High-Level Synthesis with Timing-Sensitive Information Flow Enforcement

Zhenghong Jiang, Steve Dai, G. Edward Suh, Zhiru Zhang
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{jz763,hd273,gs272,zhiruz}@cornell.edu

ABSTRACT

Specialized hardware accelerators are being increasingly integrated into today's computer systems to achieve improved performance and energy efficiency. However, the resulting variety and complexity make it challenging to ensure the security of these accelerators. To mitigate complexity while guaranteeing security, we propose a high-level synthesis (HLS) infrastructure that incorporates static information flow analysis to enforce security policies on HLS-generated hardware accelerators. Our security-constrained HLS infrastructure is able to effectively identify both explicit and implicit information leakage. By detecting the security vulnerabilities at the behavioral level, our tool allows designers to address these vulnerabilities at an early stage of the design flow. We further propose a novel synthesis technique in HLS to eliminate timing channels in the generated accelerator. Our approach is able to remove timing channels in a verifiable manner while incurring lower performance overhead for high-security tasks on the accelerator.

1 INTRODUCTION

Hardware specialization promises increased performance and energy efficiency, but also comes with added hardware/software design complexity. Notably, the current practice of manually creating hardware accelerators requires significant development effort with the register-transfer-level (RTL) methodology. In order to meet the stringent performance and power requirements, designers have to wrestle with low-level HDL descriptions to apply a variety of optimization techniques such as pipelining, resource sharing, and clock/power gating, where functional, temporal, and spatial information must be jointly considered.

In addition to meeting performance and power consumption requirements, escalating design complexities also introduce new security challenges which have not yet been thoroughly explored. In particular, hardware accelerators are often shared among multiple security levels or multiple programs and can be exploited as a backdoor to break traditional software isolation boundaries. While modern system-on-chip (SoC) adopts system-level access control mechanisms to ensure that cores running non-secure applications cannot affect resources used by secure applications, these mechanisms cannot ensure secure operations of hardware accelerators internally. Because accelerators are often shared among multiple security levels, a malicious or buggy accelerator may either leak sensitive information among security levels (break confidentiality) or allow low-security applications to affect high-security computation (break integrity). Unfortunately, the increasing design complexity makes it especially challenging to ensure that an accelerator provides proper isolation among different security levels or processes.

To handle challenges arising from increasing design complexity and security threats, we develop ASSURE, a security-constrained high-level synthesis (HLS) framework that automatically synthesizes verifiably secure hardware accelerators from high-level behavioral descriptions under user-defined security constraints. ASSURE extends a state-of-the-art open-source HLS framework with information flow control mechanisms to synthesize secure hardware accelerators. The novelty of ASSURE stems from its ability to accept security policies and constraints from the designer as HLS constraints to guide the end-to-end C-to-hardware compilation process. Our specific contributions are as follows:

(1) To our best knowledge, this is the first HLS framework with information flow enforcement for synthesizing verifiably secure hardware accelerators. Our tool automatically detects and reports information flow violation in accelerator designs.
(2) We develop a security label inference pass to automatically infer security labels from inputs to outputs of a design, reducing the need for manual security annotation typical in HDL-based information flow enforcement.
(3) We propose a synthesis technique to decouple the internal and external timing behaviors of the accelerator to eliminate timing channels with lower performance overhead for high-security operations.
(4) We are able to demonstrate the existence of a designer-attributed vulnerability in a popular HLS-targeted crypto-accelerator and detect violation of information flow caused by common HLS optimization that lacks consideration for security.

The rest of the paper is structured as follows: Section 2 provides background materials on information flow security and its relationship to HLS. Section 3 describes our information flow constrained HLS framework. Section 4 presents experimental evaluations of our tool. Section 5 discusses related work in this area, followed by conclusions in Section 6.

2 BACKGROUND

In information flow security, each piece of data or resource is associated with a security level, and the security policy is expressed in terms of allowed and disallowed information flows between security levels. The information flow constraints are typically captured by a security lattice, whose ordering relation $\sqsubseteq$ determines which flows are allowed [9]. For example, if $L \sqsubseteq H$, information is permitted to flow from security level L to level H. The lattice can be more general to have multiple security levels, as shown in the three examples of Figure 1. For integrity, in the policy shown in Figure 1(a), data from the untrusted level (U) should not be able to affect the state and execution in the trusted level (T). For confidentiality, in the policy shown in Figure 1(b), sensitive information should not

ICCAD '18, November 5–8, 2018, San Diego, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240813

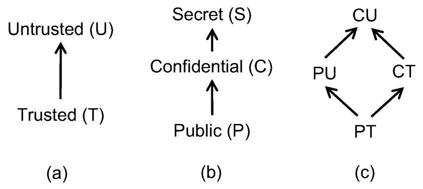


Figure 1: Example security lattices — (a) Two-level lattice for integrity; (b) Three-level lattice for confidentiality; (c) A diamond lattice represents both confidentiality and integrity.

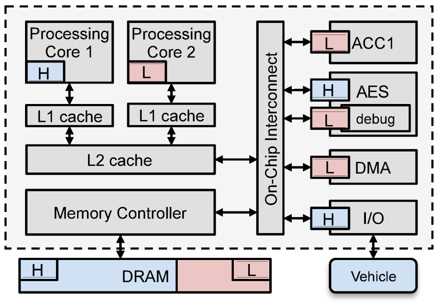


Figure 2: A heterogeneous SoC with ARM TrustZone.

leak from the confidential level (C) to the public level (P). *Noninterference* is a formal guarantee commonly imposed in information flow security that enforces confidentiality and/or integrity policies, which prevents any information flows from C to P (confidentiality) and/or from U to T (integrity) [12].

Figure 2 shows an example SoC with ARM TrustZone, which represents the state-of-the-art hardware architecture for today's SoC security [14]. In TrustZone, a system is partitioned into two security levels: secure (H) and non-secure (L). Based on this partition, software programs, memory space, accelerators, and hardware components such as I/O devices can be labeled as either H or L, as shown in the figure. TrustZone provides strong isolation between these two security levels by enforcing that non-secure programs or modules cannot access secure memory and devices.

However, even with TrustZone-like protection, system-level security can be compromised by malicious or buggy accelerators. Figure 3 illustrates two problematic designs of an AES crypto-engine that leaks the encryption key from the secure world to the non-secure world through the debug port. In the example, key is the only secret asset to be protected that carries a label H, while others are labeled as L. More concretely, the code in Figure 3(a) has an *explicit* information flow, as the key directly leaks through a non-secure debug port; Figure 3(b) shows another possible information leak via *implicit* information flow, wherein the key is encoded into the data sequence that leaks via the non-secure debug port. Unfortunately, conventional practice for hardware accelerator design typically does not account for explicit or implicit information flows, resulting in potential security vulnerabilities. For instance, recent research has discovered vulnerabilities in GPUs that leak sensitive information through the memory system [10, 22].

Accelerators may further leak information through timing interference among security levels, a vulnerability known as timing-based side channels (timing channels for short). Instead of directly observing a data value, a timing channel attack deduces secure information by measuring the timing of events at non-secure ports. For example, an attacker can measure the execution time of the accelerator by observing any changes of value at the non-secure ports. Figure 4(a) shows the code snippet for an RSA decryption

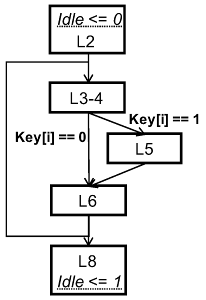
```
AES(int128 {H} Key,
    int128 {L} *Plain,
    int128 {L} *Cipher,
    int128 {L} *Debug)
{
  ...
  if (Mode == DEBUG)
    *Debug++ = Key;
  ...


}
```
(a)

```
AES(int128 {H} Key,
    int128 {L} *Plain,
    int128 {L} *Cipher,
    int128 {L} *Debug)
{
  ...
  if (Mode == DEBUG)
    for (i=0; i<128; ++i)
      if (Key[i])
        *Debug++ = i;
  ...
}
```
(b)

Figure 3: Information leakage in an insecure AES accelerator — (a) Explicit flow; (b) Implicit flow.

```
1 RSA(int1024 {H} Key,
      int1024 {L} *Cipher,
      int1024 {H} *Plain)
  {
2   int1024 R0 = 1, R1 = *Cipher;
3   for (i = 0; i < 64; ++i) {
4     if (Key[i] == 1)
5       R0 = R0 * R1 % Base;
6     R1 = R1 * R1 % Base;
7   }
8   *Plain = R0; //R0: {H}
9 }
```
(a)



(b)

Figure 4: Information leakage through a timing channel in RSA — (a) Code snippet. (b) Control flow graph.

accelerator, whose execution time may vary significantly depending on the value of the secret key (i.e., Key). Figure 4(b) further illustrates the control flow graph (CFG) of the RSA code, where the line numbers indicate the corresponding statements/operations in each basic block. Based on the CFG, the value of the secret key determines whether the branch in Line 4 (L4) is taken and whether the modular multiplication in Line 5 (L5) is executed. Besides, two additional interface statements, `Idle<=0` and `Idle<=1`, are automatically generated in synthesis to indicate the accelerator's execution status, as shown in the figure. This `Idle` interface has a label of L. Because the RSA accelerator takes longer to finish if L5 is executed, the execution time is correlated to the number of 1s in the secret key. Hence it is possible for an attacker to learn about the secret key by measuring the time interval between the deassertion and assertion of signal `Idle`. In this case, we declare that there is a timing flow from Key (H) to `Idle` (L) in Figure 4(a), wherein H and L in parenthesis represent signals' security label. Such timing channel attacks have been well documented in the literature [17, 36].

Addressing the aforementioned security challenges requires countermeasures that guarantee the absence of timing-sensitive information flows throughout the entire design flow. This involves detecting information flow in the design source as well as precisely controlling the timing of the design during synthesis. Security-typed hardware description languages (HDLs) partially address the problem by extending HDLs with security type annotation, which treats timing interference as implicit information flows [11, 40].

3 HLS WITH INFORMATION FLOW SECURITY

We argue that HLS, which automatically transforms an untimed high-level program into a timed RTL implementation, constitutes a more end-to-end approach to addressing the security challenges [5, 25]. In the front end, HLS can leverage compiler analysis to efficiently track the flow of information and identify any explicit and implicit information leakage, as the ones shown in Figure 3, at the behavioral level. In the back end, HLS can leverage scheduling and binding to impose precise control on the timing of the design to remove potential timing channels. Consequently, the timing behaviors of the public ports, such as `Idle` in Figure 4, are independent of secret values. Unlike RTL-based solutions which work with a timed hardware model that encapsulates both functionality and timing, HLS works with an untimed software model that describes only the functionality of the design. This gives the tool the freedom to independently consider functional flows (i.e., explicit and implicit information flow) and timing flows (i.e., timing channels), effectively reducing the complexity of synthesizing secure accelerators.

For simplicity, we will focus on a two-level confidentiality lattice to drive the discussions in the rest of the paper. We use H and L to indicate high confidentiality (secret) and low confidentiality (public) respectively. Our proposed approach can also handle integrity policies and more general multilevel lattice security models.

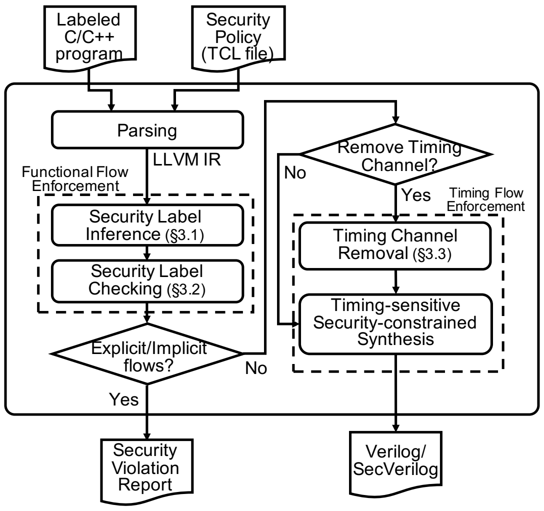
Figure 5 shows the overall design flow of our ASSURE framework, which accepts as inputs a high-level program in C/C++ and an information flow policy specified in Tcl. This Tcl file describes a security lattice that defines the available security levels and the permitted flows among different levels (as illustrated in Figure 1). In addition, the designer specifies the security level of each input and output of the program in the source code (as shown in Figure 3 and 4), based on the design specification. ASSURE performs functional flow enforcement followed by timing flow enforcement to ensure that security constraints are satisfied.

For functional flow enforcement, we perform automatic label inference to propagate security labels from inputs to outputs by tracking explicit and implicit information flow through control/data flow analysis (Section 3.1). ASSURE will examine both user-specified and inferred labels to determine if there exists any illegal or disallowed information flow (Section 3.2). In contrast to most existing RTL-based static information flow control (IFC) approaches where both external and internal variables must be manually annotated, ASSURE only requires users to specify security labels on the inputs and outputs of the top-level function. Other internal/intermediate variables are automatically labeled via inference. Besides label inference and checking at the behavioral level description, we propose a novel timing channel removal technique in ASSURE, which proactively eliminates timing flows to ensure timing-sensitive noninterference in the synthesized accelerator (Section 3.3). As an output, ASSURE generates secure hardware in either Verilog or SecVerilog [40]. In particular, SecVerilog allows us to verify that the timing channel is indeed removed in the automatically generated RTL design.

3.1 Security Label Inference

Figure 6 illustrates the label inference rules used in ASSURE, where e is an expression, and τ denotes its security label. Γ represents the environmental context of expression e, which is used to capture the implicit flows. Consider a binary operation $z = x \odot_b y$, where x has the label of τ_1 and y has the label of τ_2. With rule *T-OP*, variable z gets an inferred label of $\tau_1 \sqcup \tau_2$, which is the lowest level higher than or equal to both τ_1 and τ_2. Basically, $\tau_1 \sqcup \tau_2$ allows information flows from both x and y to z.

If we revisit the RSA example in Figure 4(a), the variable `R1` declared in Line 2 will receive an inferred label of L since input `Cipher` has a label of L. Consider the operations in Line 2 and Line 5, we infer that variable `R0` should carry a label of $L \sqcup L \Rightarrow L$, according to the binary operation inference rule in Figure 6. However, assignment to variable `R0` has a control dependency to the secret key in Line 4. Hence we infer that variable `R0` must be labeled as H to capture the implicit information flow (Rule `T-BRANCH`) from variable `Key` (H).



Labeled C/C++ program
Security Policy (TCL file)
Parsing
LLVM IR
Functional Flow Enforcement
Security Label Inference (§3.1)
Security Label Checking (§3.2)
Explicit/Implicit flows?
Yes
No
Remove Timing Channel?
No
Yes
Timing Flow Enforcement
Timing Channel Removal (§3.3)
Timing-sensitive Security-constrained Synthesis
Security Violation Report
Verilog/ SecVerilog

Figure 5: Overview of the ASSURE framework.

(T-CONST) $\quad n: \bot$

(T-OP) $\quad \dfrac{e:\tau}{\odot_u\ e:\tau} \quad \dfrac{e:\tau}{[e]:\tau} \quad \dfrac{e_1:\tau_1 \quad e_2:\tau_2}{e_1 \odot_b e_2 : \tau_1 \sqcup \tau_2}$

(T-ASSIGN) $\quad \dfrac{\Gamma:\ \tau_1 \quad e:\tau_2}{x := e \rightarrow x : \tau_1 \sqcup \tau_2}$

(T-BRANCH) $\quad \dfrac{\Gamma:\ \tau_1 \quad e:\tau_2}{br\ e,\ b_1,\ b_2 \rightarrow b_1, b_2 : \tau_1 \sqcup \tau_2}$

(T-BLOCK) $\quad \dfrac{b:\tau}{\forall i \in [1,n],\ \ \Gamma(c_i):\tau}$

(T-LOAD) $\quad \dfrac{\Gamma:\tau_1 \quad mem:\tau_2}{x := load\ [mem] \rightarrow x:\tau_1 \sqcup \tau_2}$

(T-STORE) $\quad \dfrac{\Gamma:\tau_1 \quad x:\tau_2}{store\ x, [mem] \rightarrow mem:\tau_1 \sqcup \tau_2}$

Figure 6: Security label inference rules.

Fine-grained resource sharing is an essential feature in hardware design for the consideration of cost. To support this feature, SecVerilog introduced dependent types in the type system, as shown in Figure 7. With only static labels, the multiplier belongs to either low-security user (L) or high-security user (H). Designers have to duplicate the multiplier if both users are going to use it. Nonetheless, by having dependent types, the shared multiplier can switch its security level depending on the value of ns signal that indicates the user of the resource. Similar to SecVerilog, our type system also supports dependent types to encourage fine-grained hardware sharing. Algorithm 1 demonstrates how label inference is extended to support dependent types in ASSURE. Here $\ell(x)$ and $\ell(y)$ represent the security label of variable x and y using dependent types.

The inference procedure may create a new dependent type automatically to enable additional resource sharing. Imagine that in the example in Figure 4, `Key` has a dependent type $\mathcal{LH}(ns_1)$ and `Cipher` has another dependent type $\mathcal{LH}(ns_2)$. This means that the

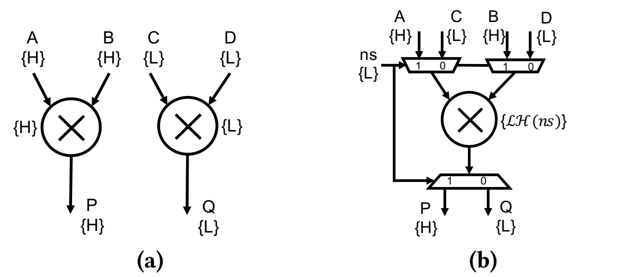


Figure 7: Fine-grained resource sharing with dependent type, $\mathcal{LH}(0) = L, \mathcal{LH}(1) = H$ — (a) Fixed label enforces every security level to have its dedicated multiplier. (b) Dependent type enables the sharing of multiplier across two levels.

Algorithm 1: Label inference for dependent labels

```
Input  : Security lattice $\mathcal{L}$, operation $z = x \odot_b y$
Output : Security Label of variable $z$
1  $f(ns_1) \leftarrow \ell(x)$ ;   // $\ell(x)$ is a dependent type on signal $ns_1$
2  $g(ns_2) \leftarrow \ell(y)$ ;   // $\ell(y)$ is a dependent type on signal $ns_2$
3  if $ns_1$ and $ns_2$ are the same variable then
4      $h(ns_1) \leftarrow \emptyset$ ;
5      for each value $i$ of $ns_1$ do
6          $h(i) = f(i) \sqcup g(i)$;
7      end
8  else
9      $h(\{ns_1, ns_2\}) \leftarrow \emptyset$ ; // $\ell(z)$ must be a label that depends
        on both $ns_1$ and $ns_2$
10     for each value $i$ of $ns_1$ do
11         for each value $j$ of $ns_2$ do
12             $k \leftarrow (i << w) + j$ ;        // $w$ is bitwidth of $ns_2$
13             $h(k) = f(i) \sqcup g(j)$;
14         end
15     end
16     compress $h$ ; // A const function becomes a static label
17     $\ell(z) \leftarrow h$
18 end
```

secret key and message may come from different users. Based on the inference algorithm, we will derive a new dependent type for R0 whose value is $dp(\{ns_1, ns_2\}) = \{(00 \rightarrow L), (01 \rightarrow H), (10 \rightarrow H), (11 \rightarrow H)\}$. $dp(\{ns_1, ns_2\})$ is the newly inferred dependent type, which indicates that R0 receives the L label only when both key and message come from the public level (L).

3.2 Security Label Checking

Once the label inference step is complete, we first compare the inferred label against the user-specified label at the outputs. If the inferred label has a higher security level than the specified one, an information flow violation is detected. For the AES example in Figure 3(a), the Debug port has an inferred label of H as it receives data from Key. However, since it is specified as a public port (L), a security violation will be flagged. Similar checks are performed on internal variables that have user-specified labels.

It is worth noting that noninterference may be too restrictive in many real-life scenarios, as sensitive data can be released to the public after protection, such as encryption. Hence many practical information flow control systems support downgrading which relaxes the security policies, so does ASSURE. Downgrading on confidential policies is called declassification, and downgrading on integrity policies is called endorsement [39]. Considering the RSA encryption example which has similar code structure to Figure 4, but instead, receives a plaintext message `Plain` (H) and returns the encrypted ciphertext `Cipher` (L). In the design, variable `R0` carries a label of H because it contains information from `Plain` (H). At the end of the function, the assignment of `R0` (H) to `Cipher` (L) would cause an explicit information flow violation. However, in practice, we know it is secure to release data in `R0` to public receivers after encryption, thus, such flow should be allowed. To address this issue, we can add a declassification command in the security policy specification to permit this specific information flow.

3.3 Timing Channel Removal

ASSURE aims to achieve timing-sensitive noninterference for the synthesized design, a property which ensures that secret values cannot be revealed by the timing of events observable at public ports. Recall the RSA example in Figure 4, where a public user can infer the value of the secret key by measuring the execution time of the accelerator. Because the value of each key bit influences the control flow and determines whether a (possibly) long-latency modular multiplication is invoked, the execution time of the accelerator is highly correlated with the value of the secret key. To eliminate this timing channel, a common approach is to enforce a constant (worst-case) completion time for the accelerator for all control paths by imposing additional operations or latency on relevant control branches. Such a constant-latency design can be implemented with code transformations [1, 28] or path-balanced scheduling [30].

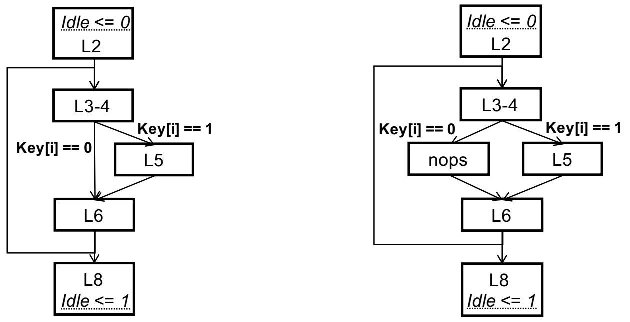
However, such conservative cookie-cutter approach results in unnecessary performance degradation for high-security operations when the accelerator is shared among multiple security levels. We illustrate the problem with an RSA accelerator time-multiplexed between a secret core and public core, as shown in Figure 8(a). The data ports, including `Key` and the `Plain`, are labeled as dependent types because they are only observable to the active accelerator user and their security levels consequently depend on who is actively using the accelerator. The automatically generated output `Idle` indicates the accelerator's occupation status and thus carries a label of L as it is seen by all users regardless of their security levels. This RSA design is vulnerable to timing attacks as the assertion time of the `Idle` signal depends on the number of 1s in the secret key.

It is possible to balance the control paths as shown in Figure 8(c) to enforce a constant assertion time for the `Idle` signal equal to the worst-case time among all control paths so that the timing of the signal provides no information about the secret key to the public core. However, this approach unnecessarily delays the assignment to the output data port indiscriminately by enforcing noninterference for users of all security levels. As a result, the performance of secret core tasks is affected along with public core tasks. While the public core should not be able to observe any variation in the execution time of the accelerator and infer the secret key, the secret core is allowed to access the secret key in the first place. As such, the secret core should be able to obtain the RSA decryption result faster because it does not require any enforcement of noninterference.

Because timing channels constitute a breach in security only if they are observable at public ports, it is necessary to only enforce constant timing behaviors at these public outputs, i.e. enforcing timing constraints on public I/O access operations. Based on this observation, we devise a new synthesis technique that decouples

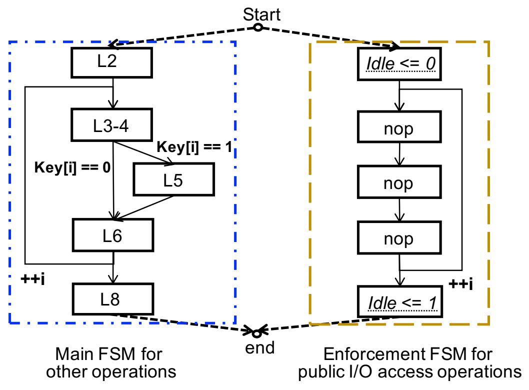
```
1 RSA(bool {L}         ns,
      int1024 {LH(ns)} Key,
      int1024 {L}      *Cipher,
      int1024 {LH(ns)} *Plain)
  {
2   int1024 R0 = 1, R1 = *Cipher;
3   for (i = 0; i < 64; ++i) {
4     if (Key[i] == 1)
5       R0 = R0 * R1 % Base;
6     R1 = R1 * R1 % Base;
7   }
8   *Plain = R0; //R0: LH(ns)
9 }
```

(a)



(b) (c)

Figure 8: Timing channel removal for RSA through path balancing — (a) C description of RSA with security labels, $\mathcal{LH}(0) = L, \mathcal{LH}(1) = H$. (b) FSM with unbalanced paths. (c) FSM after path balancing.



Start
L2
L3-4
Key[i] == 1
Key[i] == 0
L5
L6
++i
L8
Idle <= 0
nop
nop
nop
++i
Idle <= 1
end
Main FSM for other operations
Enforcement FSM for public I/O access operations

Figure 9: Timing channel removal by decoupling the execution of public I/O accesses from other operations.

the execution of public I/O accesses from other operations by using two separate FSMs. Instead of enforcing a path-balanced design with a single FSM (Figure 8(c)), our method creates another controller, called enforcement FSM, to manage public I/O accesses separately. This is shown in Figure 9, where the enforcement FSM only performs updates to the `Idle` signal and the loop induction variable i. ASSURE generates the enforcement FSM in a way that all secret-conditioned branches are balanced by applying the SDC-based scheduling [6, 7] with additional latency constraints. Other operations that do not affect timing of the public outputs are controlled by the main FSM.

Obviously, a naïve isolation approach would be insufficient when there exist data dependencies between the main and enforcement FSMs. Figure 10(a) shows an example where variable `x` is needed by operation `res[i]=f(x)` (L4) managed by the main FSM and a public write `debug[i]=x` (L7) controlled by the enforcement FSM. One solution is to duplicate the related computation `x=g(i)` on both sides (Figure 10(b)), although this approach is too expensive in general in terms of both area and power.

ASSURE avoids resource duplication by forwarding the result from the main datapath to the one controlled by the enforcement FSM. This is illustrated in Figure 10(c). However, a simple-minded data forwarding would not work unless the two concurrent FSMs are properly synchronized. Imagine the case where $\forall i$, `secret[i]==0`, operation `res=f(x)` at Line 6 takes two cycles and all other operations have a one-cycle latency. At cycle $T = 4$, the enforcement FSM is supposed to receive `x=g(0)` when it executes `debug[0]=x` at L7. Unfortunately, the main FSM would have already advanced to the second iteration of the loop where `x` becomes `g(1)`.

We resolve this issue by adding a hardware queue to buffer the data produced by the main datapath before they are used by the public I/O accesses (see Figure 10(c)). At runtime, when the queue is full, it stalls the execution of the main FSM through back pressure. Figure 11 shows the overall architectural scheme of our proposed approach that enables dynamic and complexity-effective timing channel removal. We note that the capacity of the hardware queue may influence the actual performance of the high-security operations. But regardless of the queue size, we always run the enforcement FSM at a speed independent of the secret information. Hence we are able to achieve timing noninterference from the perspective of public (low-security) users of the accelerator.

4 EXPERIMENTAL RESULTS

We develop ASSURE on top of LegUp, a state-of-the-art open-source HLS framework [4]. LegUp takes C/C++ description as input and outputs RTL implementation in Verilog. We implement the label inference pass and type system within the LLVM framework [20] to enable functional flow enforcement at the behavioral level. We further modify LegUp backend to enable timing flow enforcement with a security-constrained scheduling and binding process.

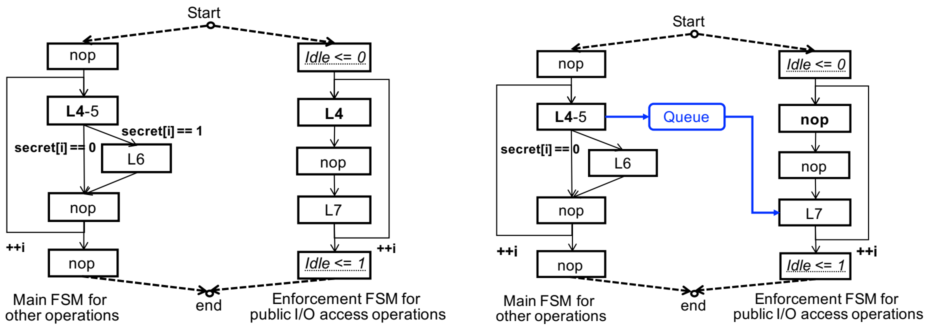
In this section, we evaluate ASSURE on its two contributing features: functional flow enforcement and timing flow enforcement. In Section 4.1, we evaluate functional flow enforcement on the popular HLS benchmark suite CHStone [13], where we are able to identify two types of security vulnerabilities among the designs. In Section 4.2, we perform experiments on timing flow enforcement with a set of five shared accelerators and compare the resulting area and performance of our timing channel removal technique with those of path-balanced scheduling [30]. For our experiments, we implement the generated RTL with Quartus 15.0 targeting an Intel Cyclone V FPGA.

4.1 Evaluation of Functional Flow Enforcement

Before the evaluation, we first partition the whole benchmarks into two categories: Cryptography and Arithmetic, based on the benchmark characteristics. For cryptographic benchmarks, we label the cryptographic key as H and other inputs as L, in order to verify whether the key is leaked in the design. For other benchmarks, all inputs are labeled with dependent types $\mathcal{LH}(ns)$, to model the application scenario where the accelerator is shared between secret

```
1 ACCEL(
    int {L}       ns,
    int {LH(ns)} secret[N],
    int {LH(ns)} res[N],
    int {L}       debug[N])
  {
2   int {L} x;
3   for (i=0; i<N; ++i) {
4     x = g(i);
5     if (secret[i])
6       res[i] = f(x);
7     debug[i] = x;
8   }
9 }
```

(a)



(b) (c)

Figure 10: Timing channel removal when dependencies exist between main and enforcement FSMs — (a) Variable x is needed on both sides. (b) Duplicating x=g(i). (c) Forwarding the data from main to the datapath controlled by enforcement FSM.

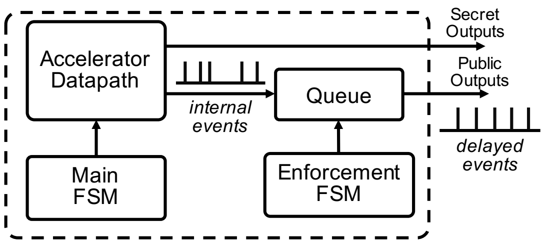


Figure 11: Overall architectural scheme for dynamic timing channel removal.

and public worlds. Here, *ns* is the signal for user identity, and $\mathcal{LH}$ is a dependent type that $\mathcal{LH}(0) = L$ and $\mathcal{LH}(1) = H$.

The third column in Table 1 demonstrates the security violations detected by ASSURE in the evaluation. The violations primarily come from two different vulnerabilities found in the benchmarks. The first vulnerability is discovered in the AES benchmark where the security violation is caused by a flaw of the user design. In the benchmark, the designer stores all plaintext, ciphertext and intermediate encryption results in the same array. The design style is helpful in improving the memory efficiency but introduces a security bug. Assume that an adversary has access to the plaintext/ciphertext, he/she can also access the intermediate encryption results that lie in the same memory. Hence the confidential information could be leaked. To address the issue, an AES design must put the intermediate results into a protected memory storage, with a label of H, where the attacker has no access. The same vulnerability also exists in the AES design in MachSuite, which is another commonly used HLS benchmark set [33].

The second vulnerability is caused by an HLS optimization in LegUp and is found among all benchmarks in the "Arithmetic" category. To achieve better resource efficiency, a synthesis optimization implemented in LegUp attempts to group multiple small arrays into a large physical memory block. Without taking security into consideration, two arrays with different security levels may be merged into the same physical memory. For such a case, it is difficult to protect the security without a complicated privilege access controller.

The above vulnerabilities reflect the fact that a reasonable security consideration is missing in existing hardware accelerator designs, which reinforces our motivation to enable a security-constrained synthesis to guide system designers (who may lack security expertise) in designing secure accelerators. In the above benchmarks, only explicit flows are detected, mainly because the

Table 1: Detecting functional flows in CHStone benchmarks — (1) "Y" indicates the corresponding information flow violation is detected while "-" indicates no violation under the corresponding category. (2) The unmodified version is the original CHStone benchmark suite, and the modified version is the one with insecure information flows manually introduced.

Benchmark	Category	Unmodified Version		Modified Version	
		Explicit	Implicit	Explicit	Implicit
AES	Crypto	Y	-	Y	Y
BLOWFISH		-	-	Y	Y
SHA		-	-	Y	Y
DFADD	Arithmetic	Y	-	Y	Y
DFMUL		Y	-	Y	Y
DFDIV		Y	-	Y	Y
DFSIN		Y	-	Y	Y
MIPS		Y	-	Y	Y
ADPCM		Y	-	Y	Y
JPEG		Y	-	Y	Y
MOTION		Y	-	Y	Y

CHStone benchmarks are arithmetic intensive. To test our framework against other types of information flows, we modify the benchmarks to introduce insecure information flows, both explicit and implicit, by intentionally mislabeling some internal variables, which are supposed to contain sensitive information, as L. The last column in Table 1 shows that ASSURE is able to detect all the introduced explicit and implicit flow violations. In addition, all designs generated by ASSURE are able to pass SecVerilog security verification.

4.2 Evaluation of Timing Flow Enforcement

The following five benchmarks with timing channel vulnerabilities are used to evaluate our timing flow enforcement.

String Comparison: String comparison is a typical operation used to check the message authentication code (MAC) in a crypto system/library. The conventional byte-by-byte implementation is exploitable to timing attacks because the comparison time is related to the number of matched bytes between the two strings, as the timing channel vulnerability in the Google Keyczar library[21].

Discrete Gaussian Sampling: Lattice-based cryptography contains a basic component, called discrete Gaussian sampling, whose task is finding the matching entity in a two-dimension table for a given input. Once the entity is found, the searching procedure returns its row index. Hence, the execution time depends on the value of the given input and the pre-computed Gaussian distribution table. Implementation with Knuth-Yao algorithm is an alternative for discrete Gaussian sampling on hardware platforms [34].

Table 2: Area and performance results with timing channel removal — ALMs: adaptive logic modules, FFs: flip-flops, BRAMs: block RAMs, DSPs: digital signal processing blocks.

Benchmark	ALMs		FFs		BRAMs		DSPs		Performance (Cycles)	
	Baseline	ASSURE	Baseline	ASSURE	Baseline	ASSURE	Baseline	ASSURE	Baseline	ASSURE
32-Byte String Comparison	303	364 (+20.1%)	340	400 (+17.6%)	4	5 (+25.0%)	0	0 (+0.0%)	135	71 (-47.4%)
Discrete Gaussian Sampling (Knuth-Yao)	285	304 (+6.7%)	421	442 (+5.0%)	5	5 (0.0%)	0	0 (+0.0%)	16452	5402 (-67.1%)
1024-bit Double-and-Add	1317	1413 (+7.3%)	1887	2017 (+6.9%)	18	18 (+0.0%)	0	0 (+0.0%)	45066	43546 (-34.9%)
64-bit Modulo Multiplication	1854	1887 (+1.8%)	4766	4834 (+1.4%)	2	3 (+50.0%)	0	0 (+0.0%)	8587	6312 (-36.0%)
64-bit RSA	2700	2806 (+3.9%)	5484	5716 (+4.2%)	16	16 (+0.0%)	12	12 (+0.0%)	8963	6527 (-37.4%)

Double-and-Add: Double-and-Add is an algorithm for multiplication of two large integer numbers, i.e. $k \cdot P$, which is commonly used in the Elliptic-curve cryptography. The conditionally executed operations in the algorithm result in a timing channel vulnerability.

Modulo-Multiplication: Modulo-Multiplication is an well-known operation in asymmetric cryptography domain, such as Diffie-Hellman key exchange. Its binary ladder implementation provides benefits on performance but is vulnerable to timing-channel attacks.

RSA Crypto: RSA is a public-key cryptographic protocol that is widely used for secure data transmission and digital signature. An implementation based on Square-and-Multiply algorithm makes it vulnerable to timing-channel attacks, because its completion time depends on the number of 1s in the secret key.

Table 2 presents the area and performance of accelerators generated with ASSURE's timing channel removal synthesis in comparison to the quality of results achieved by the path-balanced scheduling [30]. In each design, the total execution time depends on the exact value of the key (or the input value). We evaluate the performance of each benchmark with a set of randomly generated keys (or input values). The last column of the table reports the average performance achieved by secret users as well as the improvement in performance in comparison to that of path-balanced scheduling. Based on the table, ASSURE is able to achieve over 30% reduction in cycle count while incurring small area overhead. The area overhead is high at around 20% for the string comparison benchmark. This occurs because string comparison is a control dominated circuit whose datapath is only an equality check. Therefore, the controller duplicated in our timing channel removal scheme incurs relatively large overhead for this particular design.

5 RELATED WORK

Language-based static information flow control (IFC) is initially proposed to enforce security policies on software programs through type systems [35]. More recently, researchers have proposed extensions to hardware description languages that aim to enforce hardware-level information flow security [24, 40]. Cassion is among the first to use language-based solution to enable secure hardware design [24]. However, Caisson incurs significant area overhead because it does not permit fine-grained sharing of hardware resources among security levels. All registers must be duplicated for different security levels, and multiplexers are used to switch registers based on the security context. SecVerilog then introduces dependent types to address this limitation by allowing a signal to switch its security level depending on the value of another signal [40]. Dependent type enables fine-grained resource sharing and therefore allows the realization of a more efficient hardware design. These previous studies mainly focused on hardware designs at the register-transfer level (RTL). As high-level design methodology becomes more popular, IFC at RTL will be insufficient for handling security violations in the high-level design descriptions. By incorporating IFC into HLS to detect potential vulnerabilities at an early stage, ASSURE makes it possible to remove the vulnerabilities more easily and at lower costs. More importantly, ASSURE can automatically eliminate timing channels without any human intervention.

Sapper [23], GLIFT [38] and RTLIFT [2] are attempts to enforce hardware security with dynamic, instead of static, information flow control. Sapper uses static analysis at compile-time to automatically insert dynamic checks in the resulting hardware to enforce given information flow policies at runtime. GLIFT is initially proposed to perform dynamic information flow tracking at the gate-level, wherein all implicit flows manifest as explicit flows [38]. GLIFT benefits from its fine-grained precise flow tracking but also incurs significant area and timing overhead. Follow-up work demonstrates that GLIFT can also be used to check information flows at design time by running GLIFT logic through simulations or applying formal verification tools on it [15]. GLIFT logic is then removed for fabrication to prevent the additional area and performance overhead. However, formal or simulation-based verification may not be scale to large designs with complex GLIFT logic. Meanwhile, either verified statically or dynamically, generating precise GLIFT logic is NP-complete.[1] RTLIFT is similar to GLIFT but tries to reduce the complexity of the information tracking logic by raising the design abstraction to RTL.

Besides enforcement in hardware, IFC on software programs is a well-studied field. One possible alternative for generating secure RTL is applying existing IFC methods on C to generate a secure C and then feeding the secure C to HLS. However, contemporary HLS tools are driven by performance and power, and are yet to offer any security features [5, 25, 41]. While emerging HLS optimizations continue to improve the performance of generated hardware [8, 16, 26, 27], these optimizations may break the protections enforced at the software-level and result in security flaws in the generated hardware, as demonstrated by the AES example in Section 4.1.

Tackling hardware security in HLS is an emerging research field in recent years. For instance, HLS techniques are proposed to counter hardware Trojans in third-party intellectual properties during allocation, binding, and scheduling [31, 32]. Other recent efforts [18, 19] attempt to improve the resistance to malicious Trojan insertion or power-based side-channel attacks. To the best of our knowledge, no prior HLS work provides the comprehensive timing-sensitive information enforcement proposed in ASSURE.

In addition to application-specific protections, system-level efforts exist to provide security to SoCs with third-party accelerators, where the primary goal is to prevent host processors from being contaminated by untrusted accelerators [29, 37]. These techniques treat the accelerators as blackboxes and build protections on the host/accelerator interface to prevent illegal data communications. However, these efforts do not address security vulnerabilities within accelerators.

[1]Compared to the version of this paper released by IEEE/ACM, we have updated the discussion on GLIFT with a clarification on how it is applied in a static verification scenario.

6 CONCLUSIONS AND FUTURE WORK

To meet the increasing security needs in specialized hardware accelerators, we propose ASSURE, a novel HLS framework with timing-sensitive information flow enforcement. Using behavioral-level analysis, ASSURE can capture both explicit and implicit information flow violations that exist in the design and help designer discover and address potential vulnerabilities at an early design stage. In addition to functional flows, ASSURE can also eliminate information leakage caused by timing channels. In contrast to conventional timing channel removal approaches, ASSURE provides a novel synthesis technique that removes timing channels while incurring lower performance overhead for high-security operations executing on a shared accelerator.

Currently ASSURE's timing channel removal technique does not support pipeline synthesis [3, 42], which limits its application on security-critical applications with high-throughput demands. Enabling effective timing channel removal for pipelined design is a future extension of our research. While static information flow analysis used in ASSURE has the advantage on soundness, it may be too conservative for certain applications. It may be desirable to support dynamic information flow control (e.g. GLIFT [38]) to improve permissiveness and preciseness. Using a hybrid information flow control mechanism in HLS to balance the tradeoff between soundness and preciseness is an interesting future direction.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful comments. This research was supported in part by NSF award CNS-1618275, Semiconductor Research Corporation under Task 2686.001, and DARPA SSITH Award HR0011-18-C-0014.

REFERENCES

[1] Johan Agat. Transforming Out Timing Leaks. *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*, 2000.

[2] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. Register Transfer Level Information Flow Tracking for Provably Secure Hardware Design. *Design, Automation, and Test in Europe (DATE)*, 2017.

[3] Andrew Canis, Stephen D. Brown, and Jason H. Anderson. Modulo SDC Scheduling with Recurrence Minimization in High-Level Synthesis. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2014.

[4] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level Synthesis for FPGA-Based Processor/Accelerator Systems. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2011.

[5] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2011.

[6] Jason Cong and Zhiru Zhang. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. *Design Automation Conf. (DAC)*, 2006.

[7] Steve Dai, Gai Liu, and Zhiru Zhang. A Scalable Approach to Exact Resource-Constrained Scheduling Based on a Joint SDC and SAT Formulation. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[8] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[9] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 1976.

[10] Roberto Di Pietro, Flavio Lombardi, and Antonio Villani. CUDA Leaks: Information Leakage in GPU Architectures. *arXiv preprint arXiv:1305.7383*, 2013.

[11] Andrew Ferraiuolo, Weizhe Hua, Andrew C. Myers, and G. Edward Suh. Secure Information Flow Verification with Mutable Dependent Types. *Design Automation Conf. (DAC)*, 2017.

[12] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. *IEEE Symp. on Security and Privacy (SP)*, 1982.

[13] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. CHStone: A Benchmark Program Suite for Practical C-Based High-Level Synthesis. *Int'l Symp. on Circuits and Systems (ISCAS)*, 2008.

[14] ARM Holdings. ARM Security Technology Building a Secure System using a TrustZone Technology, 2009.

[15] Wei Hu, Dejun Mu, Jason Oberg, Baolei Mao, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Gate-level information flow tracking for security lattices. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 2014.

[16] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically Scheduled High-level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[17] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. *Int'l Cryptology Conf. (CRYPTO)*, 1996.

[18] S.T. Choden Konigsmark, Deming Chen, and Martin D.F. Wong. Information Dispersion for Trojan Defense Through High-Level Synthesis. *Design Automation Conf. (DAC)*, 2016.

[19] S.T. Choden Konigsmark, Deming Chen, and Martin D.F. Wong. High-Level Synthesis for Side-Channel Defense. *Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, 2017.

[20] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. *Proceedings of the Int'l Symp. on Code generation and optimization: feedback-directed and runtime optimization*, 2004.

[21] Nate Lawson. Timing Attack in Google Keyczar Library, 2009.

[22] Sangho Lee, Youngsok Kim, Jangwoo Kim, and Jong Kim. Stealing Webpages Rendered on Your Browser by Exploiting GPU Vulnerabilities. *IEEE Symp. on Security and Privacy (SP)*, 2014.

[23] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A Language for Hardware-Level Security Policy Enforcement. *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[24] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A Hardware Description Language for Secure Information Flow. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2011.

[25] Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh N. Do, and Deming Chen. High-Level Synthesis: Productivity, Performance, and Software Constraints. *Journal of Electrical and Computer Engineering*, 2012.

[26] Gai Liu, Mingxing Tan, Steve Dai, Ritchie Zhao, and Zhiru Zhang. Architecture and Synthesis for Area-Efficient Pipelining of Irregular Loop Nests. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2017.

[27] Junyi Liu, Samuel Bayliss, and George A. Constantinides. Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2015.

[28] Heiko Mantel and Artem Starostin. Transforming Out Timing Leaks, More or Less. *European Symp. on Research in Computer Security (ESORICS)*, 2015.

[29] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. Border Control: Sandboxing Accelerators. *Int'l Symp. on Microarchitecture (MICRO)*, 2015.

[30] Steffen Peter and Tony Givargis. Towards A Timing Attack Aware High-Level Synthesis of Integrated Circuits. *Int'l Conf. on Computer Design (ICCD)*, 2016.

[31] Jeyavijayan Rajendran, Huan Zhang, Ozgur Sinanoglu, and Ramesh Karri. High-Level Synthesis for Security and Trust. *Int'l On-Line Testing Symposium (IOLTS)*, 2013.

[32] Jeyavijayan JV. Rajendran, Ozgur Sinanoglu, and Ramesh Karri. Building Trustworthy Systems using Untrusted Components: A High-Level Synthesis Approach. *IEEE Trans. on Very Large-Scale Integration Systems (TVLSI)*, 2016.

[33] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Machsuite: Benchmarks for Accelerator Design and Customized Architectures. *Int'l Symp. on Workload Characterization (IISWC)*, 2014.

[34] Sujoy Sinha Roy, Frederik Vercauteren, and Ingrid Verbauwhede. High Precision Discrete Gaussian Sampling on FPGAs. *Int'l Conf. on Selected Areas in Cryptography (SAC)*, 2013.

[35] Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications (J-SAC)*, 2003.

[36] Werner Schindler. A Timing Attack Against RSA with the Chinese Remainder Theorem. *Int'l Workshop on Cryptographic Hardware and Embedded System (CHES)*, 2000.

[37] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-Flow Isolation. *IEEE Symp. on Security and Privacy (SP)*, 2016.

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

[39] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure Program Partitioning. *ACM Transactions on Computer Systems (TOCS)*, 2002.

[40] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew 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.

[41] Zhiru Zhang, Deming Chen, Steve Dai, and Keith Campbell. High-Level Synthesis for Low-Power Design. *IPSJ Transactions on System LSI Design Methodology*, 2015.

[42] Zhiru Zhang and Bin Liu. SDC-Based Modulo Scheduling for Pipeline Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2013.