Universidade do Minho
Escola de Engenharia

André Barbosa, PG50216
Carlos Soares, PG50280

Desenvolvimento de um mecanismo de segurança que garante confidencialidade e integridade de dados sensíveis em hardware reconfigurável

Mestrado em Engenharia Eletrónica Industrial e Computadores

Projeto Integrador em Eletrónica Industrial e Computadores

Orientadores:
Sandro Pinto
Sérgio Pereira

Março 6, 2023

Conteúdo

Siglas e Acrónimos

FPGA *Field Programmable Gate Array*

SoC *System on Chip*

CPU *Central Processing Unit*

GPU *Graphics Processing Unit*

AES *Advanced Encryption Standard*

NIST *National Institute of Standards and Technology*

SPN *Substitution-permutation network*

AES *Advanced Encryption Standard*

DES *Data Encryption Standard*

TCO *Total Cost of Ownership*

IP *Intellectual Property*

SRAM *Static random-access memory*

NVM *Non-volatile memory*

1 Introdução

Segurança torna-se cada vez mais fundamental para o *design* de sistemas da *Cloud*, *IoT* e redes sociais [1]. Uma das técnicas mais bem estabelecidas para garantir a integridade e confidencialidade de dados é a utilização de algoritmos criptográficos. À medida que a proliferação de dados e a necessidade de comunicações seguras aumentam, os algoritmos de encriptação estão a tornar-se uma parte indispensável da segurança moderna da informação. São utilizados numa vasta gama de aplicações, desde a segurança de canais de comunicação [2], transações online [3] e proteção de palavra-passes. Algumas destas aplicações recorrem a algoritmos de encriptação simétrica, como é bastante frequente em ambientes de segurança de *Cloud* [4]. Outras recorrem a algoritmos de encriptação assimétrica, como em encriptação de base de dados [5].

Uma das alternativas para implementação de algoritmos de encriptação em sistemas computacionais é a utilização de *Field Programmable Gate Array* (FPGA). FPGAs tornaram-se muito populares nas últimas décadas e têm sido aplicadas em vários ramos tecnológicos. São bastante utilizadas em ambientes industriais, como por exemplo em controlo industrial [6]. Outras são frequentemente utilizadas em telecomunicações [7], em sistemas de pagamentos financeiros [8] e até na saúde [9]. O *feedback* recebido por consumidores de FPGAs durante anos, resultou no nascimento de uma nova ideia de dispositivo, denominado por *FPGA SoC* [10]. A utilização destas plataformas combina os benefícios da integração de microprocessadores com a tecnologia FPGA. A utilização de algoritmos criptográficos já não é de todo novidade [11], [12], no entanto o propósito deste trabalho tende a promover a segurança destes sistemas heterogéneos ao implementar algoritmos criptográficos simétricos em hardware reconfigurável. De seguida, encontram-se representados os objetivos a alcançar neste projeto:

- Garantir a confidencialidade de dados sensíveis: a implementação deve garantir que os dados sensíveis sejam protegidos contra acesso não autorizado e permaneçam confidenciais.
- Melhorar o desempenho do algoritmo de encriptação: a implementação deve ser capaz de executar o algoritmo de forma eficiente, de modo a reduzir o tempo de processamento necessário para criptografar ou descriptografar dados.
- Adaptar o algoritmo de encriptação para hardware reconfigurável: a implementação deve ser capaz de utilizar as características de hardware reconfigurável para aumentar a eficiência e segurança do algoritmo.

- Validar a segurança da solução proposta: a implementação deve ser testada e validada para garantir que ela seja segura e não possua vulnerabilidades conhecidas que possam ser exploradas por atacantes.

Neste projeto nós apresentamos e discutimos um design de um mecanismo de segurança que confere confidencialidade e integridade de dados sensíveis em dispositivos híbridos. A principal ideia é a proteção de dados sensíveis através da implementação eficiente e segura de um algoritmo de encriptação simétrica na lógica programável de uma FPGA SoC. Nós apresentamos uma implementação de prova de conceito com alvo à plataforma Zybo Z7 e especificamos de que forma será desenvolvida uma interface agnóstica ao tipo de implementação do algoritmo. Para demonstrar a exequibilidade da nossa abordagem em aplicações do mundo real, nós vamos correr um algoritmo de encriptação Twofish que é open-source.

2 Background

2.1 Encriptação

A encriptação é uma técnica fundamental no campo da segurança de informação. Essencialmente permite proteger dados e informações de ataques internos ou externos. Como resultado, assegura a integridade, confidencialidade, não repúdio, e validade dos dados secretos. Esta é construída sobre dois termos primários: texto simples, mensagem original, e o texto encriptado, forma encriptada da mensagem. Finalmente, a encriptação é descodificada para revelar a mensagem original. A encriptação está amplamente dividida em duas categorias. encriptação de chave simétrica e encriptação de chave assimétrica [13].

2.1.1 Encriptação de Chave Simétrica

No contexto da encriptação de chave simétrica, são consideradas duas entidades. O transmissor que é responsável por encriptar a mensagem a ser enviada e o recetor que é responsável por desencriptá-la para poder lê-la. Ambos partilham uma chave comum para encriptação e desencriptação na encriptação de chave simétrica, também conhecida como chave secreta ou encriptação de chave partilhada. Esta chave é auto certificada, o que significa que deve ser comunicada através de comunicação segura para evitar compromissos. Caso contrário, se a chave for comprometida, um atacante pode decifrar rapidamente a mensagem encriptada.

A segurança é baseada no quão difícil é saber a chave. Uma chave de 128-bits pode levar bilhões de anos até alguém conseguir adivinhar ao usar um computador comum [14]. Habitualmente, quanto maior a chave, maior a segurança envolvida. Existem dois tipos de algoritmos: (i) em bloco, a mensagem é encriptada em blocos de tamanhos específicos, (ii) em stream, a mensagem é encriptada resgatando byte a byte da informação.

Focando na encriptação em bloco, o algoritmo funciona somente para um grupo fixo de bits que é chamado de bloco. De forma a aplicar este método em mais de um bloco é necessário um modo de operação. Alguns modos necessitam de uma sequência única de bits, também conhecida como vetor de inicialização, para cada operação. Como os blocos trabalham em tamanho fixo e as mensagens podem variar de tamanho, são adicionados alguns bytes ao final para que ela fique com tamanho múltiplo desse bloco.

Entre os modos de operação mais comuns contemplam-se:

- **ECB (Electronic Codebook)**: modo mais simples de encriptação simétrica, em que cada bloco de dados é encriptado independentemente

dos outros. Embora seja fácil de implementar, o modo ECB não é muito seguro, uma vez, permite que deteção de padrões nos dados encriptados.

- **OFB (Output Feedback)**: aqui a saída do algoritmo de encriptação é realimentada para criar uma sequência pseudo-aleatória de bits, que é usada para criptografar o bloco de dados. O OFB é bastante rápido e seguro, mas pode ser vulnerável a ataques de colisão.
- **CTR (Counter)**: neste modo, um contador é usado para gerar uma sequência pseudo-aleatória de bits que é usada para criptografar cada bloco de dados. O CTR é rápido e seguro, e é amplamente utilizado em sistemas de segurança de rede e de encriptação de disco.
- **CBC (Cipher Block Chaining)**: cada bloco de dados é encriptado com base no bloco anterior. Isso faz com que os dados sejam menos suscetíveis a ataques que detetam padrões. No entanto, o CBC pode ser vulnerável a ataques de manipulação de blocos.
- **CFB (Cipher Feedback)**: a saída do algoritmo de encriptação é realimentada para o próximo bloco, em vez de usar o bloco anterior. Isso torna o CFB mais flexível do que o CBC e mais resistente a ataques de manipulação de blocos. No entanto, pode ser mais lento do que o CBC.

Twofish. O Twofish [15], [16], [17], [18] é um algoritmo de encriptação de chave simétrica, em que o tamanho do bloco de cifra é de 128 bits e o tamanho da chave pode chegar até aos 256 bits. É open source, não patenteado e de livre utilização, o que o confere um excelente alvo para iniciação na área de encriptação. O algoritmo consiste em 16 rondas construídas de forma semelhante à estrutura Feistel network. Esta estrutura é uma técnica que consiste em dividir a mensagem em blocos e aplicar várias iterações de transformações a cada bloco. Cada iteração usa uma função de substituição e uma função de permutação, com base numa chave de encriptação, para produzir uma nova versão do bloco. O resultado da iteração anterior é misturado com o bloco atual em cada iteração, tornando o processo iterativo e criando um efeito de difusão e confusão na mensagem.

Casos de Uso. O algoritmo é bastante popular e destaca-se pela sua robustez e segurança. É utilizado em várias aplicações, tanto em ambientes corporativos como em ambientes pessoais. Alguns dos casos bem conhecidos que utilizam Twofish nos seus métodos de encriptação são PGP (Pretty Good Privacy), GnuPG, TrueCrypt, e KeePass [19]. PGP utiliza o algoritmo

para encriptar e-mails, onde os dados do respetivo e-mail são encriptados, mas tanto o remetente, como o sujeito não são encriptados. GnuPG é uma implementação do OpenPGP que permite aos utilizadores encriptar e enviar dados em comunicações. Utiliza sistemas e módulos de gestão de chaves para aceder a diretórios de chaves públicas, que fornecem chaves públicas publicadas por outros utilizadores na Internet. Qualquer pessoa com acesso ao diretório de chaves públicas pode desencriptar a mensagem se enviar uma mensagem encriptada com a sua chave privada. O TrueCrypt encripta dados em dispositivos, com métodos de encriptação que são transparentes para o utilizador. Funciona localmente no computador do utilizador e codifica automaticamente os dados quando sai do computador local. Por exemplo, um utilizador que envia um ficheiro do seu computador local para uma base de dados externa teria o ficheiro encriptado quando sai do computador local. Por último, KeePass é um software de gestão de senhas que encripta as senhas que são armazenadas e cria senhas usando o Twofish.

Twofish versus Standard. Tanto Twofish como *Advanced Encryption Standard* (AES) são duas cifras de bloco de chaves simétricas amplamente utilizadas que são normalmente comparadas em termos das suas características de segurança, desempenho e implementação. AES foi estabelecida como o algoritmo de encriptação padrão pelo *National Institute of Standards and Technology* (NIST) em 2001. O AES opera em blocos de dados de 128 bits e suporta comprimentos de chave de 128, 192, ou 256 bits, semelhantes ao Twofish.

Uma das principais diferenças entre Twofish e AES são os seus princípios de conceção. Twofish utiliza uma complexa S-Box dependente de chaves e um processo flexível de expansão de chaves que gera um grande número de sub-chaves a partir da chave original, o que proporciona maior segurança contra ataques conhecidos. O AES, por outro lado, utiliza uma estrutura *Substitution-permutation network* (SPN) mais simples com S-Boxes fixas e um processo de expansão de chave padrão. Em termos de desempenho, a AES é conhecida pela sua eficiência e tem sido amplamente otimizada para várias plataformas, incluindo aceleradores de hardware, o que a torna altamente adequada para muitas aplicações com requisitos de desempenho rigorosos. Twofish, por outro lado, podem não ser tão amplamente otimizados para diferentes plataformas, o que poderia ter impacto no seu desempenho em certos casos de utilização.

Vale a pena notar que a AES é amplamente apoiada, otimizada, e utilizada em várias aplicações e plataformas, o que a torna uma escolha sólida para muitos casos de utilização. No entanto, Twofish pode ser uma alterna-

tiva viável para aplicações que requerem características de segurança adicionais, flexibilidade no tamanho da chave, e um desempenho potencialmente mais rápido em sistemas com mais memória RAM [20].

2.1.2 Ataques

Existem muitos exemplos de ataques e *bugs* que podem ocorrer quando não se utiliza encriptação simétrica. A utilização deste tipo de encriptação pode auxiliar a mitigar os ataques que têm por objetivo intercetar e ler informações sensíveis através do processo de encriptação. Um dos ataques mitigados é *Eavesdropping*. Este ataque refere-se à interceção da comunicação entre duas partes sem o seu conhecimento ou consentimento. Com encriptação simétrica, mesmo que um invasor seja capaz de intercetar a comunicação, não será capaz de compreender o conteúdo da mensagem porque a mensagem foi encriptada. Outro ataque está relacionado com a manipulação de dados, conhecido como o *data tampering attack*. Essencialmente, envolvem a adulteração de dados e podem constituir uma ameaça ainda mais grave do que o roubo ou resgate de dados em alguns casos. Neste caso mesmo que um atacante tente adulterar dados encriptados não o poderá fazer sem a chave de encriptação.

Adicionalmente, um tipo de ataque também mitigado pela encriptação simétrica, são os ataques a palavras-passe. O invasor tenta adivinhar a palavra-passe de um utilizador a fim de obter acesso à sua conta. A encriptação simétrica ajuda a mitigar estes ataques através da encriptação de palavras-passe antes de serem armazenadas num servidor. Um *website* pode utilizar encriptação simétrica para proteger as palavras-passe de utilizadores armazenadas numa base de dados. Mesmo que um atacante tenha acesso à base de dados, não será capaz de ler as senhas sem a chave de encriptação [21].

2.2 FPGA SoC

Este dispositivo combina três formas de programabilidade para máxima costumização: (i) hardware; (ii) software; (iii) programabilidade I/O. De forma a ser possível fazer a migração do algoritmo de software para hardware é necessário a utilização de hardware reconfigurável, mais propriamente, uma FPGA [22].

Primeiramente, uma FPGA não é nada mais que um circuito integrado que pode ser programado e reprogramado para executar qualquer função lógica, permitindo a implementação de sistemas digitais personalizados em hardware. Elas consistem num conjunto de blocos lógicos (também conhecidos como "células lógicas") que podem ser conectados para formar circuitos complexos. Esta flexibilidade permite que as FPGA sejam usadas numa am-

pla variedade de aplicações, incluindo processamento de sinal, controle de dispositivos e encriptação.

Por sua vez, as FPGA SoC, combinam as capacidades das FPGA com um processador de sistema integrado (como um processador ARM) num único dispositivo. Isto permite que a FPGA seja usada para acelerar tarefas específicas, enquanto o processador de sistema é usado para controlar a execução geral do sistema. O uso de uma FPGA SoC é comum em aplicações que exigem alto desempenho, como processamento de imagem, redes de comunicação e sistemas de controlo. Esta combinação entre uma FPGA e um núcleo de microprocessador num único chip permite uma comunicação mais eficiente entre os dois componentes, o que pode levar a um melhor desempenho e a uma redução do consumo de energia em comparação com a utilização de componentes separados. Além disso, as FPGA SoC oferecem a vantagem de permitir que o hardware seja reprogramado para se adaptar a diferentes necessidades de aplicação, sem a necessidade de trocar todo o hardware. Isto significa que estes dispositivos possam ser usados numa ampla variedade de aplicações, desde prototipagem rápida até sistemas em produção em massa.

2.2.1 Benefícios da sua Utilização

Pode ser claramente afirmado que as FPGAs oferecem muitas vantagens. Fundamentalmente, as FPGAs SoC são dispositivos extremamente flexíveis que permitem criar hardware customizado para sistemas embebidos com alto desempenho, alta largura de banda, e baixa latência determinística [22]. A possibilidade de construir exatamente o hardware o que se necessita e a capacidade de personalizar a FPGA significa que muitas vezes que, é possível fazer operações de uma forma mais simples, mais rápida e mais eficiente em termos energéticos [23].

A escolha de uma FPGA para um sistema oferece ao projetista maior configurabilidade, bem como menor risco de impacto no calendário de desenvolvimento porque pequenas partes de FPGAs podem ser modificadas sem impacto no resto do desenho. Assim o seu uso reduziria o tempo de concepção e como o risco de erros. Para a maioria das aplicações, o consumo de energia da *FPGA* será aceitável para as suas necessidades, e devido ao seu menor *Total Cost of Ownership* (TCO) e maior flexibilidade, as FPGAs são frequentemente a melhor escolha tecnológica.

Uma das grandes vantagens está, também, no que toca à capacidade de realizar uma operação complexa numa série de instruções que funcionam em simultâneo. É possível programar a FPGA para fazer bastantes operações diferentes em simultâneo (em paralelo). Se tiver uma matriz de 128 elementos, pode se construir 128 "pipelines"aritméticos para que todas estas operações

possam ser executadas simultaneamente, dando-lhe enormes ganhos em desempenho e utilização de energia [23].

2.2.2 Aceleração de Algoritmos de encriptação Simétrica em FPGA SoC

Os algoritmos de encriptação recorrem a diversos cálculos matemáticos e transformações para ofuscar informação. Estes cálculos resultam em grandes esforços computacionais que podem afetar o desempenho da encriptação e desencriptação da informação. O desempenho é um dos fatores cruciais na utilização de algoritmos de encriptação em áreas aplicacionais. Como resultado, surge a aplicação de algoritmos de algoritmos de encriptação em dispositivos de hardware que permitem a aceleração deste algoritmo.

A escolha de lógica reconfigurável como uma plataforma alvo para implementação de algoritmos criptográficos aparenta ser uma solução prática para sistemas embebidos e aplicações *high-speed*. Podem ser destacadas algumas vantagens da sua aplicação em sistemas híbridos.

Carregamentos de algoritmos. Os dispositivos de reconfiguração de hardware podem ser atualizados com novos algoritmos de encriptação. O novo carregamento de algoritmos pode ser necessário caso sejam detetadas fragilidades no algoritmo anterior (e.g, *Data Encryption Standard* (DES)),ou o algoritmo standard expire e seja criado um novo standard (e.g, AES). Assim, dispositivos de encriptação equipados com FPGA são capazes de atualizar uma nova configuração de código sempre que necessário.

Eficiência da arquitetura. Em certos casos, uma arquitetura hardware consegue ser mais eficiente se desenhada com base num conjunto de parâmetros específicos à aplicação que se pretende implementar. Com FPGA é possível desenhar e otimizar uma arquitetura específica a um conjunto de parâmetros.

Throughput. *General-purpose CPUs* não estão otimizados para rápidas execuções. Implementações em FPGA tem o potencial de correr consideravelmente mais rápido que implementações realizadas em software. Por exemplo, a cifra de bloco AES alcança um *throughput* de 243 Mbits/s e 12 Gbit/s num 450 MHz Pentium II e numa Virtex XCV-100BG560-6 com 12,600 slices e 80 RAMs, respetivamente [24], [25].

Eficiência de custos. A FPGA apresenta um custo de desenvolvimento bastante inferior ao da tecnologia ASIC, uma vez que podem o chip reconfigurado pode ser testado inúmeras vezes sem qualquer tipo de custo. Isto

resulta num período de *time-to-market* mais curto, que é atualmente um fator importante de custo.

2.2.3 Problemas de Segurança das FPGAs SoC

A utilização de FPGAs SoC para aplicações criptográficas é altamente apelativo por diversas razões mas, ao mesmo tempo, há muitas questões em aberto relacionadas com a segurança geral dos FPGAs SoC.

Ataques Cloning. Visto que FPGAs são genéricas, um *bitstream* feito para um dispositivo pode ser usado em qualquer outro da mesma família e tamanho. Como tal, os atacantes podem e fazem clones de bitstreams, gravando-os na transmissão para a FPGA e usando-os em outros sistemas ou produtos. Como a clonagem não requer mais do que um analisador lógico e um técnico competente, é considerada a vulnerabilidade de segurança mais comum em FPGAs voláteis. O atacante, não necessita de conhecer os detalhes do design e considera-o como uma *black-box*. Assim o bitstream de configuração da FPGA é obtido por *eavesdropping* ou a partir da *Static random-access memory* (SRAM) volátil e usado para configurar FPGA *chips* [26].

Engenharia Reversa de ficheiros bitstream. FPGA *bitstreams* são ficheiros binários que estão vulneráveis a engenharia reversa. O atacante pode extrair informação de alto nível de funcionalidade (IPs) através da engenharia reversa de *bitstreams*. São necessários dois passos para realizar engenharia reversa em ficheiros *bitstream*: (i)obter ficheiros *bitstream* desencriptados; (ii)construir as relações mapeadas entre o ficheiro *bitstream* e a *gate-level netlist*. Para obtenção do ficheiro *bitstream* o atacante pode recorrer a diversos ataques, por isso não será tido em conta este passo nesta secção.

Para alcançar a *gate-level list* através do *bitstream* recorrendo a engenharia reversa, primeiramente é necessário saber a estrutura do ficheiro normalmente fornecida pelos fornecedores de FPGAs. Depois de identificar a sua estrutura, o atacante consegue extrair relações entre pontos de configuração e pontos de interconexão programável. Vários esforços têm sido realizados para encontrar soluções para reverter parcialmente ou totalmente a *bitstream* [27], [28], [29]. A engenharia reversa parcial do *bitstream* pode ser definido como a extração de segmentos de dados do bitstream, tais como chaves, conteúdo de BRAM/LUT, ou estados de célula de memória, sem reproduzir toda a funcionalidade. Assim, as chaves escondidas no *bitstream* também ficariam comprometidas, e se o atacante pretende determinar que algoritmo criptográfico é utilizado, a engenharia reversa parcial pode ser útil.

Um dos ataques comuns que requerem engenharia reversa são os ataques *tampering*. Aqui o atacante consegue modificar o design de uma aplicação. O *tampering* pode ser empregado para acrescentar lógica que permita fugas de informação de uma aplicação ou para desabilitar partes da aplicação, potencialmente derrotando outras medidas de segurança. Primeiramente, o *tampering* deve controlar a aplicação para definir valores no *bitstream*, por isso engenharia inversa também também pode ser necessária. No entanto, mais tarde, apenas poucas as partes do *bitstream* que se limitam a codificar podem ser suficiente [30].

Por último outro ataque relevante que também pode necessitar de componentes derivados da engenharia reversa é o *spoofing*. O atacante substitui o *bitstream* da FPGA com a sua próprio *bitstream*. Como resultado, o sistema pode tornar-se vulnerável, dando aos atacantes o controlo efetivo da máquina ou sistema.

Ataques Readback. *Readback* é uma característica fornecida para a maioria das famílias FPGA. Esta característica permite ler uma configuração fora da FPGA para facilitar o *debugging*. A ideia do ataque é ler a configuração da FPGA através do JTAG ou interface de programação a fim de obter informações secretas (por exemplo, chaves). Após a solicitação, a FPGA envia a imagem que inclui a configuração, as *look-up tables* e o conteúdo da memória para o PC *host* ou outro dispositivo, por meio da porta de configuração. Esta imagem é diferente do *bitstream* original por não incluir o cabeçalho, o rodapé e os comandos de inicialização. Os dados dinâmicos em LUTs e BRAMs também são diferentes de seu estado inicializado.

Para além de ser uma ferramenta poderosa de caracterização na verificação e teste de produção do FPGA pelos fabricantes, o *readback* também permite que o desenvolvedor do sistema verifique a correção do design enquanto ele está operando no próprio FPGA, no entanto, se ativado, um atacante pode fazer o *readback* do design, adicionar o cabeçalho estático e o rodapé em falta e utilizá-lo noutro dispositivo, reprogramar a FPGA com uma versão modificada ou engenharia reversa.

Ataques Replay. O ataques de *replay* da FPGA, tornou-se uma séria preocupação de segurança e privacidade para o desenho da FPGA, uma vez que aqui o invasor faz o *downgrade* do sistema baseado na FPGA para a versão anterior com vulnerabilidades conhecidas e explora tais vulnerabilidades. Os atuais mecanismos de proteção do *Intellectual Property* (IP) da FPGA que visam a proteção dos *bitstreams* de configuração FPGA através de *watermarking* ou encriptação, não conseguem impedir a repetição de ataques. O

invasor pode modificar o *bitstream* para incluir seu próprio código malicioso ou remover partes do código original, permitindo que ele execute operações não autorizadas na FPGA ou acesse informações confidenciais.

Ataques Side-Channel. Os ataques de *side-channel* dependem de manifestações mensuráveis de processos internos de dispositivo-externo para deduzir dados secretos ou modos de operação, explorando a implementação e não a construção algorítmica [31]. Essencialmente, explora-se a fuga de informação física e indesejada onde os métodos para invadir a FPGA estão precisamente em virar os padrões informativos do sistema contra ele próprio. Três grandes tipos de ataques de *side-channel* e a sua relevância para as FPGAs são, ataques de análise do consumo de energia, análise de emanação electromagnética e análise do timing de operações de processamento de dados. Quando um ficheiro *bitstream* é encriptado, o que é suportado pela maioria dos vendedores FPGA, os ataques de *side-channel* podem permitir a fuga de chaves mantidas nos chips da FPGA e tornar o *bitstream* desprotegido, figura 1.

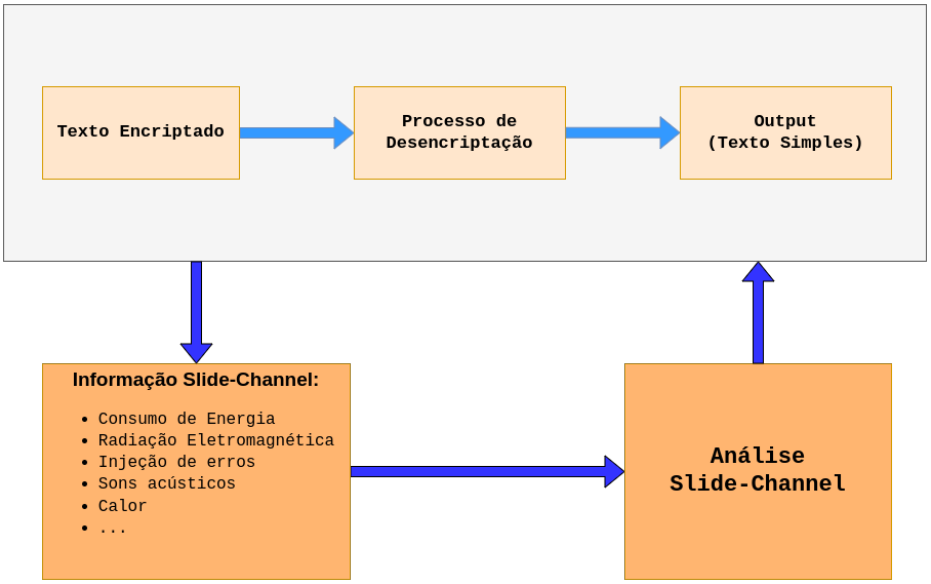

Figura 1: Como Ataques de *Side-Channel* viram os padrões informativos do sistema contra ele próprio.

A injeção de erros é o ataque de *Side-channel* comum. Os ataques *glitch* envolvem a injeção de pequenas falhas no sistema para causar erros ou manipular o comportamento do dispositivo. Os principais tipos de *glitches* são

a tensão, o *clock*, electromagnéticos e ópticos. As falhas de tensão são executadas através da queda momentânea das tensões de alimentação durante a execução de operações específicas. A falha do *clock* é executada alterando a temporização do *clock* para violar os requisitos de configuração e de tempo de espera do hardware, tal como através da inserção de impulsos de avaria do *clock* entre impulsos de *clock* normalmente cronometrados. A injecção de falha electromagnética é executada através da geração de um impulso electromagnético de alta intensidade de curta duração localizado que induz correntes no circuito interno do chip. E finalmente, a injecção de falha óptica utiliza um laser infravermelho, e normalmente requer a de capsulação do chip para expor a matriz de silício [32].

2.2.4 Métodos de proteção

Discutido as vulnerabilidades e ameaças às quais as FPGAs são propensas, agora são considerados métodos de defesa e proteção. Muitas FPGAs não têm *Non-volatile memory* (NVM), portanto os *bitstreams* são armazenados numa NVM externa e carregados durante a inicialização [30]. No entanto, a NVM externa pode ser escrita ou lida por outros componentes num sistema (por exemplo, processadores de aplicações) ou por adversários físicos, ou seja, os *bitstreams* devem ser encriptados para prevenir o roubo de propriedade intelectual e clonagem, e autenticados para evitar *tampering* [30] e inserção de Trojan.

Encriptação do Bitstream. Encriptar a *bitstream* no final do fluxo de design e desencriptá-la dentro da FPGA defende-se contra ataques de clonagem, engenharia reversa e, em alguns casos, fornece proteção limitada contra *tampering*. No final do fluxo de design, a chave de encriptação é solicitada ao utilizador e a carga de configuração do *bitstream* é encriptada. O utilizador, então, carrega a mesma chave na FPGA, que tem um deencriptador dedicado na sua unidade de configuração. O cabeçalho do *bitstream* contém informação que instruí a FPGA a passar os dados através do deencriptador antes que estes sejam enviados para as células de memória de configuração. Um atacante que obtenha a *bitstream* encriptada não pode usá-la porque não tem a chave correta. Desta forma, ele não pode reverter a encriptação ou usá-la noutro dispositivo (assumindo que chaves diferentes são carregadas em cada FPGA). Portanto, a encriptação do *bitstream* pode ajudar a proteger contra ataques que visam obter ou clonar a informação do *bitstream*, assim como pode prevenir contra engenharia reversa, ou até mesmo proteger contra ataques limitados de *tampering*.

3 Objetivos do Design e Threat Model

O nosso principal objetivo é promover a segurança dos sistemas heterogéneos ao implementar um algoritmo criptográfico simétrico de alta segurança recorrendo à utilização de hardware reconfigurável. Desta forma pode-se traçar os seguintes sub-objetivos:

1. Manter a portabilidade da implementação software do algoritmo de encriptação.
2. Desenvolver uma interface agnóstica ao algoritmo.
3. Oferecer um bom *trade-off* segurança/desempenho.
4. Proteger elementos críticos de segurança.

3.1 Threat Model

Neste threat model, temos a situação em que o utilizador pretende que a SoC possua um sistema operativo. Isto pode ser assegurado fazendo o boot através do SD Card, que contém a imagem do Sistema Operativo. Para além da partição de Boot, o cartão SD possui uma partição dedicada ao *Root Filesystem*. O *Root Filesystem* contém informação sensível à integridade do sistema e deste modo é necessário garantir que em caso de perda ou roubo do cartão SD os dados possam estar protegidos. Desta forma, desenvolveu-se o sistema representado na figura 2.

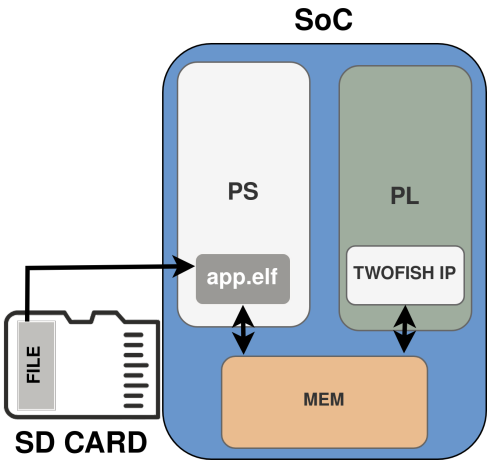

Figura 2: Sistema geral.

Como se demonstra na figura, o cartão SD contém um ou mais ficheiros, e através de um programa que corre no PS da placa, o utilizador pode especificar qual o ficheiro que pretende encriptar. Depois de determinar o ficheiro a encriptar, o programa recorre a uma interface AXI para estabelecer comunicação com o Twofish IP e efetuar a encriptação/desencriptação dos blocos de dados do ficheiro. Essa comunicação ocorre recorrendo à escrita e leitura de registos em endereços de memória conhecidos.

Como se pode observar na figura 3, o programa irá escrever na memória o valor da chave secreta e do vetor inicial para efetuar a encriptação.

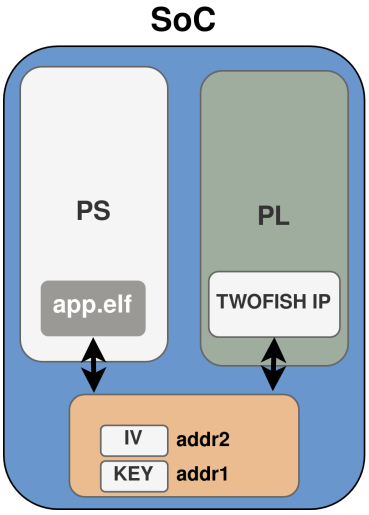

Figura 3: Dados na memória do sistema.

Isto cria vulnerabilidades no sistema uma vez que o atacante pode aceder à informação armazenada na região da memória e obter a chave secreta para proceder à desencriptação do conteúdo no cartão SD. Contudo, não pretendemos assegurar a proteção dos dados na memória, e assim surge como *assumption* de segurança do nosso projeto que os dados da memória não podem ser acedidos pelo atacante, bem como todos os dados do PS da SoC (nomeadamente o *bitstream*). A nossa solução tem exclusivamente como objetivo garantir a proteção dos dados armazenados na partição do *Root Filesystem* do cartão SD.

4 Design

O Twofish IP será obtido através de um *third party*. É requerido que este IP apresente uma interface que permita ao utilizador introduzir a chave secreta, introduzir uma mensagem fixa de 128 bits, e realizar a encriptação ou desencriptação dessa mensagem no modo CBC. A interface pretendida com o IP terá de ser semelhante à demonstrada na figura 4.

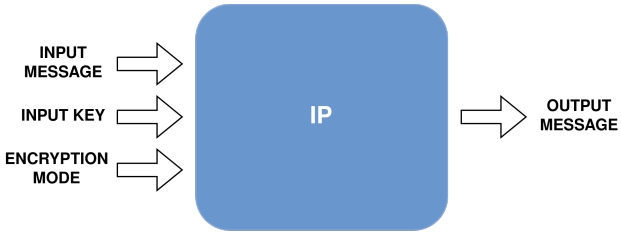

Figura 4: Interface do Twofish IP.

Para estabelecer a comunicação entre o IP de encriptação/desencriptação e o utilizador será estabelecida uma interface *AXI-Lite* entre o *Processing System* e o respetivo IP. Assim, a interação entre o PS e o IP do Twofish será mediada pela leitura e escrita de registos. Quando o PS pretende enviar informações ao IP irá proceder à escrita na memória no endereço associado ao respetivo porto que deseja modificar, e por outro lado, o IP irá efetuar a leitura do mesmo registo, como se demonstra na figura 5.

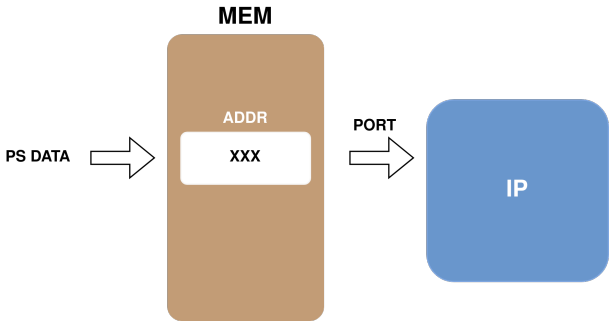

Figura 5: Interface AXI entre o IP e o PS.

A configuração do IP será feita através de um programa de software que irá correr sobre o sistema operativo Linux da placa. Isto é, será desenvolvido um programa que será executado pelo Linux, e irá modificar o modo de encriptação do IP consoante a decisão do utilizador.

De modo a auxiliar o utilizador, será ainda fornecida uma API que permita realizar algumas aplicações: (i)encriptação/desencriptação de uma mensagem de texto de 16 bytes; (ii)encriptação/desencriptação de um ficheiro de texto; (ii)encriptação/desencriptação de uma imagem; A interface a ser desenvolvida será agnóstica a hardware e software, ou seja, permite ao utilizador abstrair-se da plataforma alvo de execução.

Como se observa na figura 6, o utilizador irá comunicar com o Sistema Operativo através do protocolo UART.

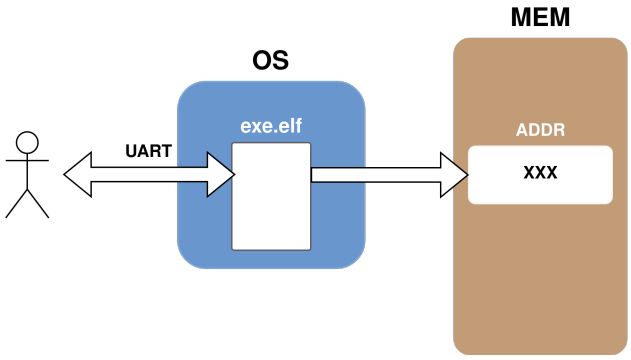

Figura 6: Comunicação entre o utilizador e o OS.

5 Implementação

5.1 Third-Party Twofish

Posteriormente a um estudo aprofundado, foi tomada a decisão de adotar uma implementação open source do algoritmo Twofish em VHDL [33] para atingir os objetivos estabelecidos. Essa escolha foi feita levando em consideração que seria desnecessário reescrever toda a lógica matemática envolvida no algoritmo a partir do zero. No entanto, ajustes substanciais precisaram de ser feitos no código original, uma vez que a implementação disponível não atendia plenamente às exigências do projeto.

Foram realizadas adaptações no código para permitir a utilização de mensagens variáveis com comprimentos de 128 bits fixos e para viabilizar a transição do modo de computação atualmente utilizado no projeto, que era o ECB (Electronic Codebook), para o modo de computação CBC (Cipher Block Chaining), geralmente considerado mais adequado para a encriptação de imagens e arquivos de texto. Adicionalmente, utilizamos apenas chaves variáveis apenas de 128 bits e a implementação final contempla tanto o processo de encriptação quanto o de desencriptação.

Inicialmente começamos por realizadar uma mudança na interface externa da implementação do algoritmo Twofish. A declaração da entidade "core"do módulo, *core.vhd*, foi modificada para incluir as novas entradas e saídas.

```
1  entity core is
2  port(
3      --inport : in std_logic_vector(127 downto 0); --input
       from cleartext entity
4      plaintext3_i : in std_logic_vector(31 downto 0);
5      plaintext2_i : in std_logic_vector(31 downto 0);
6      plaintext1_i : in std_logic_vector(31 downto 0);
7      plaintext0_i : in std_logic_vector(31 downto 0);
8      --inkey : in std_logic_vector (127 downto 0); -- input
       from keymodule entity
9      globalkey3_i : in std_logic_vector(31 downto 0);
10     globalkey2_i : in std_logic_vector(31 downto 0);
11     globalkey1_i : in std_logic_vector(31 downto 0);
12     globalkey0_i : in std_logic_vector(31 downto 0);
13     clk : in std_logic; --Clock signal
14     reset : in std_logic;
15     usr_ld_key : in std_logic; --Usr requests load key
16     usr_start : in std_logic; --Usr requests start
17     usr_encrypt : in std_logic; --Usr requests encrypt/
       decryption
18     idle : out std_logic; --Device is idle
19     -outCiphertext : out std_logic_vector(127 downto 0)); --
```

```
    output after one iteration through enc/dec
20      cryptotext3_o : out std_logic_vector(31 downto 0);
21      cryptotext2_o : out std_logic_vector(31 downto 0);
22      cryptotext1_o : out std_logic_vector(31 downto 0);
23      cryptotext0_o : out std_logic_vector(31 downto 0);
24
25      initVector3_i : in std_logic_vector(31 downto 0);
26      initVector2_i : in std_logic_vector(31 downto 0);
27      initVector1_i : in std_logic_vector(31 downto 0);
28      initVector0_i : in std_logic_vector(31 downto 0));
29  end core;
```

Listing 1: Interface Externa do Twofish.

Anteriormente, como se pode analisar na listagem 1, o algoritmo utilizava um único vetor de 128 bits para representar tanto a mensagem de entrada, *plaintext*, quanto a chave de encriptação, *inkey*, e mensagem encriptada, *outCiphertext*. No entanto, a nova abordagem opta por dividir esses dados em quatro vetores menores de 32 bits cada. Essa divisão em quatro vetores menores traz benefícios significativos para a manipulação dos dados durante a implementação do algoritmo. Ao separar em partes específicas, representadas pelos 4 diferentes vetores, torna-se mais fácil e conveniente trabalhar com diferentes partes da mensagem e chave de forma individual. Isso permite um acesso mais detalhado aos bits durante o processo de encriptação, facilitando a manipulação e a implementação do algoritmo. Além disso esta divisão é benéfica na geração do IP, uma vez que, se vai utilizar interfaces AXI4 e sendo assim estávamos restritos a usar uma largura de dados, *data width*), de 32 bits. Por fim foram adicionadas quatro novas entradas referentes ao vetor de inicialização necessário para o modo de computação CBC e foi adicionada uma nova saída, *idle* para indicar indica se o dispositivo está ocioso ou em espera.

Após estas alterações foram adicionados dois novos componentes, CBCmode e CBCmode_des, listagem 2.

```
1  component CBCmode
2  port(
3      IVvector : in std_logic_vector(127 downto 0);
4      plaintext_i : in std_logic_vector(127 downto 0);
5      cyphertext : in std_logic_vector(127 downto 0);
6      finishEnc : in std_logic;
7      clk : in std_logic;
8      usr_encrypt: in std_logic;
9      plaintext_o : out std_logic_vector(127 downto 0)
10     );
11 end component;
12
13 component CBCmode_des
```

```
14 port(
15     IVvector : in std_logic_vector(127 downto 0);
16     cypher_i : in std_logic_vector(127 downto 0);
17     cypher_text : in std_logic_vector(127 downto 0);
18     finishDec : in std_logic;
19     clk : in std_logic;
20     usr_encrypt: in std_logic;
21     delay: in std_logic;
22     plaintext_o : out std_logic_vector(127 downto 0)
23     );
24 end component;
```

Listing 2: Declaracao do Modulos CBC.

Estes são responsáveis pela implementação do modo de operação CBC. O componente *CBCmode* realiza a encriptação de um texto em claro, ou seja, um texto normal sem encriptação. Recebe um vetor de inicialização, o texto em claro e o texto encriptado gerado anteriormente, e produz como saída o texto em claro encriptado. O processo de encriptação ocorre em múltiplos blocos de 128 bits, onde cada bloco é encriptado em relação ao bloco anterior através de operações específicas do algoritmo de encriptação utilizado.

O componente opera de acordo com a respetiva solicitação de encriptação, *usr_encrypt* e o sinal de término, *finishEnc*. Quando a solicitação de encriptação é ativada, ou seja *usr_encrypt = 1*, o componente inicia o processo de encriptação utilizando o vetor de inicialização e o texto em claro fornecidos. O resultado da encriptação é gerado na saída *plaintext_o*. O sinal de término é utilizado para indicar que o processo de encriptação foi concluído.

Por outro lado, o componente *CBCmode_des* realiza a desencriptação de um texto encriptado, recebe um vetor de inicialização, o texto encriptado de entrada e o texto encriptado gerado anteriormente, e produz como saída o texto em claro desencriptado.O processo de desencriptação ocorre em múltiplos blocos de 128 bits, onde cada bloco é desencriptado em relação ao bloco anterior através de operações específicas. O componente opera de acordo com a solicitação de encriptação, *usr_encrypt*, o sinal de término, *finishDec* e o sinal de atraso, *delay*. Quando a solicitação de encriptação é, neste caso *usr_encrypt = 0* para desencriptação, o componente inicia o processo de desencriptação utilizando o vetor de inicialização e o texto encriptado fornecidos. O resultado da desencriptação é gerado na saída *plaintext_o*. O sinal de término é utilizado para indicar que o processo de desencriptação foi concluído, enquanto o sinal de atraso introduz um atraso controlado no processo de encriptação, se necessário.

Mas agora nas seguintes listagens procede-se a uma analisa em mais detalhe destes dois componentes. Primeiramente o *CBCmode*, listagem 3.

```
1  `timescale 1ns / 1ps
2
3  module CBCmode(
4      input clk,
5      input [127:0] IVvector, // vetor de inicialização
6      input [127:0] plaintext_i, // texto em claro
7      input [127:0] cyphertext, // texto encriptado anterior
8      input finishEnc, // sinaliza o término do processo de
       encriptação
9      input usr_encrypt, // 0 -> desencriptaçao | 1 -> encripta
       çao
10     output [127:0] plaintext_o
11     );
12
13     assign plaintext_o = (usr_encrypt) ? (finishEnc==0) ?
       IVvector^plaintext_i : cyphertext^plaintext_i
14                                          : plaintext_i;
15
16 endmodule
```

Listing 3: Modulo CBCmode.

Para o *assign* do resultado final, o texto em claro encriptado, utilizamos o operador ternário "?". Assim quando o sinal *usr_encrypt* está ativado, o módulo executa o processo de encriptação. Se o sinal *finishEnc* estiver desativado, o módulo realiza a operação XOR entre o vetor de inicialização e o texto em claro de entrada, uma vez que ainda não se encontrava nenhuma encriptação finalizada. Por outro lado, se o sinal *finishEnc* estiver ativado, significa que já acabou um encriptação, o módulo realiza a operação XOR entre o texto encriptado anterior e o texto em claro de entrada.

Relativamente ao módulo *CBCmode_des*, este encontra-se na seguinte listagem 4. Primeiramente foram utilizados dois registos, um para armazenar os valores do texto encriptado anterior, *last_cypher* e um para armazenar o texto encriptado anterior atrasado *last_cypher_delayed*. A operação principal do módulo ocorre em dois blocos sempre ativados na transição ascendente do sinal de relógio. Essencialmente no primeiro bloco, o texto encriptado anterior é armazenado no registo *last_cypher_delayed* quando o sinal de atraso de *delay* está ativado, o que permite que o valor seja preservado para uso posterior na desencriptação. No segundo bloco, o valor atualizado de *last_cypher_delayed* é transferido para o registo *last_cypher*. Isso ocorre novamente apenas quando o sinal de atraso de *delay* está ativado. Assim esta operação garante que o valor do texto encriptado anterior esteja sempre atualizado e disponível para a desencriptação.

Por fim a saída é atribuída novamente utilizando uma expressão condicional com o operador ternário "?". Quando o sinal de solicitação de encriptação

usr_encrypt é desativado, ou seja é zero, o módulo executa a desencriptação. Se o sinal de término de desencriptação *finishDec* também estiver desativado, não existiu nenhuma desencriptação, ocorre a operação XOR entre o vetor de inicialização e o texto encriptado atual. Caso contrário, ocorre a operação XOR entre o texto encriptado atual e o valor do texto encriptado anterior. No caso em que o sinal de solicitação de encriptação *usr_encrypt* está ativado, o módulo simplesmente repassa o texto encriptado atual como saída, sem realizar operações adicionais nem alterações à cifra.

```
1  `timescale 1ns / 1ps
2
3  module CBCmode_des(
4      input clk,
5      input [127:0] IVvector, // vetor de inicialização
6      input [127:0] cypher_text, // texto encriptado anterior
7      input [127:0] cypher_i, // texto encriptado atual
8      input finishDec, // término de desencriptação
9      input usr_encrypt, // 0 -> desencriptaçao | 1 -> encripta
       çao
10     input delay, // sinal de atraso
11     output [127:0] plaintext_o
12 );
13
14     reg [127:0] last_cypher;
15     reg [127:0] last_cypher_delayed;
16
17     initial begin
18         last_cypher = 128'h0;
19         last_cypher_delayed = 128'h0;
20     end
21
22     always @(posedge clk) begin
23         if(delay==1'b1)begin
24         last_cypher_delayed <= cypher_text; // Store previous
       value
25         end
26     end
27
28     always @(posedge clk) begin
29         if(delay==1'b1)begin
30         last_cypher <= last_cypher_delayed; // Update with
       current value
31         end
32     end
33
34     assign plaintext_o = (usr_encrypt == 0)
35         ? (finishDec == 0) ? IVvector ^ cypher_i : cypher_i ^
       last_cypher
```

```
36          : cypher_i;
37
38 endmodule
```

Listing 4: Modulo CBCmode_des.

Feita a adição dos módulos CBC para alteração do modulo de computação, restam mais algumas modificações, não menos importantes, realizados ainda no módulo core, listagem 5. Incluímos os sinais *inputP128*, *inputK128*, *output128*, *vector128*, *outCBC*, *outCBCdec*. Os sinais *inputP128*, *inputK128* e *vector128* são preenchidos com valores resultantes de concatenações de sinais individuais, criando sinais de 128 bits, necessários para o *mapping* das seguintes funções *little endian*. Pelo contrário os sinais de saída *cryptotext3_o*, *cryptotext2_o*, *cryptotext1_o* e *cryptotext0_o* recebem partes específicas do sinal final *outCBCdec*, dividindo-o em partes de 32 bits cada.

Ainda são declarados os sinais *finishEnc*, *finishDec* e *delay* que por sua vez, também são listados como portas de entrada/saída no módulo controlo. Assim ao utilizar as várias declarações como *finishEnc=>s_finishEnc*, *finishDec=>s_finishDec*, *delay=>s_delay*, estão a ser feitas conexões entre o sinais *s_finishEnc*, *s_finishDec* e *s_delay* declarados e os sinais de entrada do módulo controlo. Por exemplo a conexão permite que os valores do sinal *s_finishEnc* sejam acessíveis dentro do módulo controlo por meio do porto *finishEnc*.

Totalizando, as modificações restantes foram realizadas ao adicionar a instanciação dos modos CBC e a instanciação das funções *little-endian*. Em síntese, é contemplada a conversão de determinados sinais para o formato *little-endian* antes de encaminhá-los aos módulos responsáveis pela encriptação e desencriptação. O módulo *CBCmode_enc* requer uma entrada de texto claro, um vetor de inicialização e sinais de controlo de encriptação, e produz o texto encriptado correspondente. Por sua vez, o *CBCmode_dec* opera sobre o texto encriptado, o vetor de inicialização e sinais de controlo de desencriptação, a fim de gerar o texto claro desencriptado.

```
1 signal
2 key, input, output, outkey, inputP128, inputK128, output128,
      vector128, outCBC, outCBCdec : std_logic_vector(127 downto
      0);
3 signal
4   latch_cleartext, latch_key,
5   sel_odd, sel_k,
6   param_source_is_result,
7   latch_cipher_LS, latch_cipher_MS,
8   load_reg_evenk,
9   load_reg_oddk,
10  load_modf_reg,
```

```
11    ctrl_encrypt,
12    load_reg_pre_even, load_reg_pre_odd,
13    load_reg_post_even, load_reg_post_odd,
14    s_finishEnc,
15    s_finishDec,
16    s_delay
17    : std_logic;
18  begin
19
20  inputP128 <= plaintext3_i & plaintext2_i & plaintext1_i &
        plaintext0_i;
21  inputK128 <= globalkey3_i & globalkey2_i & globalkey1_i &
        globalkey0_i;
22  vector128 <= initVector3_i & initVector2_i & initVector1_i &
        initVector0_i;
23
24  cryptotext3_o <= outCBCdec(127 downto 96);
25  cryptotext2_o <= outCBCdec(95 downto 64);
26  cryptotext1_o <= outCBCdec(63 downto 32);
27  cryptotext0_o <= outCBCdec(31 downto 0);
28
29  --little endian convertion
30  little_endian_input: little_endian_converter
31  port map ( big_endian => outCBC,
32      little_endian => input);
33
34  little_endian_key: little_endian_converter
35  port map ( big_endian => inputK128,
36      little_endian => key);
37
38  little_endian_output: little_endian_converter
39  port map ( big_endian => output,
40      little_endian => output128);
41
42  -- Encryption
43  CBCmode_enc: CBCmode
44  port map(  IVvector => vector128,
45         plaintext_i => inputP128,
46         cyphertext => outCBCdec,
47         finishEnc => s_finishEnc,
48         clk => clk,
49         usr_encrypt => usr_encrypt,
50         plaintext_o => outCBC);
51
52  -- Desencryption
53  CBCmode_dec: CBCmode_des
54  port map(  IVvector => vector128,
55         cypher_i => output128,
56         cypher_text => inputP128,
```

```
57          finishDec => s_finishDec,
58          clk => clk,
59          usr_encrypt => usr_encrypt,
60          delay => s_delay,
61          plaintext_o => outCBCdec);
62
63
64  twofishController: control
65  port map (
66           //...
67           finishEnc=>s_finishEnc,
68           finishDec=>s_finishDec,
69           delay=>s_delay
70  );
```

Listing 5: Alteracoes no Modulo Core.

Ao analisar todas as modificações no modulo core passamos analisar algumas modificações no módulo de controlo utilizado para sincronizar as diferentes etapas de encriptação e desencriptação. Primeiramente começamos por adicionar um sinal de entrada e três sinais de saída adicionais, listagem 6. O sinal *start_i* será utilizado mais tarde como sinal de entrada no módulo oneshot. Os sinais *finishEnc* e *finishDec*, como já referido anteriormente, são usados para indicar a conclusão das operações de encriptação e desencriptação, respetivamente. Podem ser afirmados quando o processo de encriptação ou desencriptação está terminado, permitindo que outras partes do sistema tomem as ações apropriadas ou desencadeiem operações subsequentes. Já o sinal de *delay* é usado para introduzir um sincronização dentro da entidade de controlo.

```
1  entity control is
2  port(
3      //...
4      // old input and outpus
5      //...
6      start_i : in std_logic;
7      finishEnc: out std_logic;
8      finishDec: out std_logic;
9      delay: out std_logic);
10 end control;
```

Listing 6: Interface externa de Controlo.

De seguida adicionamos um novo componente, devido à necessidade de uma funcionalidade relacionada com um circuito one-shot, o módulo chamado *mainOneShot*. E ainda adicionamos um estado adicional chamado *extra_cycle*, o que implica que o controlador tem agora um novo estado na sua máquina de estados, operação extra no processo de encriptação. Es-

sencialmente a introdução dos sinais *finishAux*, *finishDecAux* e *delayAux* e a inclusão do estado *extra_cycle* sugerem modificações para tratar casos específicos e atrasos no comportamento do controlador.

```
component mainOneShot
port (i_clk : in std_logic;
  i_rst : in std_logic;
  i_signal : in std_logic;
  o_one_shot : out std_logic );
end component;

--
--type DECLARATION
--
type state_type is (my_idle, load_keyA, compute_K0,
    compute_K1, compute_K2,
         compute_K3, compute_K4, compute_K5, compute_K6,
    compute_K7, extra_cycle,
         compute_K2r_8, compute_K2r_9, compute_even,
    compute_odd);

--
--signal DECLARATION
--

signal
    finishAux: std_logic := '0'; --variable that indicates
    the finish of encryption
signal
    finishDecAux: std_logic := '0'; --variable that indicates
    the finish of decryption
signal
    delayAux: std_logic := '0';
```

Listing 7: OneShot e Declaracoes.

Passando para descrição da estrutura do controlador, num primeiro momento, módulo *mainOneShot* é instanciado com o nome *oneShotMain*, listagem 8. Os portos de entrada e saída do módulo estão ligados aos sinais correspondentes no contexto circundante. O sinal *start_i* é uma entrada para o módulo *mainOneShot*, que representa o sinal de entrada que desencadeia o comportamento one-shot. Por outro lado, o sinal de *start* é a saída do módulo *mainOneShot*, que representa o impulso one-shot resultante. Assim, quaisquer alterações no sinal de entrada *start_i* serão processadas pelo módulo *mainOneShot*, e o impulso one-shot resultante estará disponível no sinal de saída *start*. O sinal de *start* pode então ser utilizado no circuito circundante para acionar outros componentes com base no comportamento de disparo único.

```
oneShotMain: mainOneShot
port map(i_clk => clk,
      i_rst => reset,
      i_signal => start_i,
      o_one_shot => start);
```

Listing 8: mainOneShot Mapping.

Agora, no que diz respeito ao módulo *mainOneShot*, o módulo contém dois registos, *signal_dly* e *aux*, utilizados para armazenar valores intermédios. No bloco principal *always @(posedge i_clk)*, através da declaração *if-else* definimos o comportamento do circuito. Caso o reset esteja desativado, o *signal_dly* é atualizado com *i_signal* e po fim *aux* é atualizado com o valor anterior de *signal_dly*. O sinal de saída *o_one_shot* é determinado pela operação lógica AND entre *i_signal* e a negação de *aux*. Isso resulta num *o_one_shot* sendo nível lógico alto somente quando *i_signal* transita de baixo para cima e *aux* está baixo, indicando um pulso único.

```
`timescale 1ns / 1ps

module mainOneShot(
    input i_clk,                // Clock source
    input i_rst,                // Reset
    input i_signal,             // Input signal
    output o_one_shot           // One shot signal
    );

    // Register
    reg signal_dly;
    reg aux;

    // Initial conditions
    initial begin
        signal_dly <= 1'b0;
        aux <= 1'b0;
    end

    // One shot
    always @(posedge i_clk)
    begin
        if(i_rst == 1'b1)
        begin
            signal_dly <= 1'b0;
            aux <= 1'b0;

        end
        else begin
            signal_dly <= i_signal;
            aux <= signal_dly;
```

```
32          end
33      end
34
35      // Assign the output
36      assign o_one_shot = i_signal & ~aux;
37
38  endmodule
```

Listing 9: Modulo mainOneShot.

Por último, as alterações finais foram feitas na maquina de estados. Existem duas secções adicionais adicionadas. No estado *my_idle*, após verificar as condições, é incluída uma atribuição adicional *delayAux <= '0'*, o que implica que não é introduzido qualquer atraso nesta fase. Na secção para o estado *extra_cycle* existem duas atribuições: *finishAux <= '1'* e *finishDecAux <= '1'*. Portanto, quando o controlador faz a transição para o estado *extra_cycle*, ele define *finishAux* como 1 e verifica seu valor para atribuir 1 a *finishDecAux*. Além disso, ele atualiza *delayAux* para 1 na secção para o estado *my_idle*. Mais em baixo, no final do módulo são feitas as atribuições de forma a atualizar os valores dos determinados sinais, *finishEnc*, *finishDec* e *delay*, com base nos valores das variáveis correspondentes *finishAux*, *finishDecAux* e *delayAux* no código.

```
process (clk,reset)
begin
if reset ='1' then
  state <= my_idle;
elsif clk'EVENT AND clk = '1' then
  case state is
    when my_idle =>
      if ld_key = '1' then
        state <= load_keyA;
      elsif start = '1' then
        state <= compute_K0;
      else state <= my_idle;
      end if;
       delayAux <= '0';
    when compute_K0 =>
      state <= compute_K1;
    when compute_K1 =>
      state <= compute_K2;
    when compute_K2 =>
      state <= compute_K3;
    when compute_K3 =>
      state <= compute_K2r_8;
    when compute_K2r_8 =>
      state <= compute_K2r_9;
    when compute_K2r_9 =>
```

```
26        state <= compute_even;
27      when compute_even =>
28        state <= compute_odd;
29      when compute_odd =>
30        if ( ((out_count = "00000100") AND (ctrl_enc_sig = '1')
    )
31        OR ((out_count = "00000000") AND (ctrl_enc_sig = '0'))
    ) then
32          state <= compute_K4;
33        else
34          state <= compute_K2r_8;
35        end if;
36      when load_keyA =>
37        state <= my_idle;
38      when compute_K4 =>
39        state <= compute_K5;
40      when compute_K5 =>
41        state <= compute_K6;
42      when compute_K6 =>
43        state <= compute_K7;
44      when compute_K7 =>
45        state <= extra_cycle;
46          finishAux <= '1';
47          if(finishAux = '1')then
48                finishDecAux <= '1';
49          end if;
50        when extra_cycle =>
51           state <= my_idle;
52             delayAux <= '1';
53    end case;
54  end if;
55  end process;
56
57  //....
58
59  finishEnc <= finishAux;
60  finishDec <= finishDecAux;
61  delay <= delayAux;
```

Listing 10: Maquina de Estados do Modulo de Controlo.

Uma vez finalizado o design do Twofish passamos a criar e a gerar o Twofish AXI4 IP com uma interface do modo *slave* e do tipo *lite* com 18 registos. A representação gráfica do IP obtido encontra-se representada na figura 7.

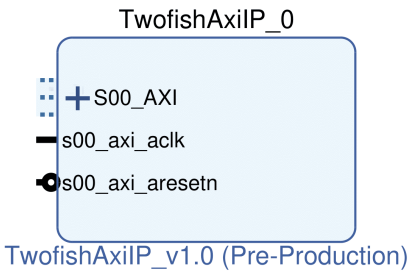

Figura 7: Twofish AXI4 IP.

Com o IP gerado torna-se necessário conectar o nosso Twofish design ao barramento AXI. Dentro do módulo superior do IP AXI, temos uma instância de outro módulo que implementa o protocolo AXI. Primeiramente o que fizemos foi instanciar o projeto twofish que projetamos dentro do módulo superior, listagem 11. Uma vez que temos agora o design dentro do módulo top do AXI IP precisamos de conectá-lo. Primeiro, tivemos que conectar o sinal *s00_ax_aclk* à entrada de clock de nosso design, e *s00_axi_resetn* à entrada de reset de nosso design.

```
// Add user logic here
core main
(
.clk(s00_axi_aclk),
.reset(~s00_axi_aresetn),
.usr_ld_key(),
.usr_start(),
.usr_encrypt(),


.plaintext3_i(),
.plaintext2_i(),
.plaintext1_i(),
.plaintext0_i(),

.globalkey3_i(),
.globalkey2_i(),
.globalkey1_i(),
.globalkey0_i(),

.idle(),
.cryptotext3_o(),
.cryptotext2_o(),
.cryptotext1_o(),
```

```
25 .cryptotext0_o(),
26
27 .initVector3_i(),
28 .initVector2_i(),
29 .initVector1_i(),
30 .initVector0_i()
31 );
32 // User logic ends
```

Listing 11: Instanciacao do Core no Twofish AXI IP top (TwofishAxiIP.v).

Dentro do módulo que implementa o protocolo AXI, encontra-se a declaração dos registos, listagem 12. Neste caso temos treze registos de entrada e cinco registos de saída. Para os registos de entrada precisamos de comentar a sua declaração, uma vez que esses registos serão convertidos em saídas.

```
1 //------------------------------------------------
2 //-- Signals for user logic register space example
3 //------------------------------------------------
4 //-- Number of Slave Registers 18
5 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg0;
6 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg1;
7 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg2;
8 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg3;
9 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg4;
10 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg5;
11 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg6;
12 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg7;
13 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg8;
14 reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg9;
15 reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg10;
16 reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg11;
17 reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg12;
18 reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg13;
19 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg14;
20 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg15;
21 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg16;
22 //  reg [C_S_AXI_DATA_WIDTH-1:0]  slv_reg17;
23 wire   slv_reg_rden;
24 wire   slv_reg_wren;
25 reg [C_S_AXI_DATA_WIDTH-1:0]   reg_data_out;
26 integer  byte_index;
27 reg  aw_en;
```

Listing 12: Declaracao dos registos (TwofishAxiIP_S00_AXI.v).

De seguida, tivemos de criar os registos de saída, correspondentes ao *slv_reg* comentado, e os registos de entrada para transferir dados do IP para o PS.

```
// Users to add ports here
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg0,
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg1,
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg2,
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg3,
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg4,
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg5,
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg6,
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg7,
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg8,
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg14,
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg15,
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg16,
    output reg [C_S_AXI_DATA_WIDTH-1:0] slv_reg17,
    input [C_S_AXI_DATA_WIDTH-1:0] idle,
    input [C_S_AXI_DATA_WIDTH-1:0] crypto3,
    input [C_S_AXI_DATA_WIDTH-1:0] crypto2,
    input [C_S_AXI_DATA_WIDTH-1:0] crypto1,
    input [C_S_AXI_DATA_WIDTH-1:0] crypto0,
// User ports ends
```

Listing 13: Criacao dos Registos de Saida (TwofishAxiIP_S00_AXI.v).

No caso do registo de saída, já está finalizado, mas os registos de entrada têm de ser enviados através da AXI. Para isso, é necessário ir à operação de leitura da interface AXI e substituir *slv_reg9*, *slv_reg10*, *slv_reg11*, *slv_reg12* e *slv_reg13* pelos nossos registos de entrada, listagem 14.

```
always @(*)
begin
      // Address decoding for reading registers
      case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB]
   )
        5'h00   : reg_data_out <= slv_reg0;
        5'h01   : reg_data_out <= slv_reg1;
        5'h02   : reg_data_out <= slv_reg2;
        5'h03   : reg_data_out <= slv_reg3;
        5'h04   : reg_data_out <= slv_reg4;
        5'h05   : reg_data_out <= slv_reg5;
        5'h06   : reg_data_out <= slv_reg6;
        5'h07   : reg_data_out <= slv_reg7;
        5'h08   : reg_data_out <= slv_reg8;
        5'h09   : reg_data_out <= idle;
        5'h0A   : reg_data_out <= crypto3;
        5'h0B   : reg_data_out <= crypto2;
        5'h0C   : reg_data_out <= crypto1;
        5'h0D   : reg_data_out <= crypto0;
        5'h0E   : reg_data_out <= slv_reg14;
        5'h0F   : reg_data_out <= slv_reg15;
        5'h10   : reg_data_out <= slv_reg16;
```

```
22          5'h11   : reg_data_out <= slv_reg17;
23          default : reg_data_out <= 0;
24        endcase
25 end
```

Listing 14: Substituicao dos Registos (TwofishAxiIP_S00_AXI.v).

Após esta modificação, criamos os vários *wires* necessários e adicionamos as novas entradas e saídas à instanciação do módulo AXI, tal como na listagem 15.

```
1  wire w_start;
2  wire w_usrKey;
3  wire w_usrEnc;
4  wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_plaintext3;
5  wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_plaintext2;
6  wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_plaintext1;
7  wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_plaintext0;
8  wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_key3;
9  wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_key2;
10 wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_key1;
11 wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_key0;
12 wire w32_idle;
13 wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_crypto3;
14 wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_crypto2;
15 wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_crypto1;
16 wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_crypto0;
17 wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_reg1;
18 wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_reg2;
19
20 wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_vector3;
21 wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_vector2;
22 wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_vector1;
23 wire [C_S00_AXI_DATA_WIDTH-1 : 0] w32_vector0;
24
25 assign w32_reg2 = {31'h0,w32_idle};
26
27
28 // Instantiation of Axi Bus Interface S00_AXI
29 TwofishAxiIP_v1_0_S00_AXI # (
30 .C_S_AXI_DATA_WIDTH(C_S00_AXI_DATA_WIDTH),
31 .C_S_AXI_ADDR_WIDTH(C_S00_AXI_ADDR_WIDTH)
32 ) TwofishAxiIP_v1_0_S00_AXI_inst (
33 .slv_reg0(w32_reg1),
34 .slv_reg1(w32_plaintext3),
35 .slv_reg2(w32_plaintext2),
36 .slv_reg3(w32_plaintext1),
37 .slv_reg4(w32_plaintext0),
38 .slv_reg5(w32_key3),
39 .slv_reg6(w32_key2),
```

```
.slv_reg7(w32_key1),
.slv_reg8(w32_key0),
.idle(w32_reg2),
.crypto3(w32_crypto3),
.crypto2(w32_crypto2),
.crypto1(w32_crypto1),
.crypto0(w32_crypto0),
.slv_reg14(w32_vector3),
.slv_reg15(w32_vector2),
.slv_reg16(w32_vector1),
.slv_reg17(w32_vector0),
.S_AXI_ACLK(s00_axi_aclk),
.S_AXI_ARESETN(s00_axi_aresetn),
.S_AXI_AWADDR(s00_axi_awaddr),
.S_AXI_AWPROT(s00_axi_awprot),
.S_AXI_AWVALID(s00_axi_awvalid),
.S_AXI_AWREADY(s00_axi_awready),
.S_AXI_WDATA(s00_axi_wdata),
.S_AXI_WSTRB(s00_axi_wstrb),
.S_AXI_WVALID(s00_axi_wvalid),
.S_AXI_WREADY(s00_axi_wready),
.S_AXI_BRESP(s00_axi_bresp),
.S_AXI_BVALID(s00_axi_bvalid),
.S_AXI_BREADY(s00_axi_bready),
.S_AXI_ARADDR(s00_axi_araddr),
.S_AXI_ARPROT(s00_axi_arprot),
.S_AXI_ARVALID(s00_axi_arvalid),
.S_AXI_ARREADY(s00_axi_arready),
.S_AXI_RDATA(s00_axi_rdata),
.S_AXI_RRESP(s00_axi_rresp),
.S_AXI_RVALID(s00_axi_rvalid),
.S_AXI_RREADY(s00_axi_rready)
);
```

Listing 15: Instanciacao do Modulo AXI (TwofishAxiIP.v).

Concluindo só falta ligar à listagem 11, os novos registos do módulo AXI. O resultado encontra-se na seguinte listagem 16.

```
// Add user logic here
core main
(
.clk(s00_axi_aclk),
.reset(~s00_axi_aresetn),
.usr_ld_key(w32_reg1[1]),
.usr_start(w32_reg1[2]),
.usr_encrypt(w32_reg1[0]),


.plaintext3_i(w32_plaintext3),
```

```
12 .plaintext2_i(w32_plaintext2),
13 .plaintext1_i(w32_plaintext1),
14 .plaintext0_i(w32_plaintext0),
15
16 .globalkey3_i(w32_key3),
17 .globalkey2_i(w32_key2),
18 .globalkey1_i(w32_key1),
19 .globalkey0_i(w32_key0),
20
21 .idle(w32_idle),
22 .cryptotext3_o(w32_crypto3),
23 .cryptotext2_o(w32_crypto2),
24 .cryptotext1_o(w32_crypto1),
25 .cryptotext0_o(w32_crypto0),
26
27 .initVector3_i(w32_vector3),
28 .initVector2_i(w32_vector2),
29 .initVector1_i(w32_vector1),
30 .initVector0_i(w32_vector0)
31 );
32 // User logic ends
```

Listing 16: Instanciacao do Core no Twofish AXI IP top (TwofishAxiIP.v).

Assim, visto que, o nosso IP está criado, procedemos à criação do nosso *block design* com a integração do IP. Na figura 8 encontra-se o *block design* final. Começamos por adicionar o IP e o *ZYNQ PS*. A automação de blocos no PS adiciona DDR e vários pinos de E/S periféricos, mas apenas deixamos a UART1, GPIO e SD0, desabilitando todos os outros. Uma modificação importante foi feita na configuração de *clock* no *ZYNQ PS*, mais precisamente nos *clocks* da *PL Fabric*. Aqui, alteraramos a frequência do *FCLK_CLK0* para 45MHz, a fim de atender aos requisitos de temporização.

Após gerar o *bitstream* é possível exportar o hardware e criar a aplicação para *Petalinux*.

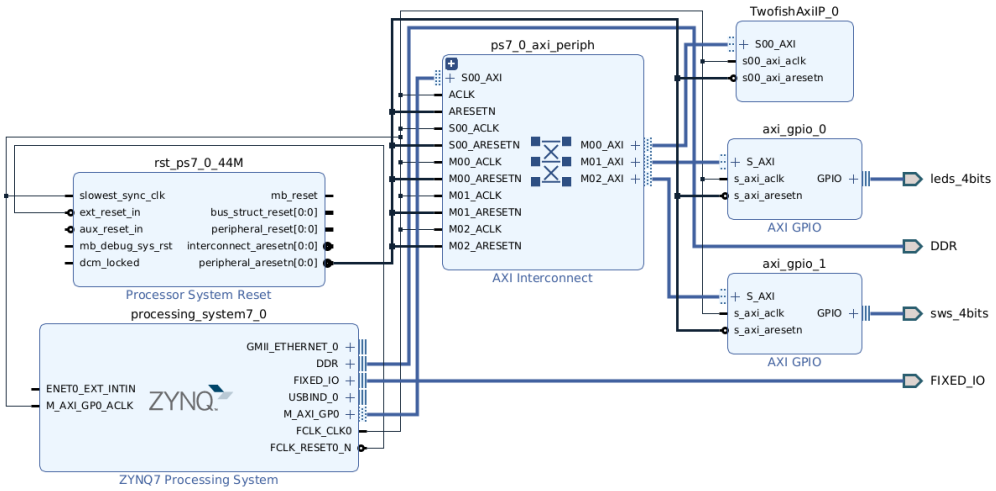

Figura 8: Twofish Block Design.

5.2 Aplicação final para a imagem Linux

A aplicação desenvolvida terá de fornecer um conjunto de alternativas ao utilizador, como foi demonstrado na fase de Design. Para isso, desenvolveu-se um menu bastante simples que possibilita ao utilizador escolher uma das aplicações oferecidas pelo programa. O código respetivo pode ser encontrado nas referências.

Decidiu-se dividir a estrutura do programa entre duas APIs. A primeira está relacionada com operações básicas relativas à configuração do IP Twofish e não deve ser acedida pelo utilizador.

Como se pode observar no listing 17, esta interface permite inicializar a plataforma de encriptação, preparar a chave e realizar a encriptação ou desencriptação de um único bloco de dados (128 bits).

```
1 void Twofish_initialise();
2
3 void Twofish_finish();
4
5 void Twofish_prepare_key(Twofish_Byte *key, int key_len,
    Twofish_key *xkey);
6
7 void Twofish_encrypt(Twofish_key *xkey, Twofish_Byte *p,
    Twofish_Byte *c);
8
9 void Twofish_decrypt(Twofish_key *xkey, Twofish_Byte *c,
```

```
    Twofish_Byte *p);
```

Listing 17: Interface abstrata.

A construção desta interface manteve a mesma estrutura que a API utilizada ao nível de software [34], permitindo manter a abstração à plataforma alvo. Relativamente à API desenvolvida para auxilio aos casos de uso, encontra-se representada no seguinte *listing*.

```
1  // Encrypt the file
2  int Enc_file(const char *encryption_file);
3  // Decrypt the file
4  int Dec_file(const char *encryption_file);
5  // Encrypt image in CBC mode
6  int Enc_img(const char *input_filename);
7  // Decrypt image in CBC mode
8  int Dec_img(const char *input_filename);
9  // General application
10 int general_app(uint8_t);
```

Listing 18: Interface do utilizador.

Por fim, para gerir o IP através do sistema operativo é necessário aceder ao endereço do IP e escrever e ler os diferentes registos. Para fazer isso, recorreu-se à função *mmap* disponível no *Unix*. Esta função é utilizada para mapear um ficheiro numa região da memória e assim aceder a essa região da memória recorrendo ao seu endereço. Assim, efetuou-se o mapeamento na região da memória dedicada aos registos do IP do *Twofish*, como se demonstra no listagem abaixo.

```
1      /* Map memory */
2      map_base = mmap(NULL, map_size, PROT_READ | PROT_WRITE,
       MAP_SHARED, fd, twofish_baseaddr);
3      if (map_base == NULL) {
4          printf("Can't mmap\n");
5          //return -1;
6      }
7      /* Write memory */
8    virt_addr = (volatile unsigned int *)map_base;
```

Listing 19: Mapeamento da memoria.

Assim, para aceder aos registos do IP, recorreu-se ao apontador retornado pela função e especificou-se o endereço do registo que se pretende alterar, listagem 20.

```
1  // Assign initial vector values
2  virt_addr[14] = IV3;
3  virt_addr[15] = IV2;
4  virt_addr[16] = IV1;
```

```
5   virt_addr[17] = IV0;
6
```

Listing 20: Escrita em registos do IP.

Nesta porção de código está-se a fazer a escrita dos valores do *Initial Vector* na região de memória em que se encontra.

6 Verificação

6.1 Testes de funcionalidade

Escreveu-se um *testbench* para verificar a correta funcionalidade da encriptação e desencriptação do IP no modo CBC. O listing 21 demonstra o comportamento do *testbench*.

```
1  module testbenchVerilog();
2
3  reg [127:0]import;
4  reg [127:0]inkey;
5  reg clk = 0;
6  reg reset;
7  reg usr_ld_key;
8  reg usr_start;
9  reg usr_encrypt;
10 wire idle;
11 wire [127:0] outCiphertext;
12 reg [127:0]initVector;
13
14 core main
15 (
16   .clk(clk),
17   .reset(reset),
18   .usr_ld_key(usr_ld_key),
19   .usr_start(usr_start),
20   .usr_encrypt(usr_encrypt),
21
22   .plaintext3_i(import[127:96]),
23   .plaintext2_i(import[95:64]),
24   .plaintext1_i(import[63:32]),
25   .plaintext0_i(import[31:0]),
26
27   .globalkey3_i(inkey[127:96]),
28   .globalkey2_i(inkey[95:64]),
29   .globalkey1_i(inkey[63:32]),
30   .globalkey0_i(inkey[31:0]),
31
32     .idle(idle),
33
34   .cryptotext3_o(outCiphertext[127:96]),
35   .cryptotext2_o(outCiphertext[95:64]),
36   .cryptotext1_o(outCiphertext[63:32]),
37   .cryptotext0_o(outCiphertext[31:0]),
38
39   .initVector3_i(initVector[127:96]),
40   .initVector2_i(initVector[95:64]),
```

```
41   .initVector1_i(initVector[63:32]),
42   .initVector0_i(initVector[31:0])
43 );
44
45
46 always begin
47     #10 clk =~clk;
48 end
49 initial begin
50     //Reset all ports
51     reset = 1'b1;
52     usr_ld_key = 1'b0;
53     usr_start = 1'b0;
54     usr_encrypt = 1'b0;
55     import ='bx;
56     inkey ='bx;
57
58     @(posedge clk);
59     import = 128'h0;     // Input message value
60     initVector = 128'hda39a3ee5e6b4b0d3255bfef95601890; //
    Init vector value
61     inkey = 128'h0; // Key value
62     @(posedge clk);
63     reset = 1'b0;
64     repeat(4) @(posedge clk);
65     usr_ld_key = 1'b1;  // Load the key value from the input
66     repeat(4) @(posedge clk);
67     @(posedge clk);
68     usr_ld_key = 1'b0;  // Reset the load key flag
69     usr_encrypt = 1'b0; // Specify the decryption mode
70     @(posedge clk);
71     usr_start = 1'b1;  // Start the decryption
72     repeat(4) @(posedge clk);
73     usr_start = 1'b0;  // Reset the start port
74
75     repeat(100) @(posedge clk);  // Wait 100 clock cycles to
    finish the first decryption
76     usr_start = 1'b1; // Start new decryption
77
78 end
79 endmodule
```

Listing 21: Modulo do testbench efetuado.

A figura 9 demonstra os resultados obtidos ao correr a simulação com o IP no modo de encriptação.

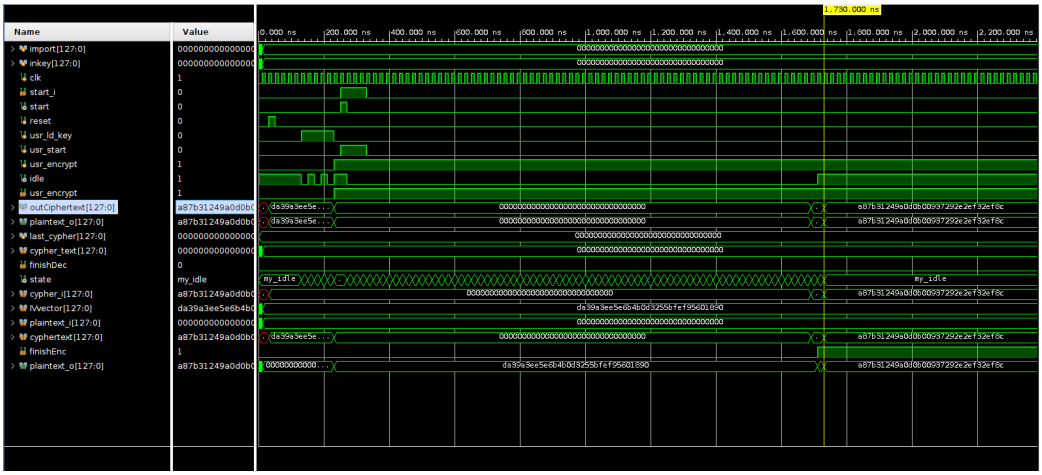

Figura 9: Simulação de uma encriptação.

Para efeito de testes utilizou-se uma chave e uma mensagem constituídas apenas por elementos com o valor zero. Como se pode observar os valores estão de acordo com o valor padrão.

A figura 10 demonstra os resultados obtidos ao correr a simulação com o IP no modo de desencriptação.

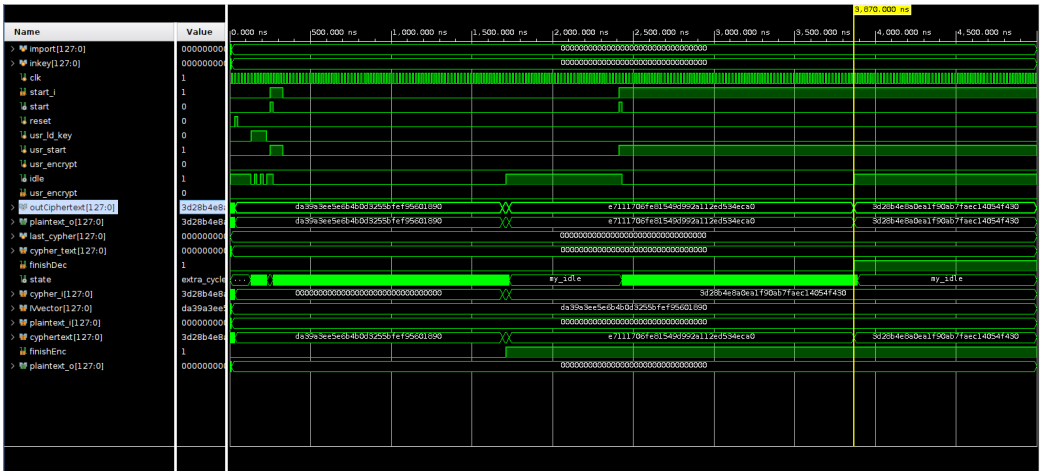

Figura 10: Simulação de duas desencriptações.

Da mesma forma, utilizou-se uma chave e uma mensagem constituídas apenas por elementos com o valor zero. Como se pode observar os valores também estão de acordo com os valores padrão.

6.2 Testes de performance

Foram realizados testes que permitem avaliar a performance do Twofish em hardware reconfigurável e em software. Foi também testado diversos casos aplicacionais para diferentes tamanhos de dados de entrada.

Para efetuar os testes de performance recorreu-se à biblioteca *C Standard* que possui a função *clock*, de modo a obter o número de *systicks* contabilizados durante a execução de um programa. Este método permite a medição do tempo de CPU gasto em funções, que pode ser utilizado para o cálculo final do *throughput* recorrendo à expressão: *n_dados*/(n_systicks/cloksPerSecond). A variável *n_dados* representa a quantidade de dados que são processados pela sistema, *n_systicks* simboliza o número de *systicks* contabilizados durante esse período de execução, e *cloksPerSecond* representa a quantidade de ciclos de relógio que foram realizados num segundo (estipulado pelo *time.h*).

A tabela 6.2 demonstra as características de performance do algoritmo *Twofish* para ambas as implementações. A implementação em software foi programada em Linguagem C e executada num *Intel i7 9th Generation 4,50 GHz*.

Mode	**Device**	**Cycles**	**Number of blocks**	**Throughput**
Encryption	Zybo Z7	16	1	8 Gbit/s
	Software	251	1	510 Kbit/s
Decryption	Zybo Z7	14	1	9,14 Gbit/s
	Software	243	1	526,75 Kbit/s
Encrypt File	Zybo Z7	2060	31	1,93 Gbit/s
	Software	1707	31	2,32 Gbit/s
Decrypt File	Zybo Z7	2023	31	1,96 Gbit/s
	Software	1500	31	2,65 Gbit/s
Encrypt Image	Zybo Z7	117217	2813	3,07 Gbit/s
	Software	-	-	-
Decrypt Image	Zybo Z7	116543	2813	3,09 Gbit/s
	Software	-	-	-

Tabela 1: Caraterísticas de desempenho do algoritmo Twofish.

Através da análise dos dados na tabela verifica-se que a implementação do algoritmo em SoC supera a implementação em software num fator de 17. Verifica-se também que a desencriptação de um único bloco de dados (128 bits), supera a eficiência do algoritmo *Twofish* em software. No entanto, para processamento de um elevado número de bloco de dados a implementação software demonstra-se mais eficiente, atingindo um fator próximo

de 1.3. A implementação do algoritmo em hardware reconfigurável para blocos de dados pequenos demonstra-se mais vantajosa, uma vez que a FPGA possuí recursos dedicados à aplicação e múltiplas operações podem ocorrer simultaneamente (paralelismo de dados). Para uma fácil visualização e interpretação dos resultados, através do seguinte gráfico, figura 11, facilmente se compreende a grande diferença que existe entre as duas implementações, em hardware e software. Para a encriptações e desencriptações isoladas, para um bloco dados, pode se observar um *throughtput* bastante maior para a implementação em hardware relativamente à implementação para software. Porém com o aumento de bloco de dados a encriptar a implementação em software apresenta uma melhoria na sua eficiência e *throughtput*. Para um número grande de blocos de dados, 2813, referentes à encriptação e desencriptação de uma imagem, compreende-se o baixo *throughtput* devido à dimensão deste ficheiro. Apenas se realizou a implementação em hardware, visto que, a implementação em software não encriptava imagens.

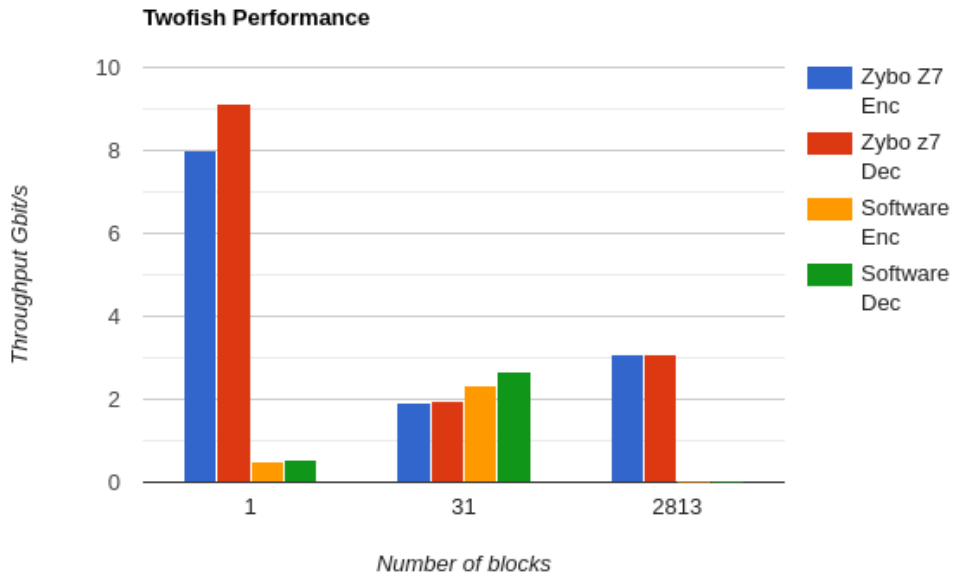

Figura 11: Throughput baseado no número de blocos tanto em encriptação como em desencriptação, na Zybo z7 e em software.

Apesar de não termos acesso à função que efetua a encriptação de ficheiros, podemos assumir que o programador recorreu a técnicas de *multi-threading* ou *multi-processing* para fazer aumentar o desempenho do sistema

num processamento de número de dados elevado. Por outro lado, a implementação em sistemas híbridos utilizada não recorre a mecanismos de paralelismo uma vez que a interface utilizada para comunicar com o PS foi a *AXI-Lite*. Este protocolo requer uma utilização de recursos reduzida e é simples de implementar, no entanto é ideal para aplicações com baixa largura-de-banda e baixa latência, o que não é o caso para encriptação de um grande número de dados.

7 Conclusão

A importância dos algoritmos de encriptação simétrica em sistemas híbridos foi examinada e os diferentes modos de computação investigados. Resultou-se numa aplicação que permite ao utilizador garantir a segurança de dados sensíveis e tirar proveito dos desempenhos da implementação do algoritmo em hardware reconfigurável. De seguida, serão identificadas algumas considerações finais sobre o projeto desenvolvido.

Relativamente ao modo de computação, ECB versus CBC, facilmente se percebeu que iria existir um *trade-off* em termos de segurança e características de desempenho. No que diz respeito à segurança, o modo CBC oferece maior proteção em comparação ao modo ECB. O CBC utiliza a realimentação do bloco encriptado anterior como entrada para o próximo bloco, criando uma dependência entre os blocos e prevenindo ataques de substituição e reordenação. Já o ECB não possui essa realimentação, resultando em blocos de texto encriptado idênticos para o mesmo bloco de texto normal e tornando-o mais suscetível a ataques de análise de padrões. Todavia em relação ao desempenho, o modo CBC requer que os blocos de texto plano sejam processados sequencialmente, com cada bloco dependendo do bloco anterior. Isso limita o paralelismo e pode resultar num desempenho mais lento em comparação ao modo ECB. Por outro lado, o ECB permite que cada bloco de texto plano seja encriptado ou desencriptado independentemente, permitindo maior paralelismo do processo e podendo resultar num desempenho mais rápido em sistemas que podem aproveitar o paralelismo. Após estudo, o modo de computação escolhido foi o CBC uma vez que é indicado para a encriptação de dados que exigem maior segurança e confidencialidade. Ele é útil quando há a necessidade de proteger dados sensíveis, como informações pessoais, comunicações seguras e transferência de arquivos confidenciais o que encaixa perfeitamente nos nossos casos de uso. O modo ECB é mais adequado para aplicativos onde a segurança não é a principal preocupação e a simplicidade e a eficiência de processamento são mais importantes, o que não encaixa com as nossas necessidades.

Em termos de desempenho e segurança, para a encriptação e desencriptação isolada (encriptação/desencriptação de um único bloco de dados), as implementações baseadas em hardware oferecem vantagens significativas em relação às soluções baseadas em software, embora sejam menos versáteis. Por outro lado, as soluções baseadas em software são flexíveis, mas dependem de CPUs de alto desempenho e muitas vezes introduzem atrasos na comunicação. No entanto, ao aproveitar o alto nível de paralelismo oferecido pelas *FPGAs* e a flexibilidade dos sistemas embebidos, é possível desenvolver

soluções de segurança que sejam versáteis e com alto desempenho.

Analisando a encriptação e desencriptação para pequenos blocos de dados, as implementações de hardware superam as implementações de software, podendo o fator entre as duas implementações atingir o valor 17. Como referido anteriormente, este desempenho superior é principalmente atribuível ao paralelismo explorado pelas implementações de hardware. Exemplos desse paralelismo explorado incluem o cálculo simultâneo de dezasseis cópias da *S-Box* de uma determinada *round function*, bem como a capacidade de calcular uma *round function* completa num único ciclo de relógio. Embora a implementação de software opere numa frequência de relógio significativamente mais alta, essa vantagem é amplamente compensada pelo aumento no número de ciclos de relógio necessários para realizar uma *round function*. Ao minimizar o número de ciclos de relógio por encriptação, as implementações de hardware alcançam uma eficiência maior do que sua contraparte de software. Além disso, é possível explorar paralelismo adicional por meio do uso de pipeline de hardware, permitindo a encriptação de um bloco de dados de 128 bits por ciclo de relógio, uma vez que a latência do pipeline tenha sido atendida. Através do uso desses métodos, as implementações de hardware alcançam resultados de rendimento muito superiores em comparação com uma implementação de software.

Apesar de o desempenho ser significativo para uma única encriptação, averiguou-se que com o aumento de dados a processar a implementação em SoC resultava em perdas de eficiência do algoritmo devido à interface utilizada pelo IP. Desta forma, pretendemos no futuro dar continuidade a este projeto implementando uma interface que permita ao IP usufruir das capacidades de paralelismo que são características em implementações em hardware reconfigurável, como é o caso da interface *AXI Master*. Esta interface oferece enumeras vantagens para esta implementação, nomeadamente, o fato de possuir canais individuais para escrita e leitura o que faz diminuir a latência no acesso de dados, e suporte um modo *burst-based* que permite atingir desempenhos até 17 GB/s.

Outro fator interessante para trabalhar no futuro seria oferecer ao utilizador a possibilidade de selecionar entre diversos métodos de computação, uma vez que já se comprovou que nem sempre o método CBC é o mais adequado.

Relativamente à aplicação criada, esta oferece ao utilizador um método de encriptar ficheiros ou imagens, no entanto, esta poderá ser otimizada para efetuar a encriptação de diretórios. Isto retira encriptação manual de cada ficheiro e evita o esquecimento das chaves secretas, uma vez que apenas é necessário uma única chave para desencriptar todos os ficheiros.

Concluindo, a implementação do algoritmo *Twofish* em sistemas híbridos demonstra-se de elevado interesse para as aplicações atuais na área de en-

criptação uma vez que, o PS da placa garante a flexibilidade da implementação, já que permite uma abstração da plataforma em que se encontra o IP, e ainda, porque conseguem superar significativamente o desempenho de implementações realizadas em software.

Referências

[1] D. Le, C. Bhatt, and M. Madhukar, *Security Designs for the Cloud, IoT, and Social Networking*, 2020.

[2] D. Lisonek and M. Drahanský, "Sms encryption for mobile communication." 2008.

[3] H. Gupta and V.Sharma, "Role of multiple encryption in secure electronic transaction." 2011.

[4] R. C. A. G. J. Mandal, *Machine Learning Techniques and Analytics for Cloud Security*, 2022.

[5] A. Boicea, F. Radulescu, C. Truica, and C. Costea, "Database encryption using asymmetric keys: A case study." 2017.

[6] J. Rodríguez-Andina, M. Valdés-Peña, and M. Moure, "Advanced features and industrial applications of fpgas - a review," pp. 858–862, 2015.

[7] Y. Yonezawa, H. Nakao, Y. Nakashima, A. Vithanage, T. Kanehira, and Y. Ueno, "Model-based development of high-current-density point-of-load converter of high performance fpga for telecommunication application," 2017.

[8] B. Rong and C.-H. Liu, "A fingerprint payment system based on fpga," 2016.

[9] M. Kabra, P. H. C., and M. Rao, "Design and evaluation of performance-efficient soc-on-fpga for cloud-based healthcare applications." 2022.

[10] Moving a generation ahead with all programmable fpgas, socs, and 3d ics. [Online]. Available: https://www.xilinx.com/publications/prod_mktg/Generation-Ahead-Backgrounder.pdf

[11] P. Visconti, R. Velazquez, S. Capoccia, and R. de Fazio, "High-performance aes-128 algorithm implementation by fpga-based soc for 5g communications," 2021.

[12] M. Y. K. S. Sutikno, "Implementation of bc3 encryption algorithm on fpga zynq-7000," 2017.

[13] S. Chandra, S. Paira, S. Alam, and G. Sanyal, "A comparative survey of symmetric and asymmetric key cryptography," 2014.

[14] B. Academy. O que é criptografia de chave simétrica? [Online]. Available: https://academy.binance.com/pt/articles/what-is-symmetric-key-cryptography

[15] W. Stallings, *The principles and practice of cryptography and network security 7th edition.* Pearson Education, 2017.

[16] J. Katz and Y. Lindell, "Introduction to modern cryptography," 2014.

[17] R. Awati. Twofish. [Online]. Available: https://www.techtarget.com/searchsecurity/definition/Twofish

[18] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., & Ferguson, N. (1998). Twofish: A 128-Bit Block Cipher.

[19] B. Schneier. Products that use twofish. [Online]. Available: https://www.schneier.com/academic/twofish/products/

[20] 2011, S. Rizvi; S. Hussain; N. Wadhwa. Performance Analysis of AES and TwoFish Encryption Schemes.

[21] X. C. and W. K., "Website information retrieval of web database based on symmetric encryption algorithm," *J Ambient Intell Human Comput*, 2021. [Online]. Available: https://doi.org/10.1007/s12652-020-02819-w

[22] R. K. Snider, *Advanced Digital System Design using SoC FPGAs*, 2023.

[23] A. Moore, *FPGAs For Dummies*, 2017.

[24] K. Aoki and H. Lipmaa, "Fast software implementations of aes candidates." 2000.

[25] K. Gaj and P. Chodowiec, "Fast implementation and fair comparison of the final candidates for advanced encryption standard using field programmable gate arrays." 2001.

[26] J. Zhang and G. Qu, "Recent attacks and defenses on fpga-based systems," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 3, aug 2019. [Online]. Available: https://doi.org/10.1145/3340557

[27] J.-B. Note and Rannaud., "From the bitstream to the netlist." 2008.

[28] Y. Kim, E. Jung, and C. Kim, "Bitstream reverse engineering of microsemi's versatile-based fpgas." 2021.

[29] M. Jeong, J. Lee, E. Jung, Y. H. Kim, and K. Cho, "Extract lut logics from a downloaded bitstream data in fpga." 2018.

[30] F. Rahman, F. Farahmandi, , and M. Tehranipoor, "An end-to-end bitstream tamper attack against flip-chip package," 2021.

[31] S. Drimer, "Volatile fpga design security - a survey," 2008.

[32] S. Q. K. J. Boone. An introduction to fault injection. [Online]. Available: https://research.nccgroup.com/2021/07/07/an-introduction-to-fault-injection-part-1-3/

[33] "Twofish core." [Online]. Available: https://opencores.org/projects/twofish_team

[34] Twofish standalone. [Online]. Available: https://github.com/sergioagp/twofish_standalone