Designing Secure Cryptographic Accelerators with Information Flow Enforcement: A Case Study on AES

Zhenghong Jiang, Hanchen Jin, G. Edward Suh, Zhiru Zhang
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{jz763,hj424,gs272,zhiruz}@cornell.edu

ABSTRACT

Designing a secure cryptographic accelerator is challenging as vulnerabilities may arise from design decisions and implementation flaws. To provide high security assurance, we propose to design and build cryptographic accelerators with hardware-level information flow control so that the security of an implementation can be formally verified. This paper uses an AES accelerator as a case study to demonstrate how to express security requirements of a cryptographic accelerator as information flow policies for security enforcement. Our AES prototype on an FPGA shows that the proposed protection has a marginal impact on area and performance.

CCS CONCEPTS

• **Security and privacy → Hardware security implementation**; **Information flow control**;

ACM Reference Format:
Zhenghong Jiang, Hanchen Jin, G. Edward Suh, Zhiru Zhang. 2019. Designing Secure Cryptographic Accelerators with, Information Flow Enforcement: A Case Study on AES. In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3316781.3317798

1 INTRODUCTION

In modern system-on-chips (SoCs), cryptography plays an integral role in protecting the confidentiality and integrity of information. For example, SoCs may need AES for encrypted data storage and use RSA/ECC for key exchange in a protected communication. The extensive use of cryptography has propelled the development of hardened cryptographic (crypto) accelerators for better performance and energy-efficiency. However, the dissimilarities between accelerators and the increasing design complexity bring challenges to the security of cryptographic hardware accelerators.

Security vulnerabilities can be introduced into crypto accelerators from various aspects, including design decisions [12], implementation flaws [6], debug peripherals [10], and even hardware Trojans [16]. Though numerous efforts have been made to protect crypto hardware, most of them only focus on specific vulnerabilities [8, 16]. In order to provide high assurance for crypto accelerators, we need a methodology that is capable of systematically checking a broad range of security requirements at design time.

In most modern SoCs, crypto accelerators are often shared among multiple applications/users. For example, multiple users in the cloud share the same AES accelerator to process encryption requests in the secure sockets layer (SSL) protocol. However, efficient and secure sharing of an accelerator is not an easy task. The traditional method of sharing an accelerator at the coarse granularity only allows one program (user) to use the accelerator at a time. For such coarse-grained sharing, security protection can largely focus on interfaces [14]. On the other hand, the coarse-grained sharing limits performance, especially for deeply-pipelined accelerators, as the entire pipeline must be drained and refilled when switching users. To improve performance, accelerators need to allow more fine-grained sharing so that data from different users can be processed inside the accelerator simultaneously. Unfortunately, the fine-grained sharing increases the difficulty of data isolation and leads to higher security risks.

In this paper, we propose to use hardware-level information flow control (IFC) in designing secure crypto accelerators while supporting fine-grained sharing. Hardware-level IFC systematically examines information flows in hardware modules and can provide strong security assurance to hardware implementations at design time using either a security-typed HDL [13, 23] or information-flow tracking logic [1, 21]. In the paper, we demonstrate that a broad range of security requirements of a crypto accelerator can be expressed as information flow policies and can be systematically verified using an IFC tool with both low design effort and low implementation overhead. As a case study, we develop a secure AES accelerator that leverages information flow control to verify its security requirements. The accelerator is implemented in a security-typed HDL at RTL, and the implementation is statically verified to be free of disallowed information flows, including timing channels. While its security properties are verified at design time, the accelerator also uses security tags and tracking logic to support flexible information flow policies at runtime.

The main contributions of this work are twofold:

(1) We show that strong security protection for crypto accelerators can be provided with high assurance using hardware-level information flow control. The main security requirements of a crypto accelerator can be expressed as information flow policies and verified at design time with low overhead.
(2) Using an AES accelerator prototype, we show how to achieve both security and efficiency together in a crypto accelerator using a careful combination of design-time and runtime policies. The runtime policies provide flexibility for the practical usability while the design-time policies ensure a formal guarantee of security on the accelerator implementation.

The rest of the paper is structured as follows: Section 2 discusses some known attacks on AES hardware, describes the threat model, and introduces the concept of HDL-level information flow control and nonmalleable downgrading. Section 3 describes the design

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317798

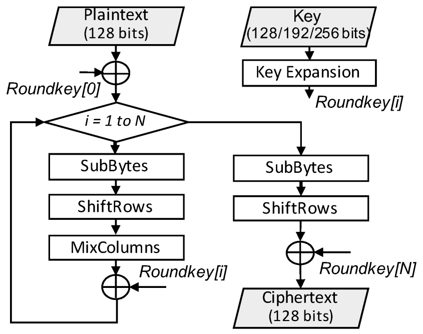

Figure 1: Typical AES encryption flow — Different key length requires different numbers of computing iterations: $N = 10$ for 128-bit key, $N = 12$ for 196-bit key, and $N = 14$ for 256-bit key.

decisions we made to the proposed AES accelerator, and illustrates how the main security properties of the accelerator can be expressed as hardware-level information flow policies. Section 4 presents the evaluation results from our AES accelerator prototype. Section 5 discusses related work, followed by conclusions in Section 6.

2 PRELIMINARIES

In this section, we first briefly summarize some known attacks on the AES hardware. Then, we describe the threat model considered in this paper. In the end, we introduce the concept of HDL-level information flow control and nonmalleable downgrading that we used to verify the security of the accelerator implementations.

2.1 Attacks on AES Hardware

AES (Advanced Encryption Standard) is a symmetric block cipher standard broadly used for encryption/decryption of sensitive data. AES encrypts a 128-bit plaintext block into a 128-bit ciphertext block by using a 128/192/256-bit cryptographic key, shown in Figure 1. A large message can be divided into multiple 128-bit blocks and fed into the AES engine in sequence. The extensive use of AES has propelled the development of custom hardware accelerators [17, 22] for better performance but also makes it a target for malicious attacks. Considering the prevalence of AES accelerators in SoCs, we choose it as a representative case study to explain our proposed protection method without losing generality.

Rather than discovering weaknesses in the AES algorithm, it is often more profitable to exploit vulnerabilities in its hardware implementations [18]. For example, prior work has demonstrated that disclosure of internal signals, via implementation flaws or rogue debug interfaces, can significantly reduce the effort in recovering secret keys [6, 10]. Moreover, attackers can leverage the side effects of a hardware implementation to infer secret keys. For example, one previous attack [12] uses key-dependent execution time of an AES implementation to infer its secret key. AES accelerators are often heavily optimized for performance, and the complex optimizations make designing a secure AES engine a challenging task without a systematic methodology [2].

2.2 Threat Model

In a typical heterogeneous SoC, multiple user applications can run on a processor concurrently, which also share crypto accelerators, DMA engines, and other peripherals. As shown in Figure 2, each user application has a security label to identify its security privilege and holds a secret key for encryption/decryption of its private data.

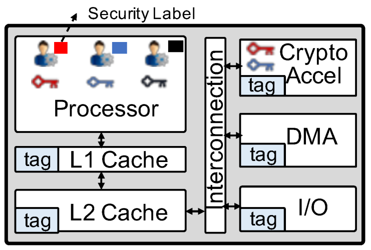

Figure 2: Modern SoCs running under multiple security levels — Multiple user applications simultaneously share the crypto accelerators and each user holds a secret key for its data encryption/decryption.

```
class CacheTags extends Module {
  val io = IO(new Bundle {
    val we    = Input(Bool(),       Label(public, trusted))
    val way   = Input(UInt(1.W),    Label(public, trusted))
    val tag_i = Input(UInt(19.W),   Label(public, DL(way)))
    val index = Input(UInt(8.W),    Label(public, trusted))
    val tag_o = Output(UInt(19.W),  Label(public, DL(way)))
  })
  val tag_0 = Reg(Vec(256, UInt(19.W)), Label(public, trusted))
  val tag_1 = Reg(Vec(256, UInt(19.W)), Label(public, untrusted))
  when (io.we) {
    when       (io.way === 0.U) { tag_0(way) := io.tag_i; }
    .otherwise                  { tag_1(way) := io.tag_i; }
  } .otherwise {
    when       (io.way === 0.U) { io.tag_o := tag_0(way); }
    .otherwise                  { io.tag_o := tag_1(way); }
  }
}
```

Figure 3: Cache tags in ChiselFlow description — DL is a dependent label that DL(0) indicates trusted and DL(1) indicates untrusted. `tag_i` and `tag_o` port switch their integrity levels depending on which way is selected.

In this paper, we consider an accelerator that is shared at a fine granularity where it can encrypt data from different users with different keys concurrently. The fine-grained sharing improves efficiency but poses a challenge for security. We assume that an adversary controls one or more applications on the SoC and can attack a crypto accelerator by misusing the interfaces for the applications. For example, the adversary may try to infer a secret that belongs to another security level or maliciously affect the encryption/decryption of another application by observing and manipulating data at or below his/her security level. The adversary can exploit implementation flaws or backdoors in an accelerator. The adversary may also use timing channels, which can be exploited in software. However, we assume that the adversary has no physical access to the SoC; therefore, physical attacks, such as fault inject and power side-channel attacks are not considered.

2.3 HDL-Level Information Flow Control

Information flow control is a security mechanism that provides security assurance by tracking information flows inside a target system. It associates a label to each data, monitors the data flowing from sources to sinks, and ensures that secret data cannot leak to public for confidentiality or that untrusted inputs cannot contaminate trusted data for integrity. HDL-level information flow control applies IFC to HDL (Hardware Description Language) in order to provide security assurance for hardware [13]. For example, given two security labels ℓ and ℓ', if label ℓ is less restrictive than label ℓ', it is written as $\ell \sqsubseteq \ell'$. In general, IFC enforces that a signal with label ℓ cannot be affected by another signal with label ℓ'. In other words, a more restrictive signal cannot influence a less restrictive signal. ChiselFlow is a newly developed security-typed HDL on the top of Chisel. Unlike prior security-typed HDLs, ChiselFlow manages confidentiality and integrity explicitly [7]. It adopts the 2-tuple label format $\ell = (c, i)$, where c and i represents confidentiality and integrity. Given two labels ℓ and ℓ', $\ell \sqsubseteq_C \ell'$ means ℓ' has higher confidentiality, and $\ell \sqsubseteq_I \ell'$ means ℓ has higher integrity.

Besides static security labels, ChiselFlow also supports dynamic (dependent) labels to enable fine-grained sharing of hardware resources. A signal with a static label belongs to a fixed security level for its entire lifetime. On the other hand, the security level of a signal with a dependent label is determined by the value of another signal. Figure 3 shows a ChiselFlow example of a shared cache tag module. In the module, the cache is statically partitioned: `tag_0` holds trusted data and carries a static label of `(public, trusted)`, whereas `tag_1` holds untrusted data and carries a static label of `(public, untrusted)`. The tag data input and output have a dependent label of `(public, DL(way))`, which means their integrity levels depend on the value of signal `way`. When `way` has a value of `0`, the tag input is treated as `trusted`; it receives data from the trusted level and writes data to the trusted `tag_0`. When `way` has a value of `1`, the tag input is treated as `untrusted`; it receives data from the untrusted level and writes data to the untrusted `tag_1`. Though the cache tag memory is partitioned, the data input and output ports are shared among two security levels.

2.4 Nonmalleable Downgrading

Information flow control generally enforces noninterference to prohibit every flow of information that violates the security policy. Unfortunately, noninterference is known to be too restrictive for most practical systems. For example, in cryptography, ciphertext contains information from the crypto key, but is considered safe and should be allowed to be released to public channels. Therefore, it is necessary to introduce *downgrading* to explicitly allow exceptions to an information flow policy. Downgrading in confidentiality is called *declassification* and downgrading in integrity is called *endorsement.* Downgrading increases usability but also weakens the security of IFC. To limit the risk of downgrading, *nonmalleable* IFC constrains the use of downgrading in systems [3].

Equation (1) shows constraints for *nonmalleable declassification* and *endorsement.* ℓ and ℓ' are the labels of data before and after downgrading ($\mapsto$), p is the label of the principal (user) perfomring downgrading. Here, ∇ means projecting confidentiality to integrity or projecting integrity to confidentiality. Subscript C indicates operation on the confidentiality dimension of the label while subscript I indicates operation on the integrity dimension. For example, consider a two-level lattice with two confidentiality levels, public (P) and secret (S), and two integrity levels, untrusted (U) and trusted (T). Then, $\nabla(P) = U$ and $\nabla(U) = P$; $(P, U) \sqcup_C (S, U) \Rightarrow (S, U)$ and $(P, U) \sqcup_I (P, T) \Rightarrow (P, U)$. The nonmalleable IFC constrains that data can only be declassified by a sufficiently trusted principal and data can only be endorsed when the principal can read it. As an example, label (S, U) cannot be declassified to (P, U) by an untrusted user ($I(p) = U$) because $S \not\sqsubseteq_C P \sqcup_C \nabla(U)$.

$$\begin{aligned} C(\ell) &\overset{p}{\mapsto} C(\ell') \text{ when } C(\ell) \sqsubseteq_C C(\ell') \sqcup_C \nabla(I(p)) \\ I(\ell) &\overset{p}{\mapsto} I(\ell') \text{ when } I(\ell) \sqsubseteq_I I(\ell') \sqcup_I \nabla(C(p)) \end{aligned} \tag{1}$$

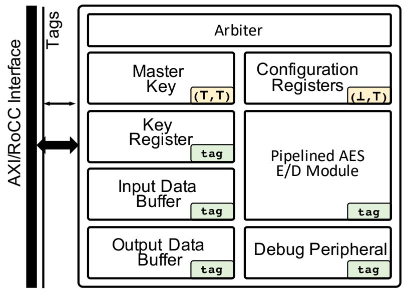

Figure 4: Overview of the proposed AES accelerator — Master key and configuration registers are associated with fixed security labels so only a certain users can access the contents; while datapath, data buffers and registers are associated with hardware tags to enable fine-grained resource sharing at runtime.

3 INFORMATION FLOW POLICIES IN CRYPTO ACCELERATORS

While there exist many types of security vulnerabilities, most exploitable vulnerabilities in practice result in insecure information flows that violate either confidentiality or integrity. In this section, we show how to prevent common vulnerabilities with information flow policies in a crypto accelerator.

3.1 Design Decisions and Vulnerabilities

To validate the effectiveness of the proposed approach in designing high-performance crypto accelerators, we choose a high-throughput pipelined architecture that processes one message block per clock cycle. Moreover, the accelerator is shared among multiple security levels in a fine granularity for better efficiency. Figure 4 shows the overview of the proposed AES accelerator.

Prior work has proposed many optimizations that improve the performance or the power-efficiency of an AES accelerator [22, 24]. However, such high-performance accelerator may have subtle security flaws unless designed carefully for security. First, pipelined architecture can introduce timing channels. For example, consider the case when two users, Alice and Eve, share the pipelined accelerator. The latency of Eve's encryption/decryption depend on the state of other pipeline stages, which may be processing Alice's data; a memory access for Alice may stall the pipeline and delay Eve's computation. The dependency can create a covert timing channel that leaks data from Alice to Eve [20]. Second, scratchpad memory holding user keys on-chip can introduce another security vulnerability. Figure 5 demonstrates a scratchpad with 64-bit cells, whose size is designed to be compatible with the host interface. Eve could leverage a buffer overflow error to override Alice's key stored in the adjacent cells if the accelerator does not properly check memory bounds. Finally, a debug peripheral is another common component in accelerators that can be misused. Prior work has demonstrated an attack that exploits an debug peripheral to compromise the secret key in an AES implementation [10].

3.2 Security Requirements and Information Flow Policies

Protecting the implementation from exploitable vulnerabilities (e.g., [5, 12]) is a primary objective of developing a secure AES accelerator. Table 1 summarizes the major security requirements and the corresponding information flow policies applied to enforce the requirements. With formulated information flow policies, IFC

Table 1: Main security requirements for a crypto accelerator and the equivalent information flow policies — For policy types, C and I represents confidentiality and integrity respectively. For restrictions, `key` $\nrightarrow$ `user` indicates any information flows from the key to the `user`'s resource is forbidden if the `user` doesn't have enough confidentiality. In security lattice, $\perp$ and $\top$ represent fully public and fully secret for confidentiality, while $\perp$ and $\top$ represent completely untrusted and completely trusted for integrity.

Security Assets	Security Requirements	Policy Type	Source (object and label)		Sink (object and label)		Restriction
Keys	1. A classified key cannot be read out by a less confidential user.	C	Key registers	$\ell(key)$	User registers/ outputs	$\ell(user)$	$key \nrightarrow user$ if $\ell(key) \not\sqsubseteq_C \ell(user)$
	2. A protected key cannot be modified by a less trusted user.	I	User inputs	$\ell(user)$	Key registers	$\ell(key)$	$user \nrightarrow key$ if $\ell(user) \not\sqsubseteq_I \ell(key)$
	3. A classified key cannot be used by a less trusted user.	C	Key registers	$\ell(key)$	Ciphertext output	$\perp$	$ciphertext \nrightarrow output$ if $\ell(key) \not\sqsubseteq_C \nabla(\ell(user))$
Plaintext	4. A low confidential user cannot read plaintext message from a higher confidential user.	C	Plaintext buffer	$\ell(pt)$	User registers/ outputs	$\ell(user)$	$plaintext \nrightarrow user$ if $\ell(pt) \not\sqsubseteq_C \ell(user)$
	5. A less trusted user cannot modify data beyond its authority.	I	User inputs	$\ell(user)$	Data buffers/ register	$\ell(data)$	$user \nrightarrow data$ if $\ell(user) \not\sqsubseteq_I \ell(data)$
Configs	6. Configuration registers can be read by any users, but only be modified by the supervisor.	I	User inputs	$\ell(user)$	Configuration registers	$\ell(cr)$	$*cr \rightarrow user$ as $\perp \sqsubseteq_C \ell(user)$ $*user \nrightarrow cr$ as $\ell(user) \not\sqsubseteq_I \top$ $*sup \rightarrow cr$ as $\ell(sup) \sqsubseteq_I \top$

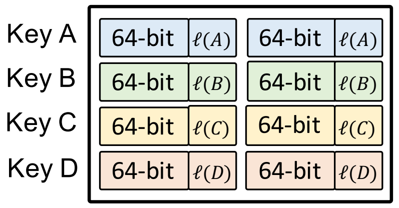

Figure 5: A key scratchpad memory with 512-bit capacity — Each cell has an associated tag to identify its security level. Any buffer overwrite or overread error will cause an information flow violation and will be prevented.

tools, such as ChiselFlow [7] and RTLIFT [1], can be leveraged to enforce these policies in the target implementation.

3.2.1 Preventing Information Disclosure within an AES Engine. The Encryption/Decryption (E/D) module is the core component in an AES accelerator. The E/D module protects plaintext data with the cryptographic key (encryption) or recovers ciphertext data into a clear message (decryption). Any disclosure of the key or the plaintext, caused by implementation errors or intentional backdoors, will undermine the security of the accelerator and even the entire system. At design time, a proper information flow policy should be formulated to rule out these information leakages. For a user program with label (c_u, i_u), its plaintext data should have a label of (c_u, i_u) and its secret key also carries a label of (c_k, i_u). Here, $c_k \not\sqsubseteq_C c_u$ and $c_k \sqsubseteq_C \nabla(i_u)$. By assigning a higher confidentiality label to the key, IFC can detect potential vulnerabilities that may leak the key. Figure 6 shows an example where the implementation contains a timing channel vulnerability [12]. In the implementation, the designer annotates the valid signal to be public $(\perp, i_u)$ to ensure that no secret leaks through that signal. On the other hand, the IFC tool infers that valid should have the label of (c_k, i_u) when its timing depends on the value of the secret key. As (c_k, i_u) cannot flow to $(\perp, i_u)$, this mismatch leads to an error that reflects the leakage

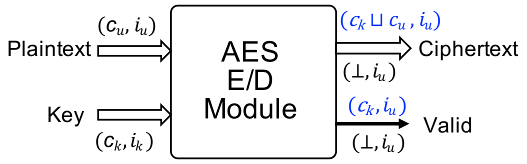

Figure 6: Information leakage leads to a label error in IFC — Blue labels are deduced from the implementation in IFC analysis, while the black labels are specified by designers. A disallowed mismatch means a potential implementation error.

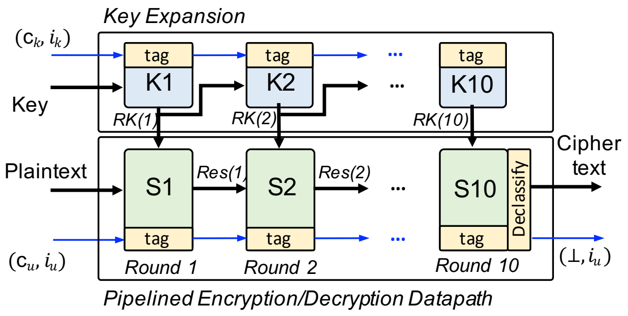

Figure 7: Each pipeline stage has a dedicated tag register to indicate its security level — Data and tag propagate through the pipeline stages, enabling fine-grained resource sharing at runtime.

from key to valid signal in the implementation. Other information leaks can be discovered in a similar fashion.

Besides the valid signal, Figure 6 also shows another label error at the ciphertext output. Because ciphertext contains information from both plaintext and key, the label of ciphertext should be $(c_k \sqcup_C c_u, i_u)$. On the other hand, the designer will consider the ciphertext as a public output. Consequently, the IFC tool raises an error if the ciphertext is released to a public channel. However, in practice, the release of the ciphertext should not compromise the confidentiality of the key and the plaintext. Therefore, we add the declassification statement to explicitly allow the ciphertext to be released at the output of the AES E/D module. As shown in Figure 7, in our AES engine, the declassification statement is placed at the end of the pipeline, only the output of the last encryption stage is declassified; outputting an intermediate result is still prevented by the IFC tool.

In a simple secure AES implementation, the E/D module is treated as a unit carrying one single security label, which implies that only one user can use the AES module at a time. To enable fine-grained sharing, we assign each pipeline stage with an independent security label; each security label is a dependent type so that the security level of each pipeline stage can change at the runtime. During the execution, the data and its label propagate through the pipeline together. In each clock cycle, a pipeline stage can change its security level and receive data from another security level. However, if there can be a mismatch between the data and the security tag in the implementation, the IFC tool will report a violation at design time.

3.2.2 Preventing Inappropriate Use of Cryptographic Keys. Even if an attacker cannot directly obtain the cryptographic key, an inappropriate use of the key can still break the security [19]. Therefore, the proposed AES accelerator prevents a less trusted user from using a high-confidential key for its encryption/decryption. Let us use a master key as an example to illustrate how an inappropriate use of the key is prevented in the proposed accelerator. The master key carries the label of $(\top, \top)$, as it is only accessible to the supervisor. Assume that a regular user (with a label of (c_u, i_u)) attempts to use the master key in encryption, the encrypted message would have a label of (c_k, i_u). Then, the AES engine tries to declassify the encrypted message after the final round in order to output to the public domain $(\bot)$. For the encryption with an authorized key, the declassification will be allowed as $c_k \sqsubseteq_C \nabla(i_u)$. However, for the encryption with the master key, $c_k == \top$ so $\top \not\sqsubseteq_C \nabla(i_u)$ and the declassification will be rejected under the nonmalleable IFC constraints. Only the supervisor has high enough integrity to declassify encryption with the master key.

3.2.3 Preventing Buffer Errors. A buffer error is another threat to crypto implementations. For example, if the accelerator does not check the length of a key when storing it into the scratchpad memory, a buffer overrun error may occur and overwrite other trusted keys. In order to prevent such errors, our implementation associates each memory block with a dedicated tag array as shown in Figure 5. Each memory cell has a corresponding tag in the tag array to indicate its security level at runtime. The accelerator checks the tag before reading data from or writing data to a memory location. If the tag checking reports a violation, the following write/read operation will be blocked. For example, consider a case where Eve sends a request to store her key into the scratchpad memory. The arbiter accepts the request and configures the cell 1 and 2 with label $\ell(Eve)$. Then, Eve writes her key to cell 1 and 2. However, if she attempts to overwrite cell 3 whose label is $\ell(Alice)$, the tag check will fail ($\ell(Eve) \not\sqsubseteq \ell(Alice)$) and the write will be blocked. The IFC analysis ensures at design time that the necessary runtime checks are implemented.

3.2.4 Access Control on Configuration Registers. As the accelerator is shared among multiple security levels, changes to configuration registers can affect multiple users. For security, only the supervisor should be able to modify the configuration registers. To enforce

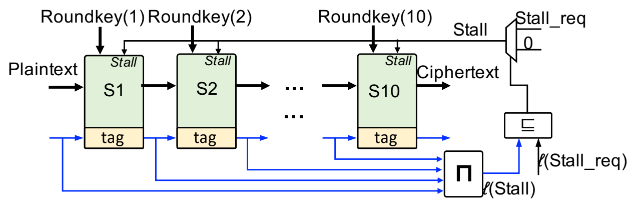

Figure 8: High confidential users can stall the pipeline when the pipeline does not contain data with low confidentiality.

this security policy, we label the configuration registers with $(\bot, \top)$, indicating that its values are public but should have the highest integrity. Any writes to the configuration registers from unprivileged users will cause an integrity violation.

3.2.5 Preventing Timing Channels in the Datapath Pipeline. In addition to information flows through signal values, timing channels can also be used to leak sensitive information. In the AES accelerator, we found that the fine-grained sharing of the datapath could introduce a timing channel, as mentioned in Section 3.1. To remove the timing channel, we only allow one security level to stall the pipeline when no pipeline stage has a lower confidentiality level. As shown in Figure 8, the stall logic determines the lowest confidentiality level across all pipeline stages by performing a meet operation $(\sqcup_C)$, which returns the security label with the lower confidentiality. When there is a request to stall the pipeline (`Stall_req`), the pipeline is stalled only when $C(\ell(Stall\_req)) \sqsubseteq_C C(\ell(Stall))$. The AES accelerator includes an extra buffer to hold outputs when the pipeline cannot be stalled when the receiver is not ready to read the outputs.

3.2.6 Discussion on downgrading. Information flow control enforces noninterference between security levels except for the violations explicitly allowed by downgrading operations. While inevitable for practical systems, the downgrading operations represent weakening of security and there is a question on what is the security assurance that we can obtain when there exists downgrading. For a traditional design without IFC analysis, potential vulnerabilities that can lead to information leakage may exist anywhere in a design. On the other hand, with IFC, potential information leakage can only occur through downgrading. Even though downgrading may be inserted incorrectly and lead to a vulnerability, it is much easier for human designers to carefully review the downgrading operations instead of inspecting the entire design for potential security vulnerabilities. Moreover, the nonmalleable IFC further constrains downgrading assignments to ensure that Only qualified principals can downgrade sensitive data, as illustrated in Section 3.2.2.

4 EVALUATION

To evaluate our protection scheme, we first built an AES accelerator baseline without information flow control. The baseline contains a deeply-pipelined datapath and a 512-bit key scratchpad. The pipeline receives one data block each cycle and completes the encryption of a data block in 30 cycles. The performance of the baseline accelerator is comparable to the performance of an existing high-throughput implementation [22]. We then extended the baseline with security tags and other information flow enforcement mechanisms, and verified the deisgn with a static IFC to remove

Table 2: Area and performance of the FPGA prototypes — LUTs: look-up tables, FFs: flip-flops, BRAMs: block RAMs.

	Baseline	Protected
LUTs	13,275	14,021 (+5.6%)
FFs	14,645	15,605 (+6.6%)
BRAMs	40	44 (+10.0%)
Frequency (MHz)	400	400 (+0.0%)

vulnerabilities. In the current implementation, we use 8-bit the security tags (4 bits for confidentiality and 4 bits for integrity), which is compatible with a state-of-the-art information flow enforced processor. To study the area and performance overhead, we implemented the prototype with Vivado 2017.1, targeting a Xilinx Virtex-7 FPGA device. The prototype implementation achieves a throughput of 51.2Gbps @ 400 MHz clock frequency.

To implement the secure (protected) AES accelerator, we changed around 70 lines of the baseline implementation in Chisel. The changes include annotating signals with security labels, building runtime checkers, and code transformations to remove vulnerabilities raised by the IFC analysis. All previously-mentioned vulnerabilities in the baseline are flagged by ChiselFlow and are addressed in the protected design. Table 2 shows the FPGA prototype results for both baseline and protected implementations. Our protection scheme incurs 5.6% and 6.6% overheads on the number of LUTs and FFs. The major BRAM overhead comes from two sources; one is the security tags stored with the on-chip data buffers, and the other is the extra buffer holding confidential outputs when the pipeline is stalled. Our protection does not have any impact on the critical path and the clock frequency.

5 RELATED WORK

Hardware implementation of cryptographic algorithms offers significantly higher performance and power-efficiency than its software equivalents. However, most of the hardware implementations only focus on performance, die area, and power consumption [22, 24], and do not address potential security concerns. Some efforts tried to protect the cryptographic accelerators from malicious attacks, but the resulting principles and techniques focus on specific vulnerabilities and do not offer systematic guarantees [4, 8].

In addition to the HDL-based approaches, hardware-level information flow control can be performed via dedicated tracking logic, e.g., gate-level information flow tracking (GLIFT) [21] and register-transfer-level information flow tracking (RTLIFT) [1]. Given a hardware design, GLIFT derives a dedicate information flow tracking logic and performs security analysis on it. Designers can either run static verification at design time or verify the security properties dynamically at runtime. GLIFT is also used to detect Trojans in hardware implementations [9]. The primary objective of this work is to formulate security requirements of a crypto accelerator as information flow policies. The formulated information flow policies can then be enforced using either security-typed HDLs or GLIFT.

6 CONCLUSIONS AND FUTURE WORK

Security vulnerabilities imposed by design decisions and other implementation flaws are threats to hardware cryptographic accelerators. In this paper, we propose to design and build cryptographic accelerators with hardware-level information flow control, which is capable of systematically checking a broad range of security requirements at design time. By expressing main security requirements as information flow policies, we can formally verify the security properties of the accelerator at design time with low overhead.

This work demonstrates that hardware-level information flow control is an effective mechanism in protecting high-performance crypto accelerators. Currently, the security requirements are manually expressed as information flow policies and enforced in the accelerator implementation. Automating the formulation procedure and integrating it into high-level design tools, such as security-related high-level synthesis [11, 15], will be promising research directions.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful comments. This research was supported in part by NSF award CNS1618275, Semiconductor Research Corporation under Task 2686.001, and DARPA SSITH Award HR0011-18-C-0014.

REFERENCES

[1] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner. Register Transfer Level Information Flow Tracking for Provably Secure Hardware Design. *Design, Automation, and Test in Europe (DATE)*, 2017.

[2] L. Bossuet, M. Grand, L. Gaspar, V. Fischer, and G. Gogniat. Architectures of Flexible Symmetric Key Crypto Engines-A Survey: From Hardware Coprocessor to Multi-Crypto-Processor System on Chip. *ACM Computing Surveys*, 2013.

[3] E. Cecchetti, A. C. Myers, and O. Arden. Nonmalleable Information Flow Control. *ACM SIGPLAN Conf. on Computer and Communications Security (CCS)*, 2017.

[4] H. Chan, P. Schaumont, and I. Verbauwhede. Process Isolation for Reconfigurable Hardware. *International Journal of Information Security*, 2013.

[5] N. V. Database. CVE-2014-0160 (Heartbleed). 2014.

[6] W. Diehl. Attack on AES Implementation Exploiting Publicly-visible Partial Result. *Technical Report, George Mason University*, 2017.

[7] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh. HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security. *ACM SIGPLAN Conf. on Computer and Communications Security (CCS)*, 2018.

[8] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting Private Keys against Memory Disclosure Attacks Using Hardware Transactional Memory. *IEEE Symp. on Security and Privacy (S&P)*, 2015.

[9] W. Hu, B. Mao, J. Oberg, and R. Kastner. Detecting Hardware Trojans with Gate-Level Information-F1ow Tracking. *Computer*, 2016.

[10] Y. Huang and P. Mishra. Trace Buffer Attack on The AES Cipher. *Journal of Hardware and Systems Security*, 2017.

[11] Z. Jiang, S. Dai, G. E. Suh, and Z. Zhang. High-Level Synthesis with Timing-Sensitive Information flow Enforcement. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[12] F. Koeune and J.-J. Quisquater. A Timing Attack Against Rijndael. 1999.

[13] X. Li, M. Tiwari, J. Oberg, V. Kashyap, F. Chong, T. Sherwood, and B. Hardekopf. Caisson: A Hardware Description Language for Secure Information Flow. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2011.

[14] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood. Border Control: Sandboxing Accelerators. *Int'l Symp. on Microarchitecture (MICRO)*, 2015.

[15] C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni. TaintHLS: High-Level Synthesis For Dynamic Information Flow Tracking. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.

[16] T. Reece and W. Robinson. Analysis of Data-Leak Hardware Trojans in AES Cryptographic Circuits. *Int'l Conf. on Technologies for Homeland Security*, 2013.

[17] J. Rott. Intel Advanced Encryption Standard Instructions (AES-NI). *Technical Report, Intel*, 2010.

[18] B. Schneier. Cryptographic Design Vulnerabilities. *Computer*, 1998.

[19] R. Stubbs. Classification of Cryptographic Keys. 2018.

[20] J. Szefer. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *IACR Cryptology ePrint Archive*, 2016.

[21] Tiwari, Mohit and Wassel, Hassan MG. and Mazloom, Bita and Mysore, Shashidhar and Chong, Frederic T. and Sherwood, Timothy. Complete Information Flow Tracking from the Gates Up. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[22] Y. Wang and Y. Ha. High Throughput and Resource Efficient AES Encryption/Decryption for SANs. *Int'l Symp. on Circuits and Systems (ISCAS)*, 2016.

[23] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A Hardware Design Language for Timing-Sensitive Information-Flow Security. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[24] X. Zhang and K. K. Parhi. High-Speed VLSI Architectures for the AES Algorithm. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 2004.