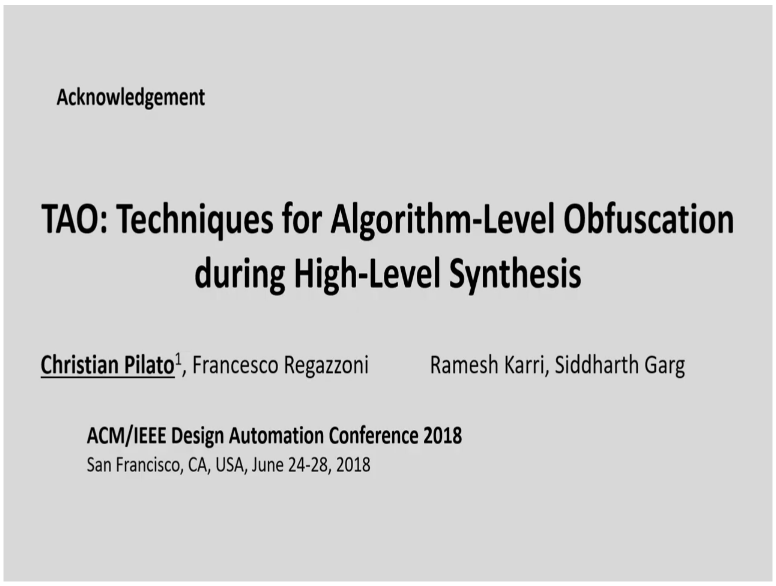
C-Based VLSI Design

Dr. Chandan Karfa
Department of Computer Science and Engineering
Indian Institute of Technology, Guwahati

Module - 11
Securing Design with High level Synthesis
Lecture - 37
Securing Design through HLS

So, welcome again to my course. In today's class I am going to see discuss about High Level Synthesis for Security.

(Refer Slide Time: 01:00)



Acknowledgement

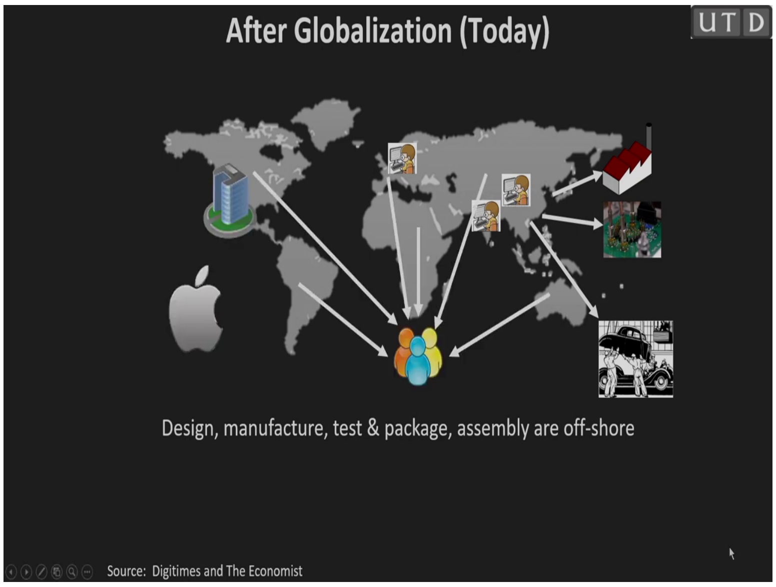
TAO: Techniques for Algorithm-Level Obfuscation during High-Level Synthesis

Christian Pilato[1], Francesco Regazzoni   Ramesh Karri, Siddharth Garg

ACM/IEEE Design Automation Conference 2018
San Francisco, CA, USA, June 24-28, 2018

So, again I want to acknowledge the authors of this particular paper via from where I have adapted the slides.

(Refer Slide Time: 01:05)



After Globalization (Today)

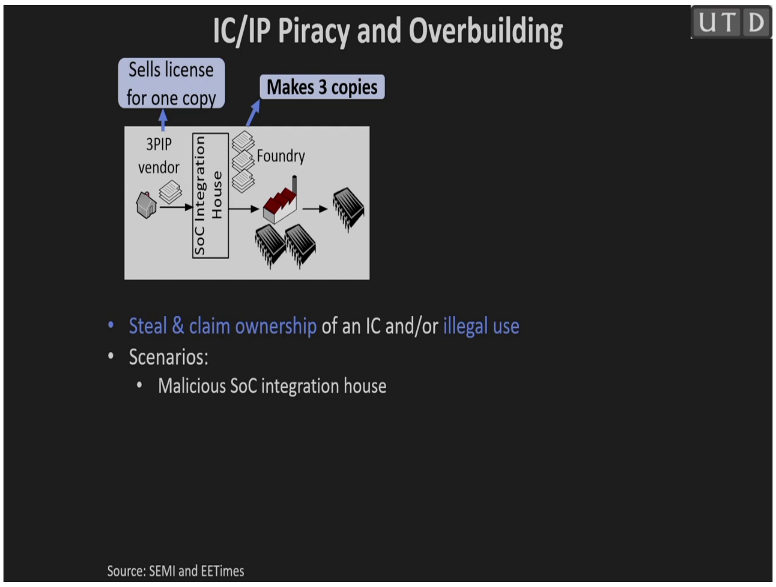
Design, manufacture, test & package, assembly are off-shore

Source: Digitimes and The Economist

So, as we have discussed in the previous class that because of the globalization of the IC design flow and we have to give our IP to somebody, to the foundry where the actual fabrication is going to happen.

And then we have also discussed that because of this foundry can be malicious or there will be some employee in the foundries which is malicious, who can steel my IP who can reverse engineer the IP and then we can generate a netlist and as a result we can have this IP piracy and overbuilding.

(Refer Slide Time: 01:34)



IC/IP Piracy and Overbuilding

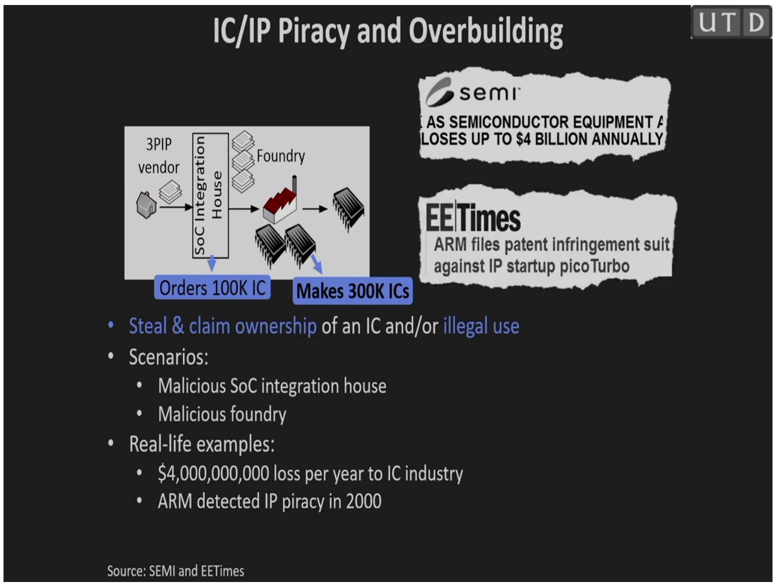
Sells license for one copy
Makes 3 copies
3PIP vendor
SoC Integration House
Foundry

- Steal & claim ownership of an IC and/or illegal use
- Scenarios:
  - Malicious SoC integration house

Source: SEMI and EETimes

So, there are two things piracy means basically you reverse engineer the netlist and you get netlist and from there you can actually do your own thing. So that means, you already get the intellectual property my intellectual property and you can actually pirate this.

(Refer Slide Time: 01:53)



IC/IP Piracy and Overbuilding

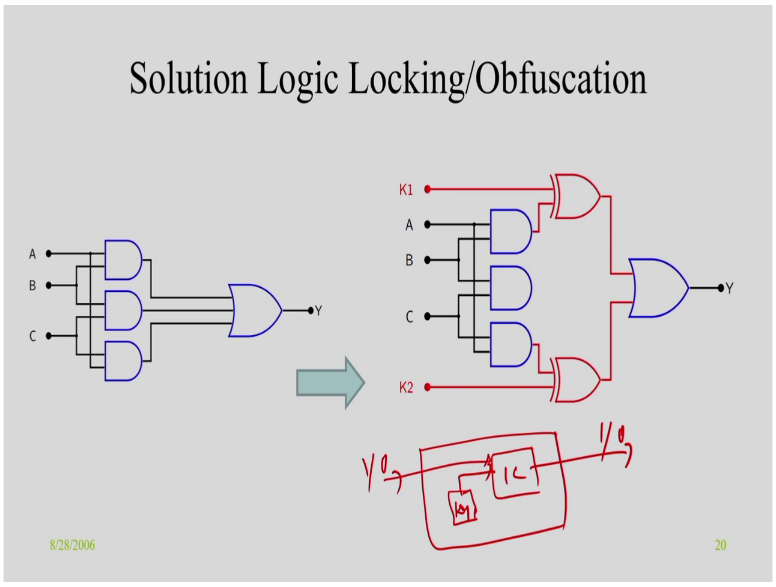
UTD
3PIP vendor
SoC Integration House
Foundry
Orders 100K IC
Makes 300K ICs
semi
AS SEMICONDUCTOR EQUIPMENT
LOSES UP TO $4 BILLION ANNUALLY
EETimes
ARM files patent infringement suit against IP startup picoTurbo

- Steal & claim ownership of an IC and/or illegal use
- Scenarios:
  - Malicious SoC integration house
  - Malicious foundry
- Real-life examples:
  - $4,000,000,000 loss per year to IC industry
  - ARM detected IP piracy in 2000

Source: SEMI and EETimes

So, and then over building is something is very easy because if I order say 100 K IC to you and you can actually build 300 K and you can sell 200 K in the black market in half price because you do not have to invest in the development of the whole IP and as a result the actual company will be in loss. So, these are the two primary concern because of the globalization of the IC design flow.

(Refer Slide Time: 02:15)



Solution Logic Locking/Obfuscation

And then we have also discussed that the logic locking which is very simple, but very powerful technique which actually can stop this IP piracy and over building. And the basic idea of logic locking I have already explained in the previous class that its basically the original circuit you add some additional key.

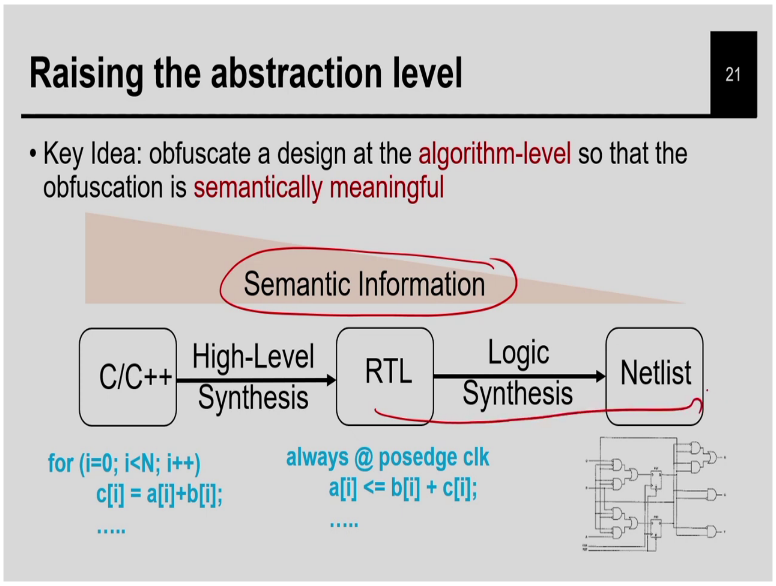
And you add some extra circuitry in your design such a result this circuit is now locked and this circuit only work for one possible value of this case. So, that is very important And if your key size is say 100 then there are 2 to the power 100 possible keys and only it will work for one possible value of that and checking this locked circuit for 2 to the power 100 is impossible as a result unless this key is available to you cannot use this circuit.

So, that was the key of this logic locking and what I am going to do in this fabrication lab I am going to give this circuit, but I will not give you the correct key. So, as a result if you just even if you do the reverse engineer, you get the netlist you produce so many IPs it will not work in practice, because this is that particular IC does not have the correct key value.

But for the original design house who can do that he actually just integrates the IC it will get the IC from the fabrication lab and then it will just put a temper from memory where he will put the correct key and it will give the input and this is the actual IC he will sell in the market.

And so, the input-output facility will not change and this key is already inside the circuit. So, the customer does not know what is the correct key value because he does not need to know. But it will just work seamlessly because the input output characteristic does not change, but only the original IP will work not the IC which is sold by the fabrication lab.So, that is something is the key idea of this and also, we have discussed that logic locking is a something where lot of work is happening in most of the work is actually on the gate level circuit.

(Refer Slide Time: 04:13)



Raising the abstraction level

21

- Key Idea: obfuscate a design at the algorithm-level so that the obfuscation is semantically meaningful

.So, what is our discussion topic today can we do something during high level synthesis.

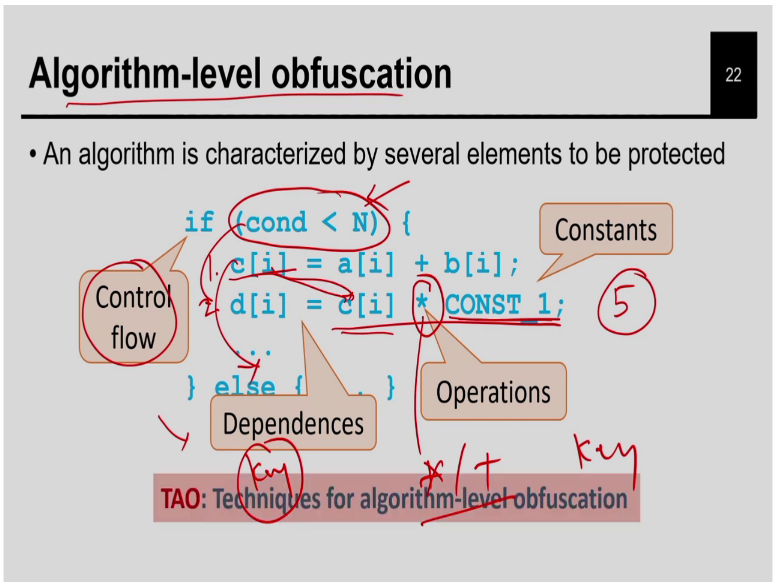
So, that the RTL generated by the high-level synthesis is already locked, so that means I have given an input output behavior I will generate the RTL which has the same input output. In addition to that it will have another key and then, that particular key is something will be given by the high-level synthesis will say that I mean this circuit is locked to it say 200-bit keys and the correct value of the keys is this.

So, if this high-level synthesis will do that means my locking thing is automated during high level synthesis and I do not have to do anything. So, that is really interesting. So, I am going to see how can I generate a locked RTL through high level synthesis.

And doing this in the high-level synthesis is something has lot of advantages, first is the code size is less and the most important thing is the semantic information. That is because you see here you have this C code and this C code has the if else, you have conditional branch and so many information, I can utilize that information during locking.

And then because I once I lock, I know what is the correct value of the key I can just produce the correct key as also the input of high-level synthesis. So, that is the idea and once you have the RTL it will just go through the complete flow and the layout will have locked layout, so that is the idea.

(Refer Slide Time: 05:38)



Algorithm-level obfuscation

- An algorithm is characterized by several elements to be protected

```
if (cond < N) {
  c[i] = a[i] + b[i];
  d[i] = c[i] * CONST_1;
  ...
} else { ... }
```

Control flow
Constants
Dependences
Operations
5
*/+
key
key

TAO: Techniques for algorithm-level obfuscation

So, this is what is called algorithm level obfuscation, because I am going to do it specifically during high level synthesis, but I am going to do the input C code. So, we have the algorithm and I am going to do that. So, what are the things we can lock so the let us try to understand. So, the first thing you can do that there are some constants and a constant has a fixed value say 5.

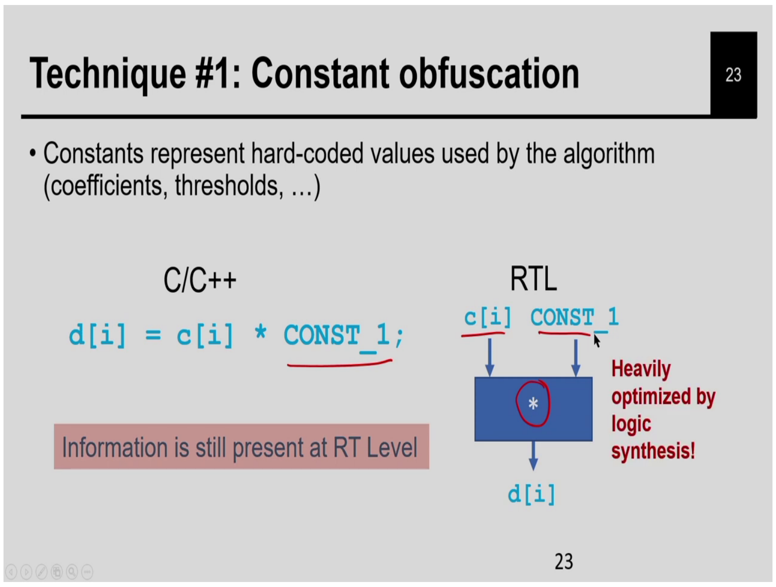
So, I can lock this value, so that the other employee who does not know what is the correct value of the key or of this constant. What else I can lock; I can lock this control behaviour. So, if this condition is true, I am going to do this, else I am going to do this, because if I do not know what is the condition without the correct key I do not know whether I have to take the if branch or the else branch.

So, then I can actually lock the control flow. I have several operations happening. So, I can lock these operations as well. So that means, the user will not understand whether it is a star or multiplication operation, or plus or addition operation until you have a key correct key. So, with a key I can lock these operations, and the fourth one which I can lock is the dependency.

So, here is this operation actually define c of i, and that is getting used here. But what I can do is, I can do something, so that there are I just add some key such that unless you know the correct key you do not know whether the dependency is between these two operations or these two operations.

So, I can lock this information, this is the semantic information in the algorithmic input behaviour which is the input C code and this I can do during high level synthesis. Because this is all internal information is available with me and then I can generate an RTL which has all these things locked. So, this is the overall idea of this and it is I am going to discuss with examples about all these four techniques, what we can do.

(Refer Slide Time: 07:27)



Technique #1: Constant obfuscation

23

- Constants represent hard-coded values used by the algorithm (coefficients, thresholds, ...)

C/C++

```
d[i] = c[i] * CONST_1;
```

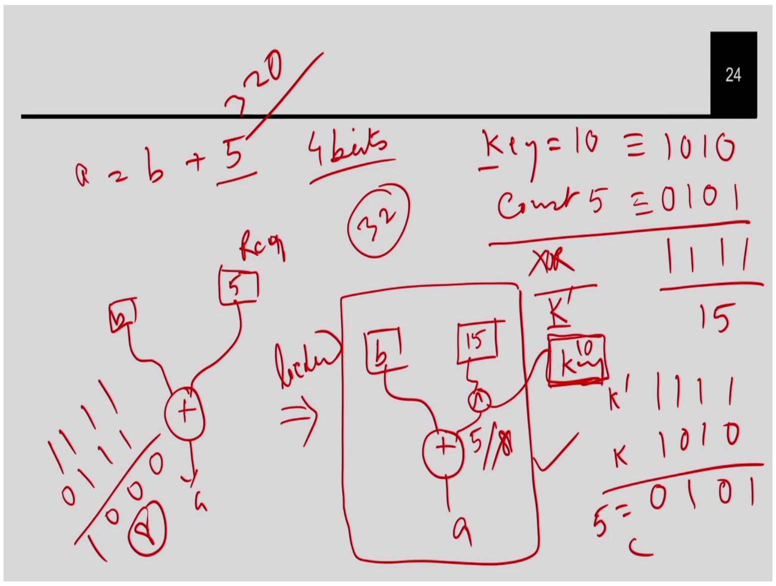
Information is still present at RT Level

RTL
c[i]
CONST_1
*
d[i]
Heavily optimized by logic synthesis!

23

So, first take this constant obfuscation. So, as I say there are some constants in your behaviour. So, suppose there is an operation like this you have a multiplier here and you are just doing c of i multiplied by some constant. So, what I am going to take I am going to take an example.

(Refer Slide Time: 07:44)



Say suppose you have a = b plus 5, and let us say this constant is of 4 bits, in hardware you can do that because there is a possible way to do that. So, what I am going to do now let us say my I have taken a key which is say 10, my correct key is 10 which is equivalent to 1 0 1 0, and my actual constant is 5 which is equivalent to 0 1 0 1, and then I am going to do XOR.

So, if you do a XOR it will be 1 1 1 1. So now, this is my locked key. So, this is my K', what I am going to do is, I am going to put in my code like this. So, you see earlier it was kind of I have an adder and one of the inputs is a, earlier in your design actually 5 was there. So, if there is a register in which I have stored 5 and I will do this.

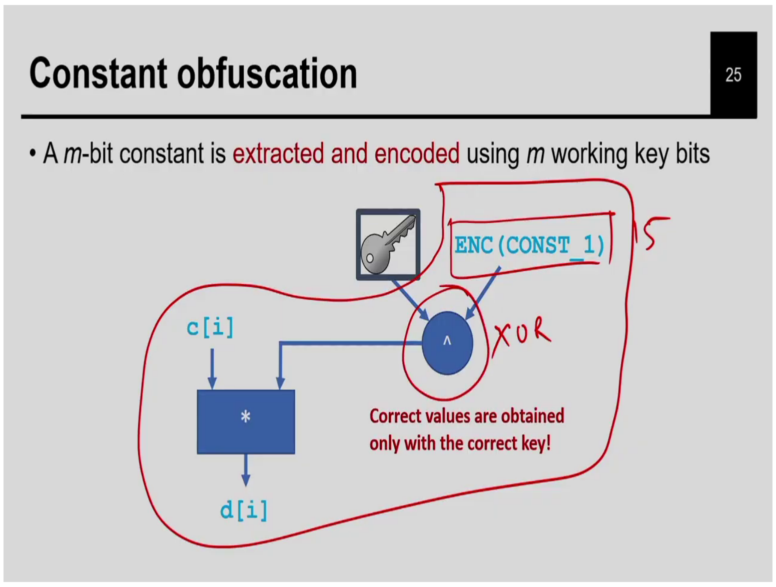
So, what I am going to do now is I am going to store in the design. So, this is my locked design. So, I have an adder I have the one input is a. What I am going to do I am not going to store this 5 and this is basically 15. So in the register I am going to store 15 and then I am going to put a XOR here and I am going to do this and this is b.

So, here I am going to do with the actual key. So, this is my key which is 10 and this is my locked circuit. So, this is my circuit and this key will come from outside. So, he does not know the key. So, unless you give 10, you do not know this is actually 5. So now, you see if you have 1 1 1 1 and you have the key as 10. So, this is my K' this is my K which is 1 0 1 0 then it will become 1 0 1 0 which is 5.

So, the idea is that if you do a XOR of K and K' you will get the constant C that is what is happening here. So, this will be my locked circuit and this 10 is not available to the design because it will come from outside, I have already explained that. So that means, now this locked circuit only work if you give the correct key 10. If you give say some input say 5 then 15 is 1 1 1 1 and 5 is say you guess the key is say 7.

So, 1 1 1 0 you will get 0 0 0 1 which is 8, but then 8 will come here, but that is not the correct key. So, actual constant was 5. So, this circuit will only work if you give the correct key 10 that was the idea of this constant locking.

(Refer Slide Time: 11:07)



Constant obfuscation

25

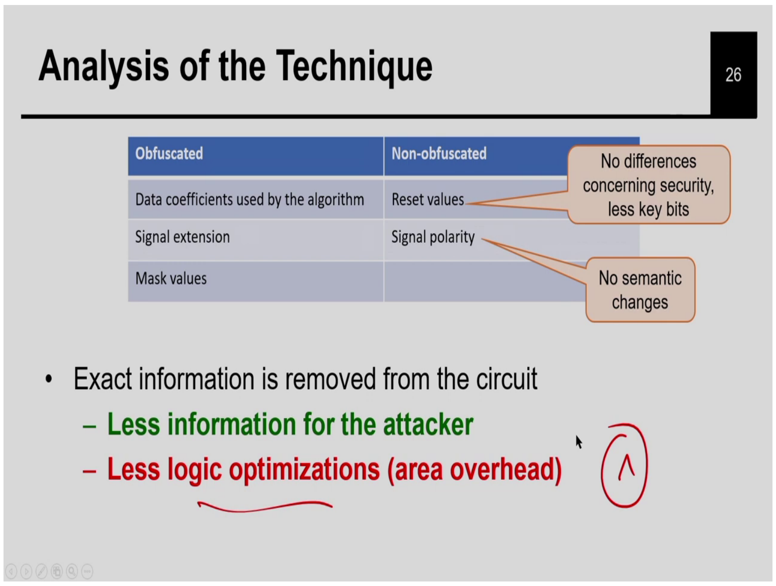
- A *m*-bit constant is extracted and encoded using *m* working key bits

. So, this is what exactly I am saying that this is a my encoded constant one which is my 15 in my example in the previous case and this is my XOR gate and this is my actual circuit and this key will come from outside and this key is only available for the genuine customer not to the pirated copies.

So, as a result I can actually lock all the constant using this technique and it actually is very powerful because I just have taken an example of 4 bit key, but if your constant is of 32 bits. You can just have a 32-bit key generated for a single constant and in general, in a C program the constants are actually of 32 bits, because this is an integer constant.

Then actually if we just lock 10 such constant you will get a 320-bit keys and it is huge. So, you can have just possible value of 2 to the power 320 just on locking 10 constant and which is simply easy. For a given program we usually have many such constant in the program and just locking those constants using say 32 bits you can generate a huge key size.

(Refer Slide Time: 12:20)



Analysis of the Technique

26

Obfuscated	Non-obfuscated
Data coefficients used by the algorithm	Reset values
Signal extension	Signal polarity
Mask values	

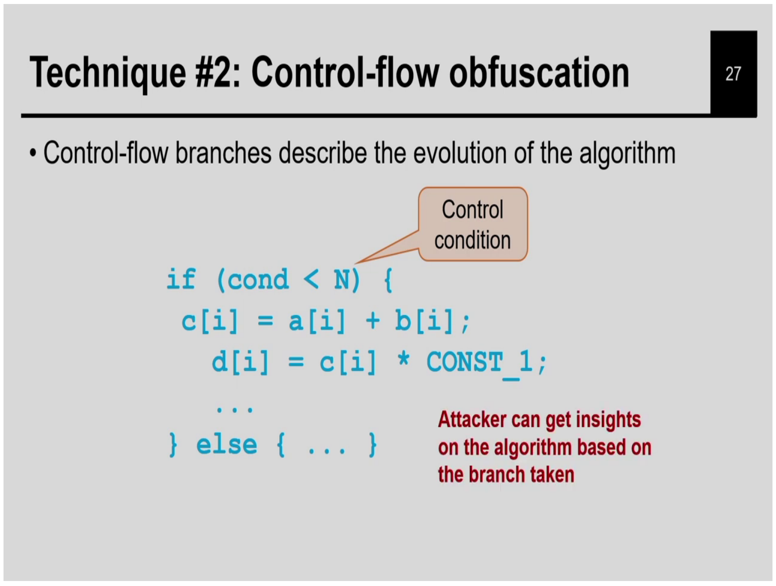
No differences concerning security, less key bits

No semantic changes

- Exact information is removed from the circuit
  - Less information for the attacker
  - Less logic optimizations (area overhead)

So, this is something very interesting and here you can see the overhead is only a XOR gate, then nothing else. So, the overhead is very less. So, you have to see whenever you do a locking what is the area overhead and you can see here what I am just doing everything is same only the things I am adding is this XOR gate and the overall area overhead value very less for this technique.

(Refer Slide Time: 12:39)



Technique #2: Control-flow obfuscation

27

- Control-flow branches describe the evolution of the algorithm

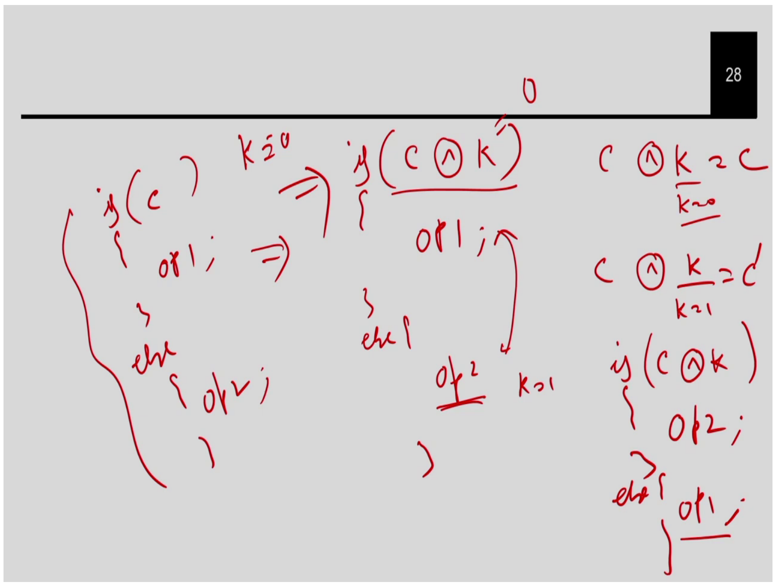
Control condition

```
if (cond < N) {
 c[i] = a[i] + b[i];
   d[i] = c[i] * CONST_1;
   ...
} else { ... }
```

Attacker can get insights on the algorithm based on the branch taken

But this can actually have a huge key into my circuit. The next one is the control flow obfuscation. So, that is something I have already told you that you have to just make sure that the control decision cannot be known without the value of the key. So, let me give an example that suppose you have a condition C here and then you are doing say operation1 else you are doing operation2, very simple.

(Refer Slide Time: 12:56)



28
if (c) { op1; } else { op2; }
K=0
if (C ⊕ K) { op1; } else { op2; }
K=1
C ⊕ K = C
K=0
C ⊕ K = C'
K=1
if (C ⊕ K) { op2; } else { op1; }

What I am going to do is, now C XOR some key then if I put operation1 else operation2. So, now suppose if my key is equal to 0. So, if you do some condition XOR 0; that means, it is C. So, if K equal to 0 this C XOR K equal to C. Then so that means, it will keep my structure is same.

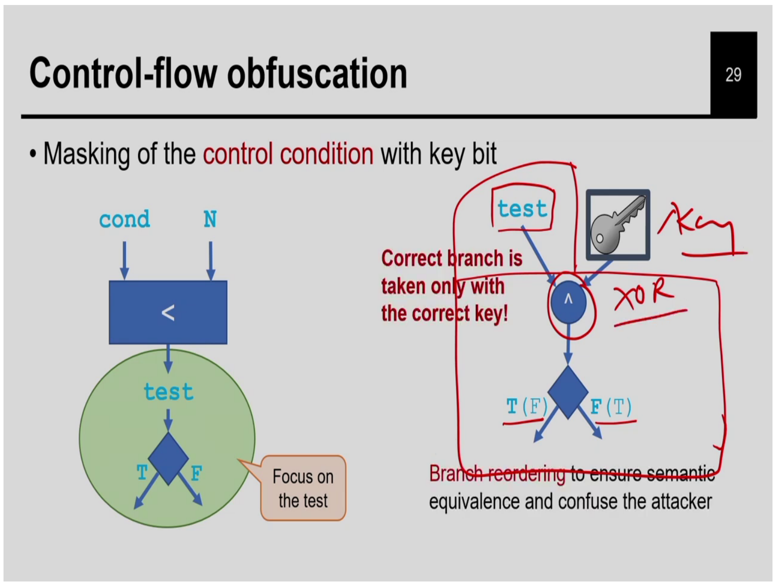
Now, if K equal to 1, if now do this XOR with K where K equal to 1, then it will just complement the C, it becomes C'. So, in this case what I have to do I have to just swap these two operations. So, if K equal to 0 implies I will just generate this code and if K equal to 1 I am going to generate this code.

So, C XOR K, I am going to put operation2 here and I am going to put operation1 here, basically I just swap this because it is this condition become C'. So, this condition will become automatically C' and if the C' is true I am usually should do this operation2. So, that is why I just put operation2 here and if C is true the else will be C which that is why I am putting operation1 here.

So, because the user/attacker does not have the information about this actual circuit, unless he knows that K equal to 0 or 1, he will not be able to understand what operation is to be executed. And whether these things are swapped or not he has no idea because he does not have the original code available with him.

So, this is something the basic idea that you have this condition you locked with a key and if you if the key is 0 and then you just keep the if else block as it is and if the key is 1 you just toggle the if block with else and else block with if.

(Refer Slide Time: 15:20)



Control-flow obfuscation

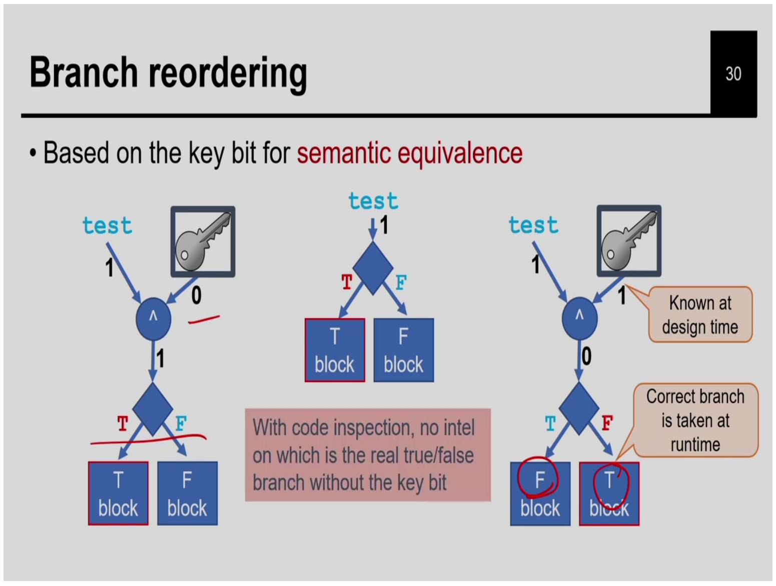
29

- Masking of the control condition with key bit

cond
N
<
test
T
F
Focus on the test
test
Key
XOR
Correct branch is taken only with the correct key!
T (F)
F (T)
Branch reordering to ensure semantic equivalence and confuse the attacker

So, this is how the control locking is going to happen. So, this is something like this. So, this is the actual condition test, this is the XOR I am adding, and this is the key and then you just have this block and then either it will be true or false, or false or true based on the value of the key. So that means, again what I am doing is this is my circuit and this is my key which is coming from input and the additional logic is I am just adding this XOR gate.

(Refer Slide Time: 15:52)



Branch reordering

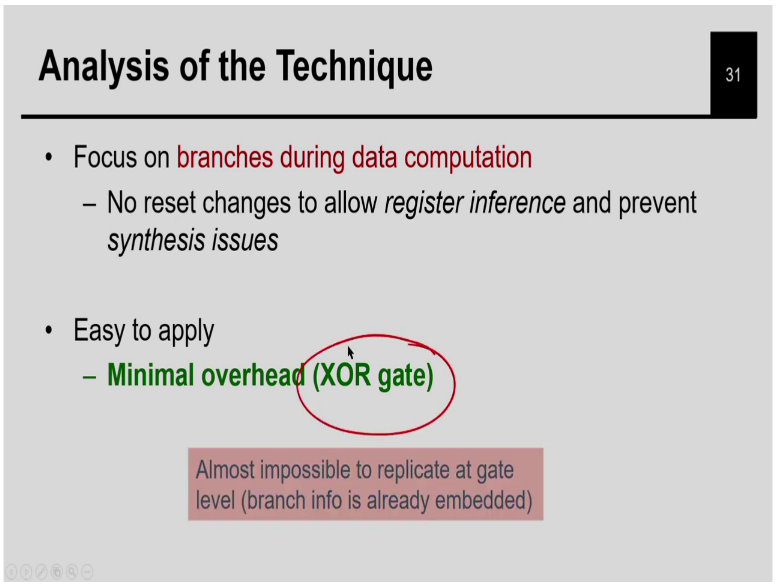
30

- Based on the key bit for semantic equivalence

test
1
0
1
T
F
T block
F block
test
1
T
F
T block
F block
With code inspection, no intel on which is the real true/false branch without the key bit
test
1
1
Known at design time
0
T
F
F block
T block
Correct branch is taken at runtime

So, area overhead will be less and this will again block the control decision. So, and now you can understand that if the if you take a wrong decision if your you give a wrong value you will go to the wrong control branch and your whole output will change. So, you will not to get the correct output. But the important factor is that if this key is equal to 0 you will keep true false as it is, but if you have this key equal to 1 you have to swap this false block with true block that I have already discussed.

(Refer Slide Time: 16:20)



Analysis of the Technique

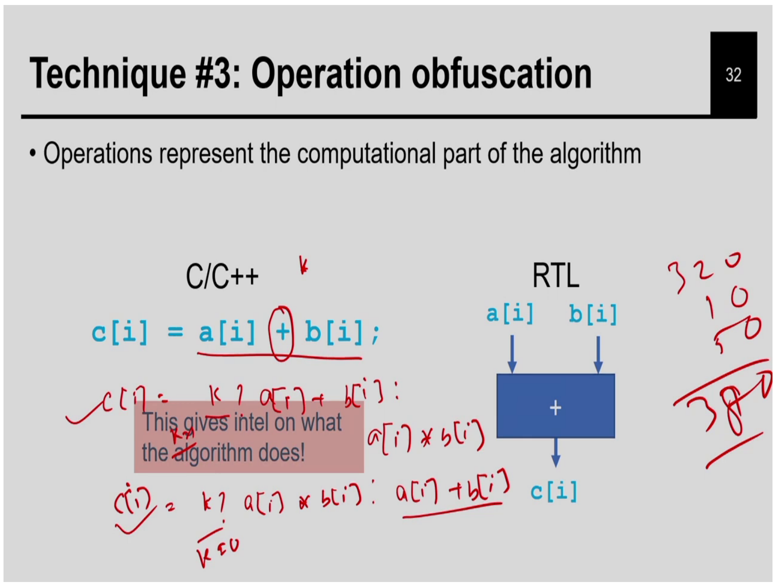
31

- Focus on branches during data computation
  - No reset changes to allow *register inference* and prevent *synthesis issues*
- Easy to apply
  - **Minimal overhead (XOR gate)**

Almost impossible to replicate at gate level (branch info is already embedded)

Again, this is something very interesting technique and the overall area overhead is very minimum, so this is the control flow.

(Refer Slide Time: 16:27)



Technique #3: Operation obfuscation

32

- Operations represent the computational part of the algorithm

This gives intel on what the ~~algorithm~~ does!

The next one is the operation obfuscation; this is very simple that you have this a plus b. What I am going to do? I am I am going to lock this. So, I am not going to reveal whether it is an addition operation or a multiplication operation if you do not know the key.

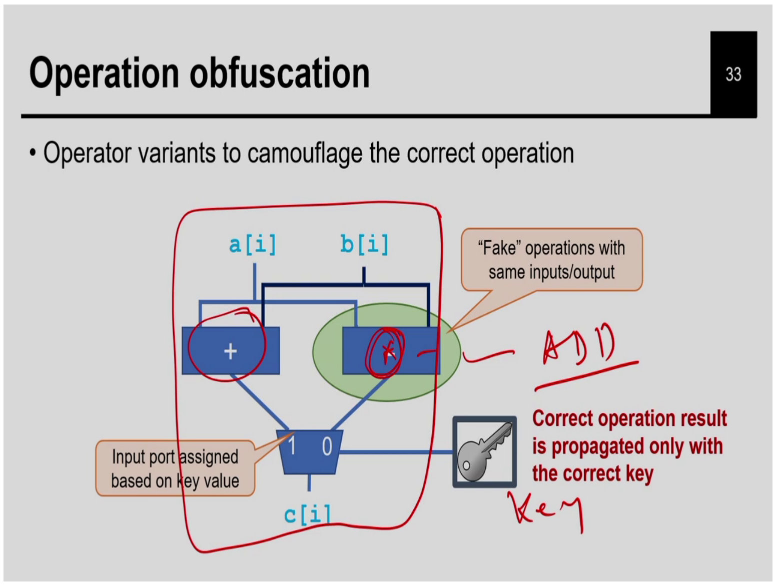
So, suppose so what I am going to do? I will just do c of i equal to K, then you do this a of i plus b of i else you do say a of i multiplication b of I, see in this case you can understand K equal to 1, because if K equal to 1 then you are going to do the actual operation else you do the same fake operation. But if you decide that my K equal to 0 then what you have to do, you have to put this a of i star b of i at the true condition and actual operation a of i plus b of i at the false operation, because here K equal to 0.

Because you are actually doing the locking you can actually decide whether my K key value is 0 or 1 based on then you will generate this line or this line, because you just know that my key value equal to 1 then you will just generate this expression. And if you decide that my key value is 0 then I am going to generate this expression. But since the attacker does not know the actual operations plus, he has no clue whether this is the correct operation or this is the correct operation.

So, as a result what is going to happen until he knows the key value correctly, he does not know what is the output and if you give a wrong key then the output will be wrong it is well understood. But you can understand from here that it is a single b see there are two choice you can do that, but if you do our lock say 100 such operations, there are 100 such operation will be locked and now you think about locked your operations with condition as well as with the constant.

You can understand the complicacies I am generating, say suppose you have locked 10 constants. So, there are 320 bits, 10 condition if else branch, so 10 bits and say 50 operations. So, 380 bit key will be generated, and you can understand this is impossible to break such circuits or it is very difficult to break such circuit.

(Refer Slide Time: 18:57)



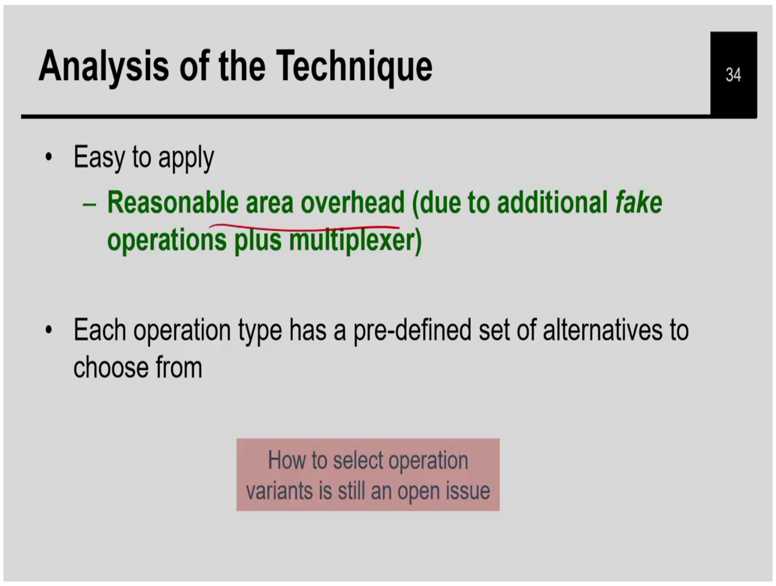
Operation obfuscation

- Operator variants to camouflage the correct operation

a[i]
b[i]
+
*
"Fake" operations with same inputs/output
ADD
1 0
Input port assigned based on key value
c[i]
Correct operation result is propagated only with the correct key
Key

So, so that was the idea of this operator obfuscation that you have the original operations. And say this is my fake operation it can be anything star minus anything and then based on the key value you will actually decide which is the correct operation. So, this is my circuit and this is the key which is coming from input. So, this is the operation obfuscation.

(Refer Slide Time: 19:17)



Analysis of the Technique

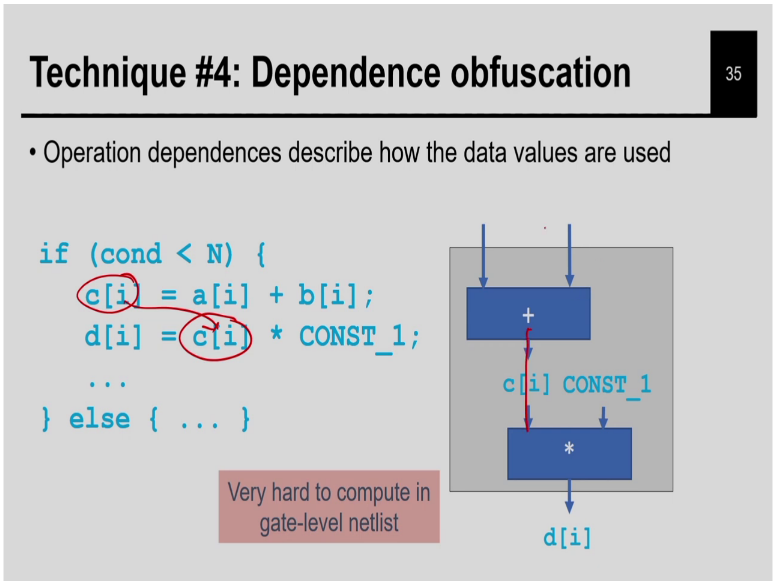
34

- Easy to apply
  - Reasonable area overhead (due to additional *fake* operations plus multiplexer)
- Each operation type has a pre-defined set of alternatives to choose from

How to select operation variants is still an open issue

The last one is the dependency. So, here you can see there are some overhead is little bit more because this is an operation it was not there earlier. Here you are going to add this, so here the overhead of area will be more because it is not XOR gate, here its complete multiplier or a subtractor we are adding to the circuit. So, as a result your overall will be little bit reasonable area overhead has for this technique.

(Refer Slide Time: 19:42)



Technique #4: Dependence obfuscation

35

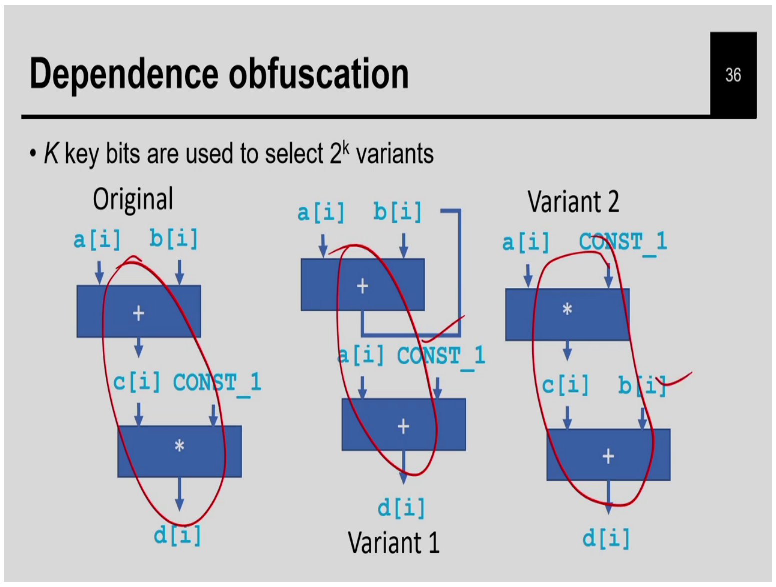
- Operation dependences describe how the data values are used

```
if (cond < N) {
   c[i] = a[i] + b[i];
   d[i] = c[i] * CONST_1;
   ...
} else { ... }
```

Very hard to compute in gate-level netlist

The fourth one is the dependency. So, as I mentioned that if this operation is defined here and you are happening here what we can do we can create different such variance. So, you can understand here this is the dependency. So, c of i is happening here and this is say operation chaining is happening.

(Refer Slide Time: 19:59)

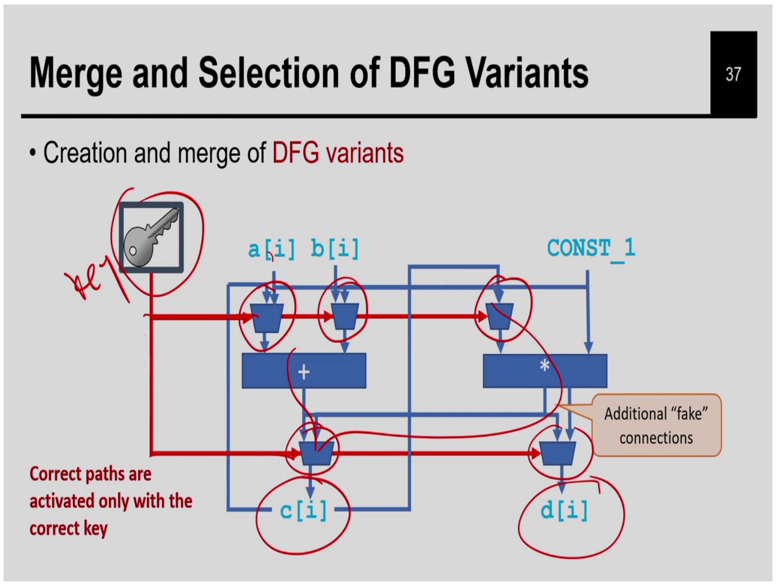


Dependence obfuscation

- K key bits are used to select 2^k variants

So, what we can do? We can create different variance of the data paths. So, this is the actual correct dependency this is some a plus plus this is a star plus and so on. So, you generate different such data paths.

(Refer Slide Time: 20:10)



Merge and Selection of DFG Variants

- Creation and merge of DFG variants

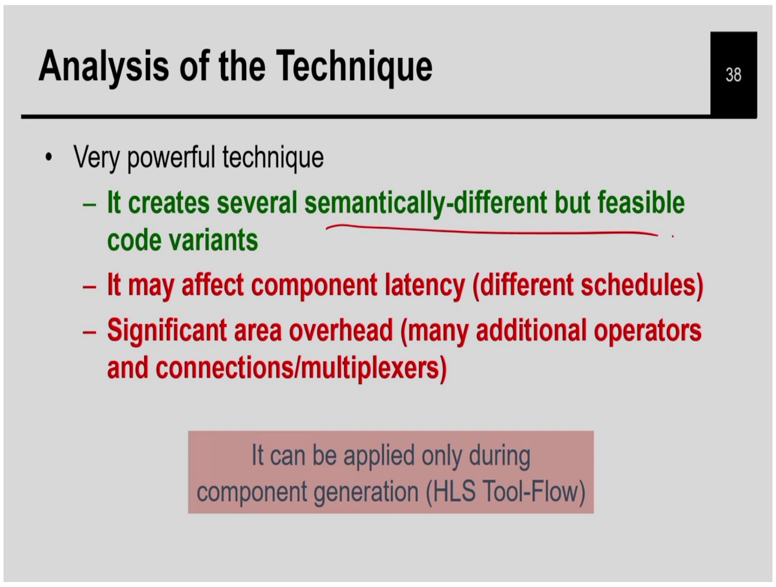
key
a[i] b[i]
CONST_1
+
*
Additional "fake" connections
Correct paths are activated only with the correct key
c[i]
d[i]

And then based on the key this is the key which will determine the actual operation actual dependency. So, if you give the correct key then only the actual a of i plus b of i will come, if you give this wrong data then this will come. So, on then as a result your output will be wrong.

So, the dependency also I hide using a key. So, with the key will create different data paths variant and I will combine them into a single data path and then using the MUX I will give the correct key and as a result only if you give the correct key then only this circuit will work.

So, this is the idea of the DFG you can also understand here that and since I am doing many such extra operations and I am adding many such additional multiplexers just to fake. I mean just to hide the actual dependencies my area overhead will be also significant in this case.

(Refer Slide Time: 21:05)



Analysis of the Technique

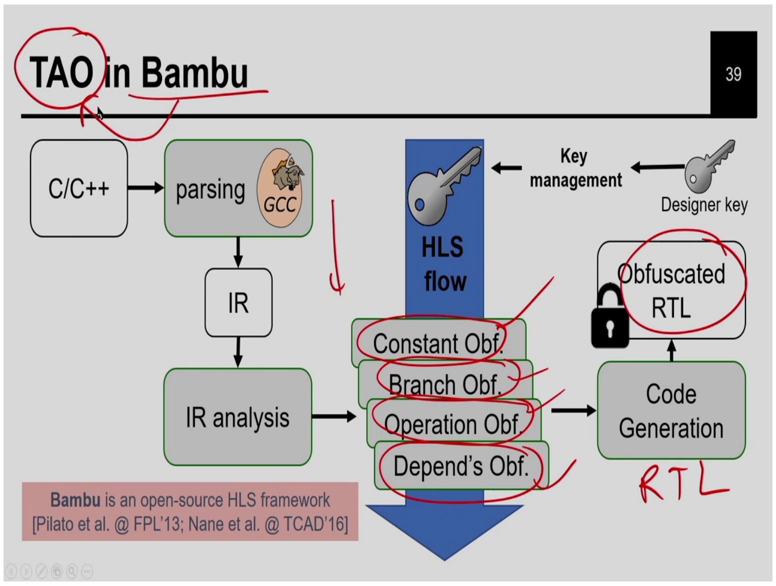
38

- Very powerful technique
  - It creates several semantically-different but feasible code variants
  - It may affect component latency (different schedules)
  - Significant area overhead (many additional operators and connections/multiplexers)

It can be applied only during component generation (HLS Tool-Flow)

So, it is some there will be some area operate in this.

(Refer Slide Time: 21:11)



TAO in Bambu

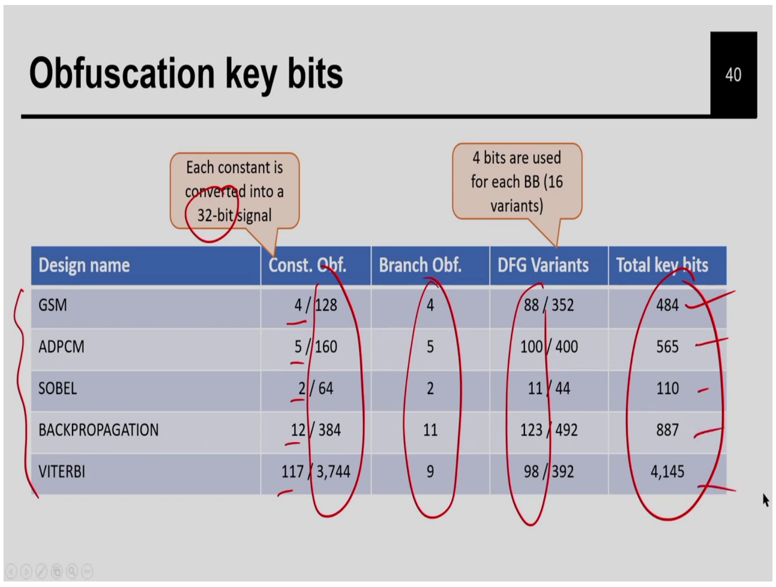
Bambu is an open-source HLS framework [Pilato et al. @ FPL'13; Nane et al. @ TCAD'16]

And you can understand that I have taken this C code and after doing the pre-processing I will get the intermediate representation and there what I can do I can just do is, I can identify all the constant and I can lock them, I will identify all the if else branch I can lock them I will identify all the operations I can lock them.

And then I can do this high-level synthesis right scheduling allocation binding. And during this also for these dependency things during this allocation, because I have to allocate these additional things and now there I can actually identify the dependency I can add some fake resource in the data path and I can generate a obfuscated RTL.

So, that is something the idea. So, then you have the RTL here and there you need the key. So, that is how I can generate this and this has been implemented in a tool called TAO which is an open source high level synthesis tool is Bambu we had this all this technique has been integrated and this is a public available tool TAO which does all these things.

(Refer Slide Time: 22:13)



Obfuscation key bits

40

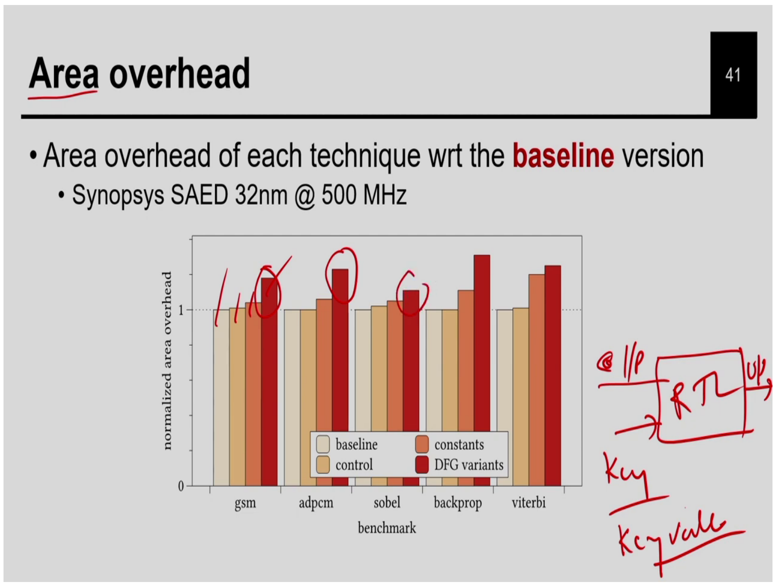
Each constant is converted into a 32-bit signal

4 bits are used for each BB (16 variants)

Design name	Const. Obf.	Branch Obf.	DFG Variants	Total key bits
GSM	4 / 128	4	88 / 352	484
ADPCM	5 / 160	5	100 / 400	565
SOBEL	2 / 64	2	11 / 44	110
BACKPROPAGATION	12 / 384	11	123 / 492	887
VITERBI	117 / 3,744	9	98 / 392	4,145

So, just to understand that I have some examples say high level synthesis benchmarks where you just you just do this, these are the number of constants you just locked it and it is 32 bits then you can understand the key size here. This is the key size this is the number of branch you obfuscated there this is the DFG variant you have done and this is the key size is getting you can see here the key size is always very high 400, 500, 100, 800, 4000. So, we can actually create a very high key and it will be very difficult to identify the correct value of the keys and the RTL will look like this.

(Refer Slide Time: 22:54)



Area overhead

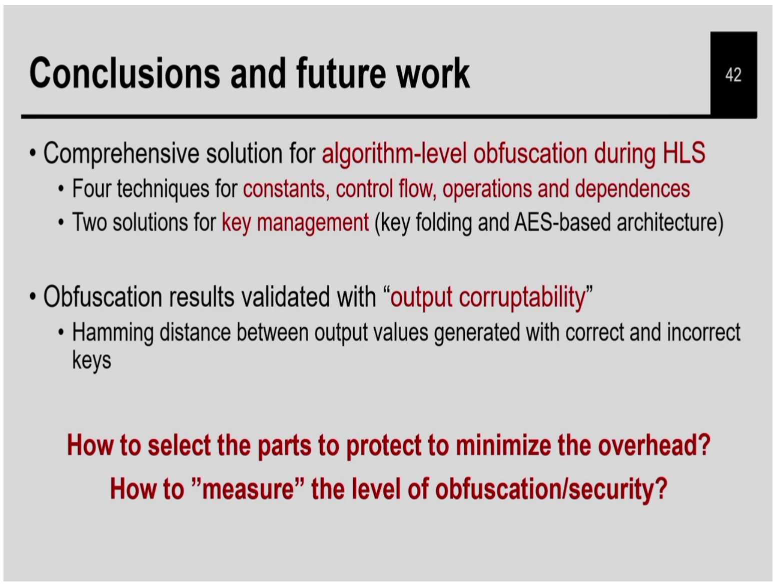
41

- Area overhead of each technique wrt the **baseline** version
  - Synopsys SAED 32nm @ 500 MHz

So, basically you have the RTL now which has the actual inputs and it has the actual outputs and it will actually of the additional input key and this high level synthesis tool will generate this locked RTL plus the actual key value.

And then this key value is only known by the genuine design house and if you give if this RTL to the during layout, they will not able to get the correct value of the keys. So, as I mentioned there are some areas over it and I have seen that this first two techniques like constant obfuscation and this condition obfuscation have just adding a XOR gate. So, the area overhead is very less, but once you do these operation obfuscations there is some area overhead because it is adding some extra thing. So, this is something has been shown in the particular paper but it is not so important.

(Refer Slide Time: 23:55)



Conclusions and future work

42

- Comprehensive solution for algorithm-level obfuscation during HLS
  - Four techniques for constants, control flow, operations and dependences
  - Two solutions for key management (key folding and AES-based architecture)
- Obfuscation results validated with “output corruptability”
  - Hamming distance between output values generated with correct and incorrect keys

How to select the parts to protect to minimize the overhead?

How to ”measure” the level of obfuscation/security?

So, just to conclude in this particular lecture what we have understood that hardware security can be achieved against the IC design flow the things that IP piracy and over building can be done through logic locking. And high-level synthesis can be a tool which can actually generate a locked RTL by locking the constant, locking the operation, locking the condition, locking the dependencies.

And the RTL that will be generate will be locked with a very high key value. As a result, that particular RTL even if you go through this the different design house without the correct key value, they will not able to use that particular RTL. So, high level synthesis can be used for securing the design.

Thank you.