunhofer SIT, Germany

Abstract. The diversity of computing platforms is increasing rapidly. In order to allow security applications to run on such diverse platforms, implementing and optimizing the same cryptographic primitives for multiple target platforms and heterogeneous systems can result in high costs. In this paper, we report our efforts in developing and benchmarking a platform-independent Crypto Tools Library (CTL). CTL is based on a dataflow programming framework called Reconfigurable Video Coding (RVC), which was recently standardized by ISO/IEC for building complicated reconfigurable video codecs. CTL benefits from various properties of the RVC framework including tools to 1) simulate the platform-independent designs, 2) automatically generate implementations in different target programming languages (e.g., C/C++, Java, LLVM, and Verilog/VHDL) for deployment on different platforms as software and/or hardware modules, and 3) design space exploitation such as automatic parallelization for multi- and many-core systems. We benchmarked the performance of the SHA-256 and AES implementations in CTL on single-core target platforms and demonstrated that implementations automatically generated from platform-independent RVC applications can achieve a run-time performance comparable to reference implementations manually written in C and Java. For a quad-core target platform, we benchmarked a 4-adic hash tree application based on SHA-256 that achieves a performance gain of up to 300% for hashing messages of size 8 MB.

Keywords: Crypto Tools Library (CTL), Reconfigurable Video Coding (RVC), dataflow programming, reconfigurability, platform independence, multi-core.

1 Introduction

Nowadays we are living in a fully digitized and networked world. The ubiquitous transmission of data over the open network has made security one of the most important concerns in almost all modern digital systems, being privacy another. Both security and privacy concerns call for support from applied cryptography. However, the great diversity of today's computing hardware and software platforms is creating a big challenge for applied cryptography since we need building blocks that should ideally be reused at various platforms without reprogramming. For instance, a large-scale video surveillance system (like those we have already been seeing in many big cities) involves many different kinds of hardware and software platforms: scalar sensors, video sensors, audio sensors, mobile sensors (e.g. mobile phones), sensor motor controller, storage hub, data sink, cloud storage servers, etc. [12]. Supporting so many different devices in a single system or cross the boundary

* This is the extended edition of a full-length paper accepted to 16th International Conference on Financial Cryptography and Data Security (FC 2012), whose proceedings is to be published as a volume of Lecture Notes in Computer Science (LNCS) by Springer in 2012.

of multiple systems is a very challenging task. Many cryptographic libraries have been built over the years to partly meet this challenge, but most of them are written in a particular programming language (e.g. C, C++, Java and VHDL) thus their applications are limited in nature. While it is always possible to port a library written in one language to the other, the process requires significant human involvement on reprogramming and/or re-optimization, which may not be less easier than designing a new library from scratch.

In this paper, we propose to meet the above-mentioned technical challenges by building a platform-independent[5] library based on a recently-established ISO / IEC standard called RVC (Reconfigurable Video Coding) [37, 38]. Unlike its name suggests, the RVC standard offers a general development framework for all data-driven systems including cryptosystems, which is not surprising because video codecs are among the most complicated data-driven systems we can have. The RVC framework follows the dataflow paradigm, and enjoys the following nice features at the level of programming language: *modularity*, *reusability*, *reconfiguration*, *code analyzability* and *parallelism exploitability*. Modularity and reusability help to simplify the design of complicated programs by having functionally separated and reusable computational blocks; reconfigurability makes reconfiguration of complicated programs easier by offering an interface to configure and replace computational blocks; code analyzability allows automatic analysis of both the source code and the functional behavior of each computational block so that code conversion and program optimization can be done in a more systematic manner. The automated code analysis enables to conduct a fully-/semi-automated design-space exploitation to find critical paths and/or parallel data-flows, which suggests different optimization refactorings (merging or splitting) of different computational blocks [47], and/or to achieve concurrency by mapping different computational blocks to different computing resources [22]. In contrast to the traditional sequential programming paradigm, the dataflow programming paradigm is ideally suited for such optimizations thanks to its data-driven nature as described next.

The dataflow programming paradigm, invented in the 1960s [66], allows programs to be defined as a directed graph in which the nodes correspond to computational units and edges represent the direction of the data flowing among nodes [27, 44]. The modularity, reusability and reconfigurability are achieved by making each computational unit's functional behavior independent of other computational units. In other words, the only interface between two computational units is the data exchanged. The separation of functionality and interface allows different computational units to run in parallel, thus easing parallelism exploitation. The dataflow programming paradigm is suited ideally for applications with a data-driven nature like signal processing systems, multimedia applications, and as we show in this paper also for cryptosystems.

Our Contributions: In this paper, we present the Crypto Tools Library (CTL) as the first (to the best of our knowledge) open and platform-independent cryptographic library based on a dataflow programming framework (in our case the RVC framework). In particular, the CTL achieves the following goals:

- **Fast development/prototyping:** By adapting the dataflow programming paradigm the CTL components are inherently *modular*, *reusable*, and easily *reconfigurable*. These properties do not only help to quickly develop/prototype security algorithms but also make their maintenance easier.
- **Multiple target languages:** The CTL cryptosystems are programmed only once, but can be used to automatically generate source code for multiple programming languages (C, C++, Java, LLVM, Verilog, VHDL, XLIM, and PROMELA at the time for this writing[6]).
- **Automatic code analyzability and optimization:** An automated design-space exploitation process can be performed at the algorithmic level, which can help to optimize the algorithmic structure by refactor-

[5] In the context of MPEG RVC framework, the word "platform" has a broader meaning. Basically, it denotes any computing environment that can execute/interpret code or compile code to produce executable programs, which includes both hardware and software platforms and also hybrid hardware-software systems.

[6] More code generation backends are going to be made in the future, especially OpenCL for GPUs.

ing (merging or splitting) selected computational blocks, and by exploiting multi-/many-core computing resources to run different computational blocks in parallel.
- **Hardware/Software co-design:** Heterogenous systems involving software, hardware, and various I/O devices/channels can be developed in the RVC framework [67].
- **Adequate run-time performance:** Although CTL cryptosystems are highly abstract programs, the run-time performance of automatically synthesized implementations is still adequate compared to non-RVC reference implementations.

In this paper, along with the development of the CTL itself, we report some performance benchmarks of CTL that confirm that the highly abstract nature of the RVC code does not compromise the run-time performance. In addition, we also briefly discuss how different key attributes of the RVC framework can be used to develop different cryptographic algorithms and security applications.

Outline: The rest of the paper is organized as follows. In Sec. 2 we will give a brief overview of related work, focusing on a comparison between RVC and other existing dataflow solutions. Sec. 3 gives an overview of the building blocks of the RVC framework and Sec. 4 describes the design principles of CTL and the cryptosystems that are already implemented. In Sec. 5, we give performance benchmarks of SHA-256 and AES implemented in CTL on a single-core and a quad-core machine. In Sec. 6, we conclude the paper by giving directions for future works.

2 Related Work

Many cryptographic libraries have been developed over the years (e.g., [17, 26, 32, 45, 50, 60, 61, 68, 69]), but very few can support multiple programming languages. Some libraries do support more than one programming language, but often in the form of separate sets of source code and separate programming interfaces/APIs [68], or available as commercial software only [9, 45]. There is also a large body of optimized implementations of cryptosystems in the literature [19, 20, 23, 48, 49, 59, 72], which normally depend even more on the platforms (e.g., the processor architecture and/or special instruction sets [30, 49, 71, 72]).

Despite being a rather new standard, the RVC framework has been successfully used to develop different kinds of data-driven systems especially multimedia (video, audio, image and graphics) codecs [13–15,21,39] and multimedia security applications [11]. In [11], we highlighted some challenges being faced by developers while building multimedia security applications in imperative languages and discussed how those challenges can be addressed by developing multimedia security applications in the RVC framework. In addition, we presented three multimedia security applications (joint H.264/MPEG-4 video encoding and decoding, joint JPEG image encoding and decoding and compressed domain JPEG image watermark embedding and detecting) developed using the CTL cryptosystems and the RVC implementations of H.264/MPEG-4 and JPEG codecs. Considering the focus of that paper, we only used and briefly summarized CTL. In this paper, we give a detailed discussion on CTL, its design principles, features and benefits, and performance benchmarking results.

The wide usage of RVC for developing multimedia applications is not the only reason why we chose it for developing CTL. A summary of advantages of RVC over other solutions is given in Table 1 (this is an extension of the table in [11]). We emphasize that this comparison focuses on the features relevant to achieve the goals of CTL, so it should not be considered as an exhaustive overview of all pros and cons of the solutions compared.

3 Reconfigurable Video Coding (RVC)

The RVC framework was standardized by the ISO/IEC (via its working group JTC1 / SG29 / WG11, better known as MPEG – Motion Picture Experts Group [52]) to meet the technical challenges of developing more

Table 1: Comparison of RVC framework with other candidate solutions. Candidates with similar characteristics are grouped together. These categories include 1) high-level specification languages for hardware programming languages, 2) frameworks for hardware/software co-design, 3) commercial products, and 4) other cryptographic libraries. The columns in the table represent the following features: A) high-level (abstract) modeling and simulation; B) platform independence; C) code analyzability (i.e., semi-automated design-space exploitation); D) hardware code generation; E) software code generation; F) hardware/software co-design; G) supported target languages; H) open-source or free implementations; I) international standard.

Cat.	Candidate	A	B	C	D	E	F	G	H	I
	RVC	✓	✓	✓	✓	✓	✓	C, C++, Java, LLVM, Verilog, VHDL, XLIM, PROMELA	✓	✓
1	Handel-C [43]	✗	✗	✗	✓	✗	✗	VHDL	✗	✗
	ImpulseC [16]	✗	✗	✗	✓	✗	✓	VHDL	✗	✗
	Spark [31]	✗	✗	✗	✓	✗	✓	VHDL	✗	✗
2	BlueSpec [53]	✓	✗	✓	✓	✓	✗	C, Verilog	✗	✗
	Daedalus [70]	✓	✓	✓	✓	✓	✓	C, C++, VHDL	✓	✗
	Koski [42]	✓	✓	✓	✓	✓	✓	C, XML, VHDL	✗	✗
	PeaCE [33]	✓	✓	✓	✓	✓	✓	C, C++, VHDL	✓	✗
3	CoWare [63]	✓	✓	✗	✓	✓	✓	C, VHDL	✗	✗
	Esterel [1]	✗	✓	✗	✓	✓	✗	C, VHDL	✓	✗
	LabVIEW [4]	✓	✓	✓	✗	✗	✗	-	✗	✗
	Simulink [5]	✓	✓	✓	✓	✓	✗	C, C++, Verilog, VHDL	✗	✗
	Synopsys System Studio [8]	✓	✓	✓	✓	✓	✓	C++, SystemC, SystemVerilog	✗	✗
4	CAO [10, 51]	✓	✓	✗	✗	✓	✗	C, x86-64 assembly, ARM	✗	✗
	Cryptol [9, 45]	✓	✓	✓	✓	✓	✗	C, C++, Haskell, VHDL, Verilog	✗	✗

and more complicated video codecs [37, 38]. One main concern of the MPEG is how to make video codecs more reconfigurable, meaning that codecs with different configurations (e.g., different video coding standards, different profiles and/or levels, different system requirements) can be built on the basis of a single set of platform-independent building blocks. To achieve this goal, the RVC standard defines a framework that covers different steps of the whole life cycle of video codec development. The RVC community has developed supporting tools [2, 6, 7] to make the RVC framework not only a standard, but also a real development environment.

While the RVC framework is developed in the context of video coding, it is actually a general-purpose framework that can model any data-driven applications such as cryptosystems. It allows developers to work with a single platform-independent design at a higher level of abstraction while still being able to generate multiple editions of the same design that target different platforms like embedded systems, general-purpose PCs, and FPGAs. In principle, the RVC framework also supports hardware-software co-design by converting parts of a design into software and other parts into hardware. Additionally, the RVC framework is based on two languages that allow automatic code analysis to facilitate large-scale design-space exploitation like enhancing parallelism of implementations running on multi-core and many-core systems [15, 22, 47].

The RVC standard is composed of two parts: MPEG-B Part 4 [38] and MPEG-C Part 4 [37]. MPEG-B Part 4 specifies the dataflow framework for designing and/or reconfiguring video codecs, and MPEG-C Part 4 defines a video tool library that contains a number of Functional Units (FUs) as platform-independent building blocks of MPEG standard compliant video codecs [37]. To support the RVC dataflow framework, MPEG-B

Part 4 specifies three different languages: a dataflow programming language called RVC-CAL for describing platform-independent FUs, an XML dialect called FNL (FU Network Language) for describing connections between FUs, and another XML dialect called RVC-BSDL for describing the syntax format of video bitstreams. RVC-BSDL is not involved in this work, so we will not discuss it further.

The real core of the RVC framework is RVC-CAL, a general-purpose dataflow programming language for specifying platform-independent FUs. RVC-CAL is a subset of another existing dataflow programming language CAL (Caltrop Actor Language) [28]. In RVC-CAL, FUs are implemented as actors containing a number of fireable actions and internal states. Figure 1 shows the internal structure of a RVC-CAL actor in an FU network. In the RVC-CAL's term, the data exchanged among actors are called tokens. Each actor can contain both input and output port(s) that receive input token(s) and produce output token(s), respectively. Each action may fire depending on four different conditions: 1) input token availability; 2) guard conditions; 3) finite-state machine based action scheduling; 4) action priorities. In RVC-CAL, actors are the basic functional entities that can run in parallel, but actions in an actor are atomic, meaning that only one action can fire at one time. This structure gives a balance between modularity and parallelism, and makes automatic analysis of actor merging/splitting possible.

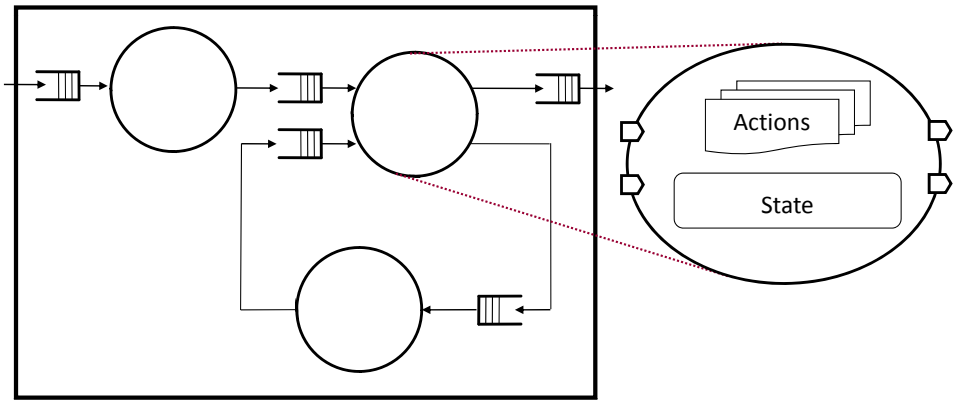


Fig. 1: The internal structure of a RVC-CAL actor in a dataflow network.

Figure 2 illustrates how an application can be modeled and how target implementations can be generated with the RVC framework. At the design stage, different FUs (if not implemented in any standard library) are first written in RVC-CAL to describe their I/O behavior, and then an FU network is built to represent the functionality of a whole application. The FU network can be built by simply connecting all FUs involved graphically via a supporting tool called Graphiti Editor [2], which translates the graphical FU network description into a textual description written in FU Network Language (FNL). The FUs and the FU network are instantiated to form an abstract model. This abstract model can be simulated to test its functionality without going to any specific platform. Two available supporting tools allowing the simulation are OpenDF [6] and ORCC [7]. At the implementation stage, the source code written in other target programming languages can be generated from the abstract application description *automatically*. OpenDF includes a Verilog HDL code generation backend, and ORCC contains a number of code generation backends for C, C++, Java, LLVM, VHDL, XLIM, and PROMELA. ORCC is currently more widely used in the RVC community and it is also the choice of our work reported in this paper.

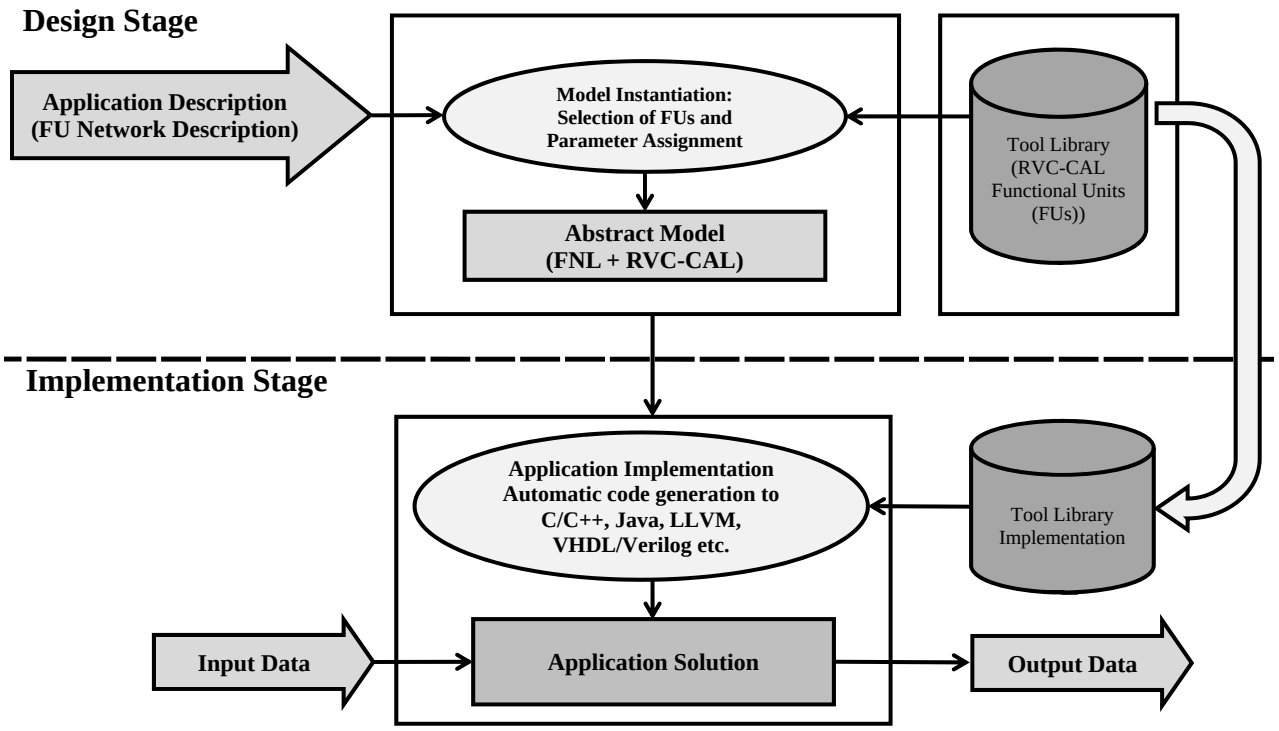


Fig. 2: Process of application implementation generation in the RVC framework.

4 Crypto Tools Library (CTL)

Crypto Tools Library (CTL) is a collection of RVC-CAL actors and XDF networks for cryptographic primitives such as block ciphers, stream ciphers, cryptographic hash functions and PRNGs (see Sec. 4.2 for a list of currently implemented algorithms). Being an open project, the source code and documentation of CTL is available at `http://www.hooklee.com/default.asp?t=CTL`.

As mentioned in Sec. 1, most existing cryptographic libraries are developed based on a single programming language (mostly C/C++ or Java) that can hardly be converted to other languages. In contrast, CTL is a platform-independent solution whose source code is written in RVC-CAL and FNL that can be automatically translated into multiple programming languages (C, C++, Java, LLVM, Verilog, VHDL, XLIM, PROMELA). More programming languages can be supported by developing new code generation tools for RVC applications.

4.1 Design Principles

The CTL is developed by strictly following the specifications/standards defining the implemented cryptosystems. For block ciphers, both enciphers and deciphers are implemented so that a complete security solution can be built. When it is possible, the CTL FUs are designed to exploit inherent parallelism in the implemented cryptosystems. For instance, for block ciphers based on multiple rounds, the round number is also transmitted among different FUs so that encryption/decryption of different blocks can be parallelized.

The CTL is designed so that different cryptosystems can share common FUs. We believe that this can help enhance code reusability and ease reconfigurability of the CTL cryptosystems. In addition, CTL includes

complete solutions (e.g., both encipher and decipher) of the implemented cryptosystems, normally a set of CAL and XDF files.

4.2 Cryptosystems Covered

CTL contains some standard and frequently used cryptosystems. In the following, we list the cryptosystems currently implemented in CTL. The correctness of all cryptosystems has been validated using the test vectors given in the respective standards.

- Block Ciphers:
  - AES-128/192/256 [55],
  - DES [54] and Triple DES [54, 56],
  - Blowfish [64],
  - Modes of operations: CBC, CFB, OFB, CTR.
- Stream Ciphers: ARC4 [65] and Rabbit [25].
- Cryptographic hash functions: SHA-1, SHA-2 (SHA-224, SHA-256) [57].
- PSNRs: 32-bit and 64-bit LCG [65] and LFSR-based PRNG [65].

CTL also includes some common utility FUs (e.g., multiplexing/demultiplexing of dataflows, conversion of bytes to bits and vice versa etc.) that are shared among different cryptosystems and can also find applications in non-cryptography systems. Currently implemented utilities are listed below. In addition, we also present AES, DES and Blowfish as three exemplar cryptosystems from CTL in the Appendix.

- `XOR_1b` and `XOR_8b`: bitwise and bytewise XOR of two token sequences;
- `Mux2` and `Mux8`: merging 2 and 8 sequences of tokens into a single one;
- `Demux2` and `Demux8`: splitting a token sequence into 2 and 8 sub-sequences;
- `Any2Bits`: converting n-bit tokens into binary (i.e., 1-bit) tokens;
- `Bit2Any`: converting binary tokens into n-bit tokens;
- `Smaller2Bigger`: converting n_2/n_1 input tokens of bit size n_1 into one output token of bit size $n_2 > n_1$;
- `Bigger2Smaller`: converting each input token of bit size n_1 into n_1/n_2 output tokens of bit size $n_1 > n_2$.

In each RVC-CAL file of CTL FUs and testbeds, there is a header comments section giving detailed information about that RVC-CAL file: FU name, FU interface (input ports, output ports, FU parameters), how to use the CAL file, reference to corresponding standard document, and so forth. Furthermore, under each folder there is also a readme file containing a list of all files in the corresponding folder.

5 Performance Benchmarking of CTL

Previous work has demonstrated that the RVC framework can outperform other sequential programming languages in terms of implementing highly complex and highly parallelizable systems like video codecs [21]. However, there are still doubts about if the high-level abstraction of RVC-CAL and the automated code generation process may compromise the overall performance to some extent at the platform level. In this section, we clarify those doubts by showing that the automatically generated implementations from a typical RVC-based application can usually achieve a performance comparable to manually-written implementations in the target programming language. This was verified on SHA-256 (Sec. 5.1) and AES (Sec. 5.2) applications in CTL. In Sec. 5.1, we take SHA-256 as an example to show how we did the benchmarking on a single-core machine and a quad-core one. The main purpose of getting the quad-core machine involved is to show how easy one can divide an FU network and map different parts to different cores to make a better use of the computing

Table 2: Configuration of the test machine.

Machine	**Hardware and Operating System Details**
Desktop PC:	– Model: HP Centurion – CPU: Intel(R) Core(TM)2 Quad CPU Q9550 2.83GHz – Memory: 8GB RAM – OS1: Windows Vista Business with Service Pack 2 (64-bit Edition) – OS2: Ubuntu Linux (Kernel version: 2.6.27.11)

resources. In the given example, the partitioning and mapping were both done manually, but they can be automated for large applications thanks to the code analyzability of RVC-CAL. In addition, Sec. 5.2 presents the results of the performance benchmarking of AES on varying single-core platforms (general-purpose PCs, a resource-constrained embedded device and Java virtual machines).

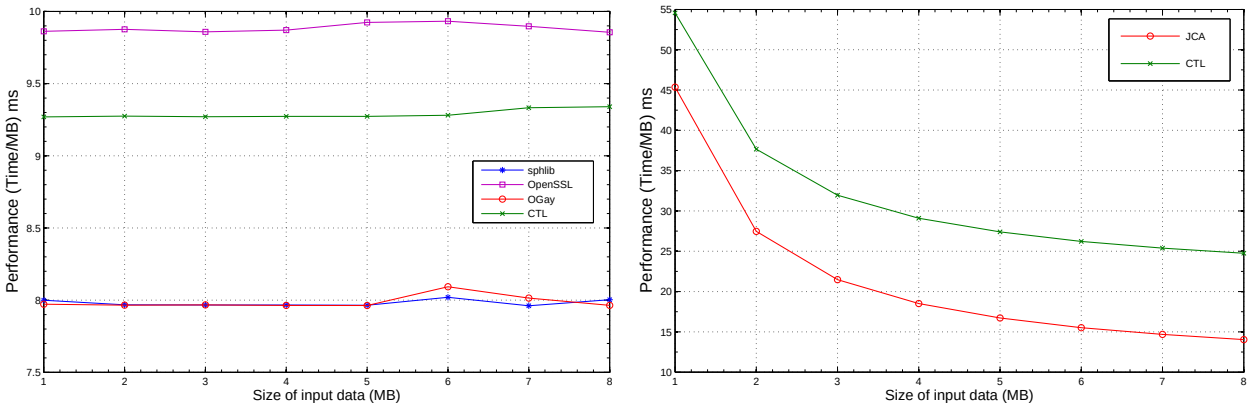
5.1 Benchmarking of SHA-256

We ran our experiments on Microsoft both Windows and Linux (see Table 2 for details). Both operating systems support high resolution timers to measure time in nanoseconds. More specifically, we used the `QueryPerformanceCounter()` and `QueryPerformanceFrequency()` functions (available from Windows API) on Windows, and the `clock_gettime()` and `clock_getres()` functions with `CLOCK_MONOTONIC` clock (available from the Higher Resolution Timer [24] package) on Linux. In addition, to circumvent the caching problem, we conducted 100 independent runs (with random input data) of each configuration and used the average value as the final performance metric.

The concrete specifications of our test machines can be found in Table 2. Due to the multi-tasking nature of Windows and Linux operating systems, the benchmarking result can be influenced by other tasks running in parallel. In order to minimize this effect, we conducted all our experiments under the safe mode of both OSs. We used Microsoft Visual Studio 2008 and GCC 4.3.2 as C compilers for the Windows and the Linux operating systems, respectively. Both compilers were configured to maximize the speed of generated executables. For Java programs, we used Eclipse SDK 3.6.1 and Java(TM) SE Runtime Environment (build 1.6.0_12-b04).

Benchmarking of SHA-256 on Single-Core Platform In this subsection, we present the results of benchmarking a single SHA-256 FU against some non-RVC reference implementations in C (OpenSSL [69], OGay [29], and sphlib [60]) and Java (Java Cryptography Architecture (JCA) [58]). Figure 3 shows the results of our benchmarking under Windows operating system while our test machine was configured to run only one CPU core. One can see that the run-time performance of CTL implementation is better than OpenSSL but inferior to carefully optimized (OGay and sphlib) implementations. In addition, the CTL's Java implementation of SHA-256 does not outperform the JCA implementation. This can probably be explained by the fact that the current edition of the ORCC Java backend does not generate very efficient code. These results indicate that the CTL's SHA-256 implementation can achieve a performance similar to reference implementations. We also did similar benchmarking experiments on the AES block cipher in CTL (presented in the next subsection Sec. 5.2) and came to a similar conclusion.

Benchmarking of SHA-256 on Multi-Core Platform On a platform with multiple CPU cores, one can map different parts of an FU network to different CPU cores so that the overall run-time performance of the application can be improved. The C backend of the RVC supporting tool ORCC [7] supports multi-core mapping, so one can easily allocate different FUs or FU sub-networks to different CPU cores. To see



(a) C Implementations (b) Java Implementations

Fig. 3: Benchmarking of CTL's SHA-256 implementation.

how much benefit we can get from a multi-core platform, we devised a very simple RVC application called HashTree (shown in Fig. 4) that implements the following functionality using five hash H operations: given an input signal $x = x_1 \parallel x_2 \parallel x_3 \parallel x_4$ consisting of four blocks x_i, hash each block $h_i = H(x_i)$ and then output $H(h_1 \parallel h_2 \parallel h_3 \parallel h_4)$. In our implementation of HashTree, we instantiated H with SHA-256. By comparing this application with the simple single-core SHA-256 application computing H on the same input (i.e., $H(x_1 \parallel x_2 \parallel x_3 \parallel x_4)$), we can roughly estimate the performance gain.

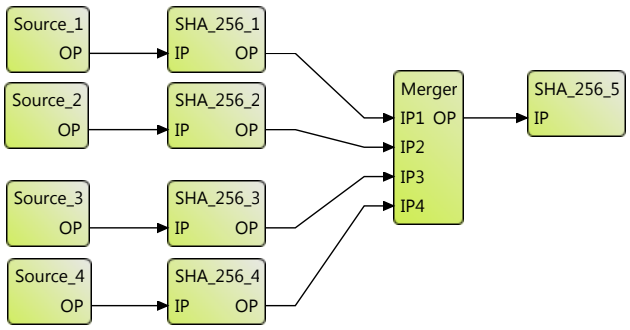
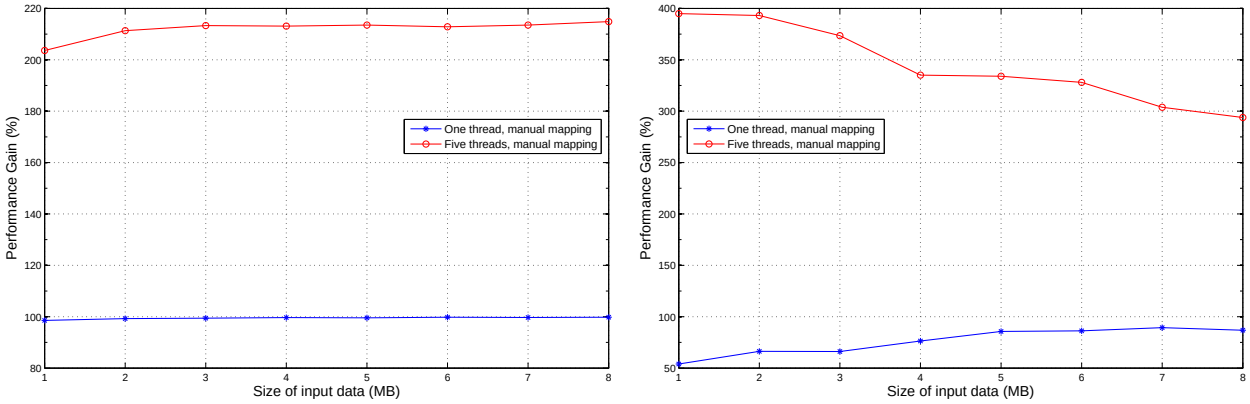


Fig. 4: HashTree Application.

In the benchmarking process, we considered three different configurations:

- **Single SHA-256:** This configuration represents a single SHA-256 FU running on a single-core, which processes an input x and produces the hash. We used this configuration as the reference point to evaluate the performance gain of the following two configurations, which implement HashTree using five SHA-256 instances.
- **5-thread with manual mapping:** In this configuration, each SHA-256 instance is programmatically mapped to run as a separate thread on a specific CPU core of our quad core machine. At the start of the



(a) Windows (b) Linux

Fig. 5: The performance gain we got from the benchmarked configurations.

hashing process, we manually mapped the 4 threads (processing $h_i = H(x_i)$) to four CPU cores. The 5th thread performing the final hashing operation is created and mapped after the preceding 4 threads are finished with their execution.

– **1-thread with manual mapping:** Similar to above configuration, this configuration also implements HashTree. However, all five SHA-256 instances are bounded to run in a single thread on a specific CPU core of our quad core machine.

It should be noted that thread creation and mapping also consume some CPU time, which is the cost one has to pay to achieve concurrency. Therefore, in order to make the study judicial, we also count the times spent on thread creation and thread mapping.

The benchmarking results are shown in Fig. 5. One can see that the performance gain is between 200% to 300% when five threads are used.

5.2 Benchmarking of AES

In this section, we present the performance benchmarking of AES against some reference implementations on varying single-core platforms (two general-purpose PCs, a resource-constrained embedded device and two Java virtual machines). In this section, we first give the details of our experimental setup required to reproduce our results. The last subsection presents the performance benchmarking results.

Tested AES implementations As presented in Appendix A.1, currently CTL contains two different implementations of AES targeting two different objectives. One implementation is created for educational/understanding viewpoint and has been implemented by strictly following the AES standard [55]. The second implementation has been created to achieve better run-time performance and has been implemented by following the look-up-tables based optimized algorithm used in the Rijndael's optimized reference implementation [62]. Both of these AES implementations are benchmarked in this study. In the rest of this subsection, we will respectively use "CTL-STD" and "CTL-LUT" as the short names for these two AES implementations in CTL.

Along with AES ECB encipher and decipher, AES running in CTR mode (shown in Fig. 8d) is also included for this benchmarking study because the CTR mode has the benefit of being able to encrypt multiple blocks

in parallel, so it can be considered as a better candidate for benchmarking cryptosystems implemented in RVC-CAL. Hence, the following three CTL implementations of AES-128 were benchmarked in our study:

- AES-128 CTR Cipher
- AES-128 ECB Encipher
- AES-128 ECB Decipher

Reference implementations To benchmark the performance of the ORCC generated C code of AES-128, some reference implementations are needed to compare with. For this study, the following three implementations are selected:

- Rijndael reference implementation ver. 2.2 [18]
- AES implementation available at www.X-N2O.com [73]
- Rijndael optimized reference implementation ver. 3.0 [62]

In the rest of this subsection, "Ref. 2.2", "Ref. 3.0" and "X-N2O" are the short names used to refer to the three reference implementations of AES-128. Not all of these three implementations support CTR mode, so the code has been manually modified to add CTR support.

We select these three implementations because to make this performance benchmarking study judicial, we need the reference implementations that follow the same implementation style and optimizations as our CTL implementations. For instance, Ref.2.2 and X-N2O implementations are similar to our CTL-STD implementation because they do not contain any optimizations and are implemented by following the AES standard. Similarly, Ref. 3.0 implementation is similar to our CTL-LUT implementation as both are optimized by using pre-computed look-up tables.

Similarly, we also benchmarked the ORCC generated Java code of AES-128 against the AES implementation available as part of the Java Cryptography Architecture (JCA) [58].

Platforms Our experiments were run on two PCs and one embedded system. This was done to represent two typical configurations of today's PCs, one new desktop and one old laptop are selected. For embedded systems, we selected a resource constrained wireless sensor mote for our study. For both PCs, we have conducted this performance evaluation under Windows and Linux operating systems whilst our embedded system runs a stripped-down version of Linux operating system. The concrete configurations of these platforms are given Table 3.

M1 has a dual-core CPU so the performance benchmarking may be less accurate due to the internal scheduling of CPU instructions. So we switched the dual-core support off in M1's BIOS setup. Furthermore, the multi-task nature of operating systems may also influence the benchmarking results, so we ran all our tests under their Safe Mode shell to minimize such effects.

To generate the executables running under different operating systems, we selected Microsoft Visual Studio 2008 as the C compiler for Windows XP/7, GCC 4.3.2 for Linux Debian Live and arm-linux-gcc 3.4.1 for Imote2-Linux on M3. For Java programs, we used Eclipse SDK 3.6.1.

Run-time performance metric For each executable running under a specific OS, a continuous encryption/decryption process was run over the same test vector of 4096 bytes. We measured the total number of CPU cycles consumed for the whole encryption/decryption process and divided it by 4096 to get the run-time performance in cycles/byte. Since we only care about the core part of the encipher/decipher, the time consumed in initial inputs and final output is not counted.

Table 3: Configurations of Testing Platforms.

Platform	**Hardware and Operating System Details**
Machine 1 (M1): Desktop PC	– Model: HP Compaq 8000 Elite Convertible Minitower – CPU: Intel Pentium Dual-Core CPU E5400 2.70GHz – Memory: 2GB RAM – OS1: Windows 7 Professional (32-bit Edition) – OS2: Linux Debian Live (rescue image kernel version 2.6.26-2-686) – Windows 7 Java(TM) SE Runtime Environment: build 1.6.0_26-b03
Machine 2 (M2): Laptop PC	– Model: Samsung Q25 – CPU: Intel Pentium M 1.3GHz – Memory: 504MB RAM – OS1: Windows XP Professional SP 2 – OS2: Linux Debian Live (rescue image kernel version 2.6.26-2-686) – Windows XP Java(TM) SE Runtime Environment: build 1.6.0_27-b07
Machine 3 (M3): Embedded System	– Model: Imote2 Wireless Sensor Mote [3] – CPU: Intel ARM XScale PXA271 CPU 415.33MHZ [36] – Memory: 32MB SRAM – OS1: Imote2-Linux (Kernel version 2.6.29.1)

On M1 and M2, the CPU cycles were measured using `RDTSC` and `CPUID` instructions of Intel processors [35]. For M3, we used `Cycle CouNT (CCNT)` register available in Intel ARM XScale processors [36]. However, the measured CPU cycles may vary depending on the availability of needed data/instructions in the data/instruction cache. To solve this problem, we follow the suggestion given in [35] to run the same executables for 100 times and use the averaged value of CPU cycles as the final measurement.

Benchmarking Results In this subsection, we present the results of our performance benchmarking study on the CTL implementations of AES-128 and the corresponding reference implementations on all three machines.

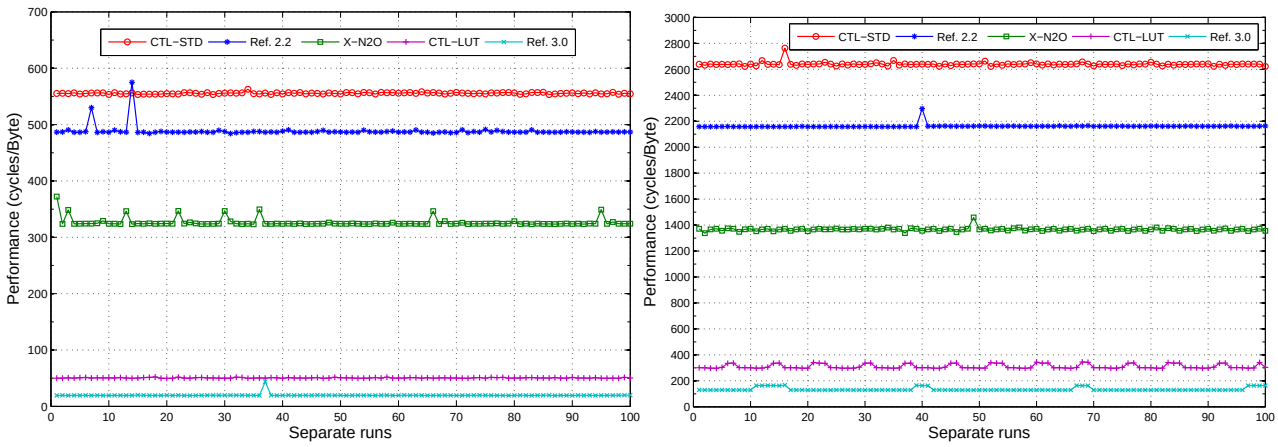
AES-128 CTR Cipher As mentioned in the previous subsection, in order to compensate the cache effects, each executable was run 100 times. Figure 6 shows the results obtained under Windows and Linux operating systems on M1 and M3, respectively.

The results in these plots show some abrupt fluctuations in the performance curve, but they do not occur very often. Most of the times, the CPU cycles counting remains consistent. Therefore, the CPU cycles counting method based on RDTSC and CPUID instructions is stable enough as a metric of the run-time performance for the evaluated implementations.

The average performance results for all implementations on PCs and embedded system are given in Tables 4a and 4b. It can be observed that, under both Windows 7/XP and Linux operating systems, the CTL-STD and CTL-LUT implementations have a performance comparable to Ref. 2.2 and X-N2O, and Ref. 3.0, respectively.

It deserves noticing that all algorithms perform better on M1 than M2 or M3. This can be easily explained by the more powerful CPU and larger memory available on M1. On the contrary, all algorithms consume higher number of CPU cycles on M3 because of its limited resources.

AES-128 ECB Encipher Tables 4a and 4b also contain the performance benchmarking results on AES-128 ECB encipher. One can see that the results of CTL-STD implementation in ECB mode are similar to the



(a) On general-purpose PC – M1. (b) On embedded system – M3.

Fig. 6: Separate Runs of AES-128 CTR cipher under Windows and Linux OS.

corresponding CTL-STD implementation in CTR mode. In addition, unlike for the AES CTR cipher, the performance of CTL-LUT encipher implementation in ECB mode is quite close to the corresponding Ref.3.0 implementation. This sudden performance gain in the ECB mode can be explained by the design of ECB and CTR ciphers in the RVC framework. In CTR mode, the encryption process was jointly performed by CTR and AES Cipher FUs while at any given time only one of them processing data and other one waiting for the data (i.e., when the CTR FU processes data, AES Cipher waits for the data and vice versa). However, in ECB mode the whole encryption process is encapsulated within a single FU and saves a considerable time that could be depleted in waiting for the dataflows from other FUs. Based on these results it can safely established that: 1) if the needed optimization are implemented within the same FU, the run-time performance similar to reference implementations can be achieved; 2) on single-core machines, the introduction of dataflow networks between FUs affects the run-time performance. In other words, these results suggest that the run-time performance of the RVC applications on single-core machines is inversely related to the number of intermediate FUs (and FIFOs). However, very small RVC applications (like cryptosystems) can be implemented within a single FU. With the increase in the algorithmic complexity of applications, it become difficult to implement in fewer FUs, which in a way also creates an opportunity to achieve better run-time performance by parallelizing the FUs on multi-core machines. In the next section, this point is further highlighted while presenting the performance benchmarking study on a quad-core machine.

AES-128 ECB Decipher Tables 4a and 4b also give the performance benchmarking results on AES-128 ECB decipher. It can be seen that CTL-STD's decipher implementation also has a performance similar to Ref. 2.2 and X-N2O. In addition, for CTL-LUT's decipher implementation, we obtained the results similar to the results of AES-128 ECB encipher.

Benchmarking of AES Java Implementations Since the C implementations of CTL-LUT achieved reasonable performance, we also benchmarked the Java implementation of CTL-LUT against the AES implementation available as part of the Java Cryptography Architecture (JCA) [58]. Tables 4c gives the results of the performance benchmarking, which was conducted under the Windows 7/XP Java run-time environment of M1

(a) Averaged performance benchmarking of AES-128 C implementations on PCs.

CTR Cipher	M1					M2				
	CTL-STD	**Ref.2.2**	**X-N2O**	**CTL-LUT**	**Ref.3.0**	**CTL-STD**	**Ref.2.2**	**X-N2O**	**CTL-LUT**	**Ref.3.0**
Win XP/7	555.6	488.4	326.4	50.5	19.8	637.2	1292.0	393.7	65.4	24.4
Linux	3979.6	1632.4	353.1	71.6	37.0	4711.8	1991.3	581.7	77.3	41.5

ECB Encipher	M1					M2				
	CTL-STD	**Ref.2.2**	**X-N2O**	**CTL-LUT**	**Ref.3.0**	**CTL-STD**	**Ref.2.2**	**X-N2O**	**CTL-LUT**	**Ref.3.0**
Win XP/7	418.8	484.5	296.3	21.8	18.8	472.2	1296.2	374.1	27.2	23.5
Linux	3088.1	1599.4	1439.6	36.3	34.7	4205.2	1948.3	1558.1	41.7	37.8

ECB Decipher	M1					M2				
	CTL-STD	**Ref.2.2**	**X-N2O**	**CTL-LUT**	**Ref.3.0**	**CTL-STD**	**Ref.2.2**	**X-N2O**	**CTL-LUT**	**Ref.3.0**
Win XP/7	719.1	668.9	544.4	21.8	19.3	1769.9	1899.9	1572.1	26.6	23.8
Linux	4333.6	2229.4	1761.8	37.9	35.2	5630.2	1463.7	2641.2	42.0	37.9

(b) Averaged performance benchmarking of AES-128 C implementations on an embedded system.

	CTL-STD	**Ref.2.2**	**X-N2O**	**CTL-LUT**	**Ref.3.0**
CTR Cipher	2639.8	2161.8	1366.1	311.1	135.0
ECB Encipher	2058.1	2167.6	1364.2	147.6	132.1
ECB Decipher	4706.4	4231.9	2982.7	147.7	129.3

(c) Averaged performance benchmarking of AES-128 Java implementations on PCs.

	M1		M2	
	CTL-LUT	**JCA**	**CTL-LUT**	**JCA**
CTR Cipher	3926.5	1391.8	4632	1581.2
ECB Encipher	2820.3	2820.3	3731.5	1362
ECB Decipher	2791.2	779.5	3798.1	850.6

and M2. As we observed in the previous subsection for SHA-256, ORCC Java backend does not generate very efficient code at this moment and these benchmarking results are not surprising.

6 Future Works

In order to allow researchers from different fields to extend CTL and use it for more applications, we have published CTL as an open-source project at `http://www.hooklee.com/default.asp?t=CTL`. In our future work, we plan to continue our research on the following possible directions.

Cryptographic Primitives. The CTL can be enriched by including more cryptographic primitives (especially public-key cryptography), which will allow creation of more multimedia security applications and security protocols. Another direction is to develop optimized versions of CTL cryptosystems. For instance, bit slicing can be used to optimize parallelism in many block ciphers [30, 49].

Security Protocols. Another direction is to use the RVC framework for the design and development of security protocols and systems with heterogenous components and interfaces. While RVC itself is platform independent,

"wrappers" [67] can be developed to bridge the platform-independent FUs with physical I/O devices/channels (e.g., a device attached to USB port, a host connected via LAN/WLAN, a website URL, etc.). Although there are many candidate protocols that can be considered, as a first step we plan to implement the hPIN/hTAN e-banking security protocol [46], which is a typical (but small-scale) heterogeneous system involving a hardware token, a web browser plugin on the user's computer, and a web service running on the remote e-banking server. We have already implemented an hPIN/hTAN prototype system without using RVC, so the new RVC-based implementation can be benchmarked against the existing system.

Cryptographic Protocols. Many cryptographic protocols require a high amount of computations. One example are garbled circuit protocols [74] that allow secure evaluation of an arbitrary function on sensitive data. These protocols can be used as basis for various privacy-preserving applications. On a high-level, the protocol works by one party first generating an encrypted form of the function to be evaluated (called garbled circuit) which is then sent to the other party who finally decrypts the function using the encrypted input data of both parties and finally obtains the correct result. Recent implementation results show that such garbled circuit-based protocols can be implemented in a highly efficient way in software [34]. However, until now, there exist no software implementations that exploit multi-core architectures. It was shown that such protocols can be optimized when using both software and hardware together: For generation of the garbled circuit, a trusted hardware token can generate the garbled circuit locally and hence remove the need to transfer it over the Internet [40]. Here, the encrypted versions of the gate which require four invocations of a cryptographic hash function can be computed in parallel similar to the 4-adic hash tree we have shown in Sec. 5. Furthermore, the evaluation of garbled circuits can be improved when using hardware accelerations as shown in [41]. We believe that the RVC framework can serve as an ideal basis for hardware-software co-designed systems with parallelized and/or hardware-assisted garbled circuit-based protocols.

References

1. Esterel Synchronous Language. `http://www-sop.inria.fr/esterel.org/files/`
2. Graphiti. `http://graphiti-editor.sf.net`
3. Intel® Mote 2 engineering platform, a copy available at `http://ubi.cs.washington.edu/files/imote2/docs/imote2-ds-rev2.0.pdf`
4. LabVIEW. `http://www.ni.com/labview/whatis/`
5. Mathworks Simulink: Simulation and Model-Based Design. `http://www.mathworks.com/products/simulink/`
6. Open Data Flow (OpenDF). `http://sourceforge.net/projects/opendf`
7. Open RVC-CAL Compiler (ORCC). `http://sourceforge.net/projects/orcc`
8. Synopsys Studio. `http://www.synopsys.com/SYSTEMS/BLOCKDESIGN/DIGITALSIGNALPROCESSING/Pages/SystemStudio.aspx`
9. Cryptol: The Language of Cryptography. Case Study, `http://corp.galois.com/downloads/cryptography/Cryptol_Casestudy.pdf` (2008)
10. CAO and `qhasm` compiler tools. EU Project CACE deliverable D1.3, Revision 1.1, `http://www.cace-project.eu/downloads/deliverables-y3/32_CACE_D1.3_CAO_and_qhasm_compiler_tools_Jan11.pdf` (2011)
11. Ahmad, J.J., Li, S., Amer, I., Mattavelli, M.: Building multimedia security applications in the MPEG Reconfigurable Video Coding (RVC) framework. In: Proc. 2011 ACM SIGMM Multimedia and Security Workshop (MM&Sec 2011) (2011)
12. Akyildiz, I.F., Melodia, T., Chowdhury, K.R.: Wireless multimedia sensor networks: Applications and testbeds. Proc. IEEE 96(10), 1588–1605 (2008)
13. Ali, H.I.A.A., Patoary, M.N.I.: Design and Implementation of an Audio Codec (AMR-WB) using Dataflow Programming Language CAL in the OpenDF Environment. TR: IDE1009, Halmstad University, Sweden (2010)
14. Aman-Allah, H., Maarouf, K., Hanna, E., Amer, I., Mattavelli, M.: CAL dataflow components for an MPEG RVC AVC baseline encoder. J. Signal Processing Systems 63(2), 227–239 (2011)

15. Amer, I., Lucarz, C., Roquier, G., Mattavelli, M., Raulet, M., Nezan, J., Déforges, O.: Reconfigurable Video Coding on multicore: An overview of its main objectives. IEEE Signal Processing Magazine 26(6), 113–123 (2009)
16. Antola, A., Fracassi, M., Gotti, P., Sandionigi, C., Santambrogio, M.: A novel hardware/software codesign methodology based on dynamic reconfiguration with Impulse C and CoDeveloper. In: Proc. 2007 3rd Southern Conference on Programmable Logic (SPL 2007). pp. 221–224 (2007)
17. Barbosa, M., Noad, R., Page, D., Smart, N.P.: First steps toward a cryptography-aware language and compiler. Cryptology ePrint Archive: Report 2005/160, http://eprint.iacr.org/2005/160.pdf (2005)
18. Barreto, P., Rijmen, V.: Rijndeal reference implementation (+ KATs and MCTs) v2.2. Public domain software (1999)
19. Bernstein, D.J., Schwabe, P.: New AES software speed records. In: Progress in Cryptology – INDOCRYPT 2008. LNCS, vol. 5365, pp. 322–336 (2008)
20. Bertoni, G., Breveglieri, L., Fragneto, P., Macchetti, M., Marchesin, S.: Efficient software implementation of AES on 32-bit platforms. In: Cryptographic Hardware and Embedded Systems – CHES 2002. LNCS, vol. 2523, pp. 159–171 (2002)
21. Bhattacharyya, S., Eker, J., Janneck, J.W., Lucarz, C., Mattavelli, M., Raulet, M.: Overview of the MPEG Reconfigurable Video Coding framework. J. Signal Processing Systems 63(2), 251–263 (2011)
22. Boutellier, J., Gomez, V.M., Silvén, O., Lucarz, C., Mattavelli, M.: Multiprocessor scheduling of dataflow models within the Reconfigurable Video Coding framework. In: Proc. 2009 Conference on Design and Architectures for Signal and Image Processing (DASIP 2009) (2009)
23. Canright, D., Osvik, D.A.: A more compact AES. In: Selected Areas in Cryptography (SAC 2009). LNCS, vol. 5867, pp. 157–169 (2009)
24. Corbet, J.: The high-resolution timer (API). http://lwn.net/Articles/167897 (2006)
25. Cryptico A/S: Rabbit stream cipher, performance evaluation. White Paper, Version 1.4, available online at http://www.cryptico.com/DWSDownload.asp?File=Files%2FFiler%2FWP%5FRabbit%5FPerformance%2Epdf (2005)
26. Dai, W.: Crypto++ library. http://www.cryptopp.com
27. Dennis, J.: First version of a data flow procedure language. In: Programming Symposium, Proceedings Colloque sur la Programmation Paris, April 9-11, 1974, LNCS, vol. 19, pp. 362–376 (1974)
28. Eker, J., Janneck, J.W.: CAL language report: Specification of the CAL actor language. Technical Memo UCB/ERL M03/48, Electronics Research Laboratory, UC Berkeley (2003)
29. Gay, O.: SHA-2: Fast Software Implementation. http://www.ouah.org/ogay/sha2
30. Grabher, P., Großschädl, J., Page, D.: Light-weight instruction set extensions for bit-sliced cryptography. In: Cryptographic Hardware and Embedded Systems – CHES 2008. LNCS, vol. 5154, pp. 331–345 (2008)
31. Gupta, S., Dutt, N., Gupta, R., Nicolau, A.: SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In: Proc. 2003 16th International Conference on VLSI Design (VLSI Design 2003) (2003)
32. Gutmann, P.: Cryptlib. http://www.cs.auckland.ac.nz/~pgut001/cryptlib
33. Ha, S., Kim, S., Lee, C., Yi, Y., Kwon, S., Joo, Y.P.: PeaCE: A hardware-software codesign environment for multimedia embedded systems. ACM Trans. on Design Automation of Electronic Syststems 12(3), Article 24 (2007)
34. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: Proc. 20th USENIX Security Symposium (2011)
35. Intel Corporation: Using the RDTSC instruction for performance monitoring (1997), a copy available at http://www.ccsl.carleton.ca/~jamuir/rdtscpm1.pdf
36. Intel Corporation: Intel XScale® core developer's manual (2004)
37. ISO/IEC: Information technology – MPEG video technologies – Part 4: Video tool library. ISO/IEC 23002-4 (2009)
38. ISO/IEC: Information technology - MPEG systems technologies - Part 4: Codec configuration representation. ISO/IEC 23001-4 (2009)
39. Janneck, J., Miller, I., Parlour, D., Roquier, G., Wipliez, M., Raulet, M.: Synthesizing hardware from dataflow programs: An MPEG-4 Simple Profile decoder case study. J. Signal Processing Systems 63(2), 241–249 (2011)
40. Järvinen, K., Kolesnikov, V., Sadeghi, A.R., Schneider, T.: Embedded SFE: Offloading server and network using hardware tokens. In: Financial Cryptography and Data Security (FC 2010). LNCS, vol. 6052, pp. 207–221 (2010)
41. Järvinen, K., Kolesnikov, V., Sadeghi, A.R., Schneider, T.: Garbled circuits for leakage-resilience: Hardware implementation and evaluation of one-time programs. In: Cryptographic Hardware and Embedded Systems – CHES 2010. LNCS, vol. 6225, pp. 383–397 (2010)

42. Kangas, T., Kukkala, P., Orsila, H., Salminen, E., Hännikäinen, M., Hämäläinen, T.D., Riihimäki, J., Kuusilinna, K.: UML-based multiprocessor SoC design framework. ACM Trans. on Embedded Compututer Systems 5, 281–320 (2006)
43. Khan, E., El-Kharashi, M.W., Gebali, F., Abd-El-Barr, M.: Applying the Handel-C design flow in designing an HMAC-hash unit on FPGAs. Computers and Digital Techniques 153(5), 323–334 (2006)
44. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. Proc. IEEE 75(9), 1235–1245 (1987)
45. Lewis, J.R., Martin, B.: Cryptol: High assurance, retargetable crypto development and validation. In: Proc. 2003 IEEE Military Communication Conference (MILCOM 2003). pp. 820–825 (2003)
46. Li, S., Sadeghi, A.R., Heisrat, S., Schmitz, R., Ahmad, J.J.: hPIN/hTAN: A lightweight and low-cost e-banking solution against untrusted computers. In: Financial Cryptography and Data Security (FC 2011), LNCS, vol. 7035, pp. 235–249 (2011)
47. Lucarz, C., Mattavelli, M., Dubois, J.: A co-design platform for algorithm/architecture design exploration. In: Proc. 2008 IEEE International Conference on Multimedia and Expo (ICME 2008). pp. 1069–1072 (2008)
48. Manley, R., Gregg, D.: A program generator for intel AES-NI instructions. In: Progress in Cryptology – INDOCRYPT 2010. LNCS, vol. 6498, pp. 311–327 (2010)
49. Matsui, M., Nakajima, J.: On the power of bitslice implementation on Intel Core2 processor. In: Cryptographic Hardware and Embedded Systems – CHES 2007. LNCS, vol. 4727, pp. 121–134 (2007)
50. Moran, T.: The Qilin Crypto SDK: An open-source Java SDK for rapid prototyping of cryptographic protocols. `http://qilin.seas.harvard.edu/`
51. Moss, A., Page, D.: Bridging the gap between symbolic and efficient AES implementations. In: Proc. 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2010). pp. 101–110 (2010)
52. Moving Picture Experts Group (MPEG): Who we are. `http://mpeg.chiariglione.org/who_we_are.htm`
53. Nikhil, R.: Tutorial – BlueSpec SystemVerilog: Efficient, correct RTL from high-level specifications. In: Proc. 2nd ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004). pp. 69–70 (2004)
54. NIST: Data Encryption Standard (DES). FIPS PUB 46-3 (1999)
55. NIST: Specification for the Advanced Encryption Standard (AES). FIPS PUB 197 (2001)
56. NIST: Recommendation for the Triple Data Encryption Algorithm (TDEA) block cipher. Special Publication 800-67, Version 1.1 (2008)
57. NIST: Secure Hash Standard (SHS). FIPS PUB 180-3 (2008)
58. Oracle®: Java™ Cryptography Architecture (JCA) Reference Guide. `http://download.oracle.com/javase/6/docs/technotes/guides/security/crypto/CryptoSpec.html`
59. Osvik, D.A., Bos, J.W., Stefan, D., Canright, D.: Fast software AES encryption. In: Fast Software Encryption (FSE 2010). LNCS, vol. 6147, pp. 75–93 (2010)
60. Pornin, T.: sphlib 3.0. `http://www.saphir2.com/sphlib`
61. PureNoise Ltd Vaduz: PureNoise CryptoLib. `http://cryptolib.com/crypto`
62. Rijmen, V., Bosselaers, A., Barreto, P.: Optimised ANSI C code for the cipher (now AES). Rijndael Reference Implementation, Version 3.0, public domain software (2000)
63. Rompaey, K.V., Verkest, D., Bolsens, I., Man, H.D.: CoWare – a design environment for heterogeneous hardware/software systems. Design Automation for Embedded Systems 1(4), 357–386 (1996)
64. Schneier, B.: Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish). In: Fast Software Encryption (FSE'94). LNCS, vol. 809, pp. 191–204 (1994)
65. Schneier, B.: Applied Cryptography: Protocols, algorithms, and source code in C. John Wiley & Sons, Inc., New York, second edn. (1996)
66. Sutherland, W.R.: The On-Line Graphical Specification of Computer Procedures. Ph.D. thesis, MIT (1966)
67. Thavot, R., Mosqueron, R., Dubois, J., Mattavelli, M.: Hardware synthesis of complex standard interfaces using CAL dataflow descriptions. In: Proc. 2009 Conference on Design and Architectures for Signal and Image Processing (DASIP 2009) (2009)
68. The Legion of the Bouncy Castle: Bouncy Castle Crypto APIs. `http://www.bouncycastle.org`
69. The OpenSSL Project: OpenSSL cryptographic library. `http://www.openssl.org/docs/crypto/crypto.html`

70. Thompson, M., Nikolov, H., Stefanov, T., Pimentel, A.D., Erbas, C., Polstra, S., Deprettere, E.F.: A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In: Proc. 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2007). pp. 9–14 (2007)
71. Tillich, S., Großschädl, J.: Instruction set extensions for efficient AES implementation on 32-bit processors. In: Cryptographic Hardware and Embedded Systems – CHES 2006. LNCS, vol. 4249, pp. 270–284 (2006)
72. Tillich, S., Herbst, C.: Boosting AES performance on a tiny processor core. In: Topics in Cryptology – CT-RSA 2008. LNCS, vol. 4964, pp. 170–186 (2008)
73. X-N2O: AES explained. `http://www.x-n2o.com/aes-explained`
74. Yao, A.C.: How to generate and exchange secrets. In: Proc. 27th Annual Symposium on Foundations of Computer Science (FOCS'86). pp. 162–167 (1986)

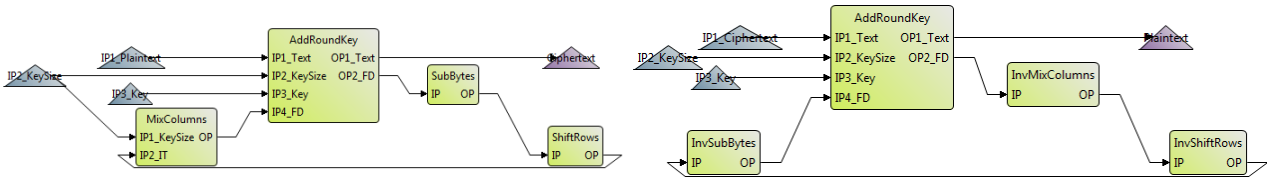
A Exemplar CTL Cryptosystems

With the objective of giving a feel of how CTL cryptosystems look like, in this subsection, we present AES, DES, and Blow fish block ciphers as examples of cryptosystems from CTL. We present the XDF networks for encipher and decipher along with a brief description about the associated FUs.

A.1 AES

CTL includes two different implementations of AES: 1) one for the educational purpose, which has been implemented by strictly following the AES standard [55]; 2) a look-up-tables (LUTs) based optimized implementation following the Rijndael's optimized reference implementation [62]. In the following, we present both of them.

Standard Implementation Figure 7 shows encipher and decipher FU networks of the standard AES implementation in the CTL. Both have three input ports and one output tokens, all of type byte. Thus, AES always consumes 16 byte tokens as plaintext/ciphertext and produces 16 byte tokens as ciphertext/plaintext. The key size and key are always read at the beginning of the encryption/decryption process and remains the same until it is not changed. All the four basic operations are implemented as separate RVC-CAL FUs. The key expansion function (i.e., key scheduler) is implemented as part of the `AddRoundKey` FU since it is not used in other FUs. It should be noted that both AES encipher and decipher have similar structure. However, for AES decipher the four basic components are connected in a reversed order.



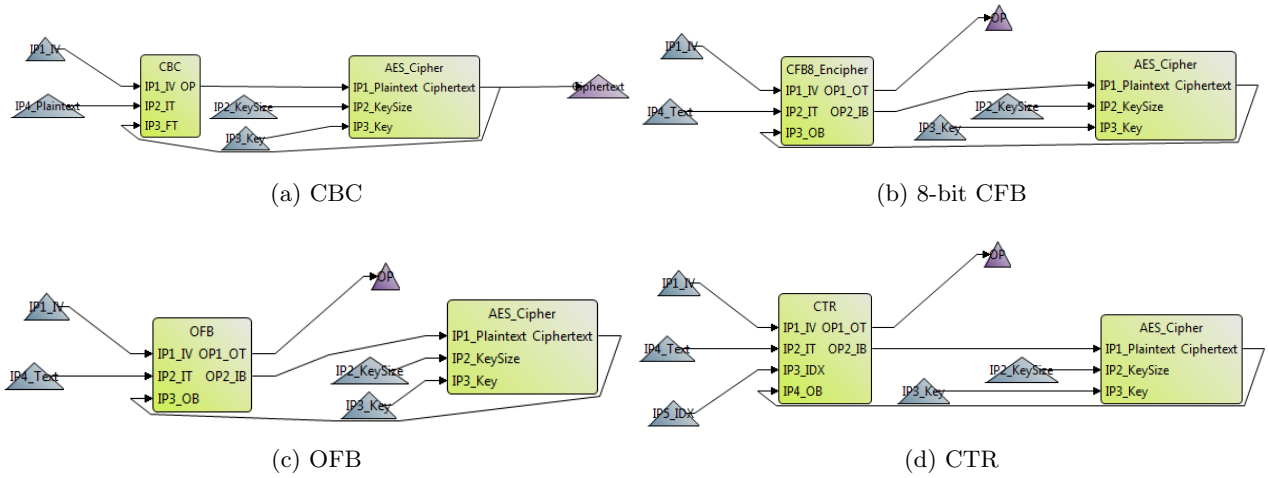
(a) AES Encipher (b) AES Decipher

Fig. 7: AES encipher and decipher in CTL.

To enhance the parallelism of the AES encipher and decipher, we transmit a token representing the round index together each plaintext/ciphertext block. This helps in parallel processing multiple blocks. Hence, each data block in AES consists of 17 tokens (one round number + 16 data tokens).

The AES encipher and decipher shown above are running in the simplest Electronic Code Book (ECB) mode. Since block ciphers running in ECB mode have the potential risks of known/chosen plaintext attack and chosen-ciphertext attack, block ciphers are often run in other modes involving feedback of ciphertext and/or use of counters. Each mode of operation is implemented as a single RVC-CAL FU, which can be connected with the AES network running in ECB mode to make it work under the expected mode of operation.

Figure 8 shows the AES encipher running at four other modes of operation where `AES_Cipher` FU in each sub-figure encapsulates AES ECB encipher of Fig. 7a. It should be noted that, changing the mode from ECB to another required target mode is just a matter of connecting the target mode's FU with the basic FU network of AES.



(a) CBC (b) 8-bit CFB

(c) OFB (d) CTR

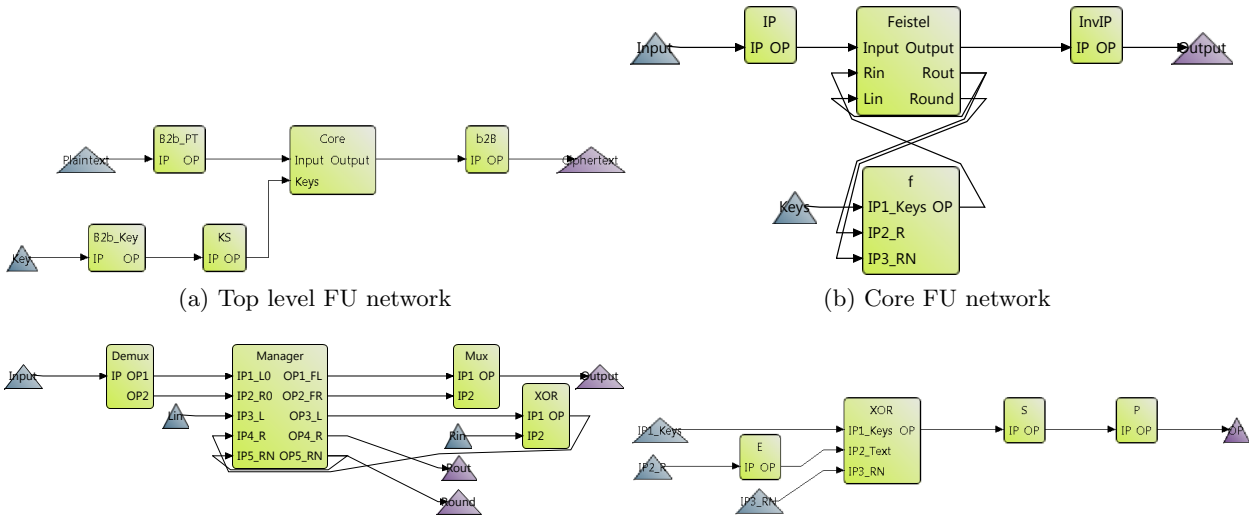
Fig. 8: AES encipher running at different modes of operation.

Look-up-tables based Optimized Implementation The main objective for developing such implementation was to evaluate the run-time performance of AES when it is implemented in a way similar to an optimized sequential program. This optimized AES implementation follows the look-up-tables based optimization algorithm used in the Rijndael's optimized reference implementation [62]. We implemented `CTL-LUT` encipher and decipher as single FU each. Both FUs contain three actions: 1) `readKeyInfo` – receives the key size; 2) `keyExpansion` – receives the key and performs the key scheduling/expansion; 3) `encrypt/decrypt` – receives the plaintext/ciphertext, performs encryption/decryption operation and produces ciphertext/plaintext.

Similar to AES standard implementation, both encipher and decipher FUs of this optimized AES implementation can also be used with any operational mode FU without any problem.

A.2 DES

Different from AES, DES is more bit-oriented. The current DES implementation in the CTL is based on bit tokens. Figure 9a shows the top level FU network of the DES encipher in the CTL, where the two `B2b` FUs (instances of `Any2Bits`) are used to convert byte tokens to bit tokens and the `b2B` FU (instance of `Bits2Any`) is used to convert bit tokens into byte tokens. The `KS` FU is key scheduler generating round keys. DES is a Feistel cipher that has an identical structure for the encipher and the decipher (except the key scheduler), so the top level network of the DES decipher is the same as the DES encipher except that `KS` FU is reconfigured to send keys in a reversed order.



(a) Top level FU network

(b) Core FU network

(c) Feistel network

(d) `f` FU network

Fig. 9: DES encipher implementation in CTL.

The `Core` FU network in Fig. 9b is composed of an initial permutation (`IP`) FU, a Feistel cipher network, an inverse permutation (`InvIP`) FU and the round function (`f`) FU, as shown in Fig. 9b. The Feistel network is shown in Fig. 9c, where the `Manager` FU controls the dataflow inside the Feistel network in different rounds and the `f` FU network implements the core round function, which is shown in Fig. 9d. As shown in Fig. 10, the `S` FU in the `f` FU network is further composed of a demuxer FU, eight parallel 6×4-bit S-boxes and a muxer FU.

A.3 Blowfish

Just like DES, Blowfish [64] is also a Feistel cipher but with a different round function `f`. Figure 11 shows the top level FU network of the Blowfish encipher in the CTL. In this implementation of Blowfish, we have *reused* the Feistel network of Fig. 9c, which we previously used to implement the DES block cipher. The processes of building of Blowfish sub-keys and S-boxes have been grouped with the implementation of the round operation in `f` FU. After 16 iterations through the Feistel network, the data block streams through the `Final_XOR` FU, which XORs last two sub-keys with the data block to generate the ciphertext.

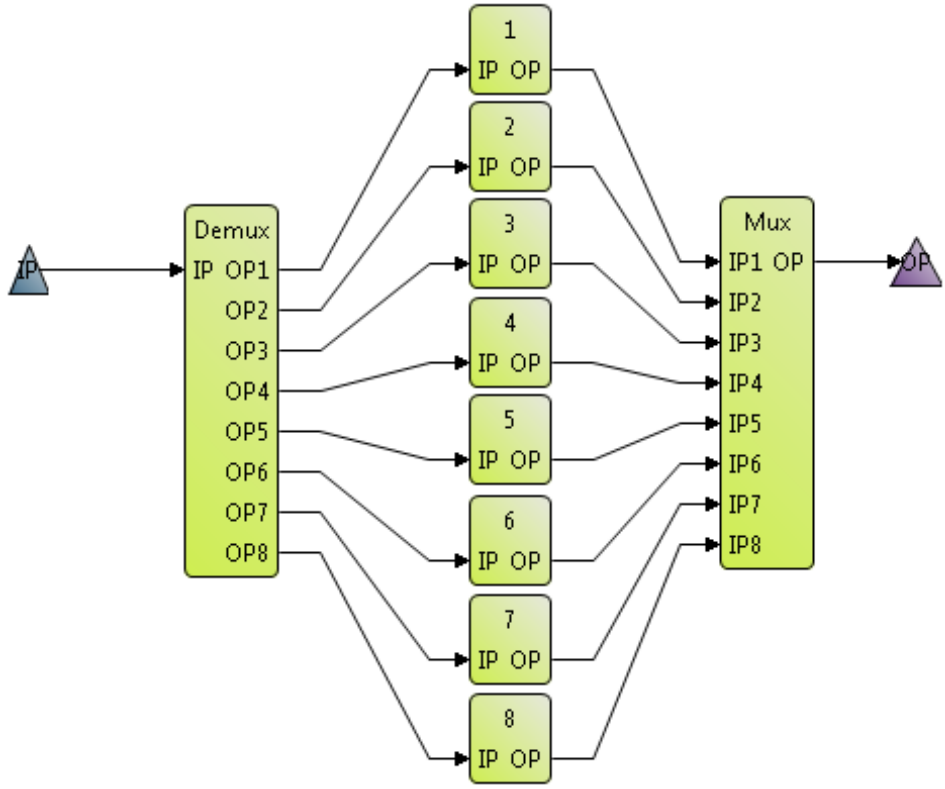


Fig. 10: The eight S-boxes in the core `f` FU network of DES encipher and decipher in CTL.

Similar to our DES implementation, Blowfish encipher and decipher also have an identical structure. An instance parameter is used to reconfigure `f` FU to use the sub-keys in either sequential order (for encipher) or reversed order (for decipher).

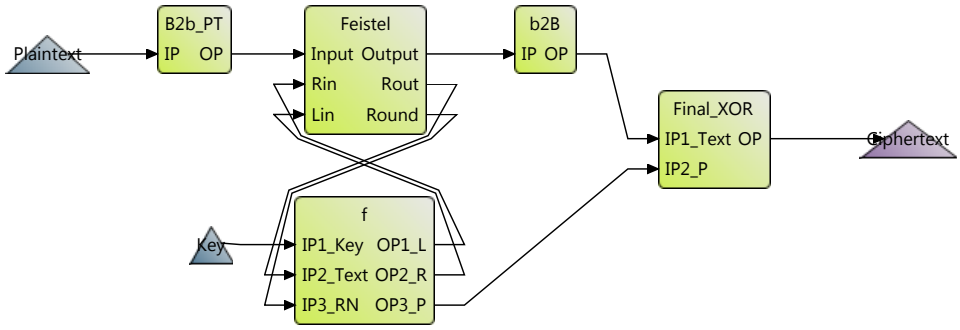


Fig. 11: Blowfish encipher implementation in CTL.