Eliminating Power Side-Channel Leaks during Register Allocation

Jingbo Wang
Department of Computer Science
University of Southern California
Los Angeles, USA
jingbow@usc.edu

***Index Terms*—Verification, Mitigation, Cryptography, LLVM Compiler, SMT Solver**

I. Technical Background

Side channel attacks allow attacker to infer the security-critical information of a system by observing its external behaviour, which are possible due to the presence of an underlying vulnerability. For example, in the case of timing channel, attackers can infer the relevant information about security-sensitive data by observing the execution time it takes to perform specific branches or some operations. After the introduction of execution timing channel as the first practical side channel to recover the secret key of implementation of cryptographic algorithms, other side-channel analysis approaches have been introduced later [4]. For instance, power consumption, acoustic, optical emission and temperature are those which have been brought to the attention of research community. Nevertheless, due to the efficiency, low-cost and simplicity power consumption and electromagnetic emanation side channels have been widely investigated in academia more than others. For instance, power side channel attacks are feasible because the program exhibits different power consumptions based on the properties involving *key*. Countermeasures proposed to overcome Simple Power Analysis (SPA) and Differential Power Analysis (DPA), are dummy instruction insertions, data masking, table masking, balancing bit-flips and so on. All these methods are either unable to cover all encryption algorithms or burden the system with high area cost and energy consumption. [1]

Hence, the most effective defense against side-channel attack is to eradicate the underlying vulnerabilities by ensuring the resource usage won't differ with respect to *key*. In this work, we pay most attention to the power side channel. Recent works have explored some countermeasures for avoiding the power side channel and develop side-channel-free programs by automatically analyzing the correlations between difference in power consumption and variation in security-sensitive data. This work [3] proposes new synthesis method for generating countermeasures for cryptographic software code, which takes an unprotected program as input and returns a functionally equivalent but power channel leak free new program as output. This countermeasure is based on the Hamming Weight model and is guaranteed to be perfectly masked in all intermediate results which is independent of *key*. However, this work assumes that all the intermediate register will be distributed into distinct registers and estimate the power consumption on the basis of this assumption. After register allocation, there are multiple intermediate variables sharing the same register and this approach is still possible to leak secret information as the leakage model is not precise enough. Another work [1] injects random instructions at random places during the execution of an application which protects the system from both SPA and DPA. But this solution is coarse-grained, it just choose a region and set random operations on this region. In order to minimize performance overhead, [5] inserts two dummy loads before and after a collection of sensitive data loads and around any loads of changing data, such as input data. However, it will incur performance overhead and this analysis leak will reappear, presumably based on the Hamming weight of the state before the mixing of the round key in AES.

For the sake of solving the above problems existing in state of the art methods, we propose a more accurate leakage model and apply it to check whether the given source program has potential leakages. In addition, we also design a mitigation approach for generating power channel leak-free program. This practical countermeasure can be introduced in software to increase resistance and meet multiple security requirements.

II. Problem Statement

The practical result [4] of using static power analysis to mount a successful power side-channel attack illustrated that the registers' content and gates' output should be the main leakage resource in FPGA platform. We apply this conclusion and the classic model [2] used for the power consumption of cryptographic devices in our work to build the leakage model. Followings are leakage analysis of a source program under different power models and register allocation scenarios. With respect to this example, we will propose a more accurate leakage model for power channel which can explore more underlying vulnerabilities. In this section, we present an example owning potential power side channel leakage under finer-grained leakage model, and illustrate how this data leakage problem can be solved by our approach in *Section III (Solution Section)*.

A. Motivating Example

Given the *leak()* function in Figure 1. Here we assume each input and the output are all 1-bit data. Input *key* is the secret data while input *i1,i2* and *i3* are plaintext. The output *n3* is regarded as the ciphertext and *n1,n2* are both intermediate variables involving with the key and the plaintext value. This simple cryptographic program consists of the linear($\oplus$) and non-linear($\wedge$) calculations. In order to verify this program is leak-free, we need to make sure that values of all the intermediate variables are statistically independent of *key*. For instance, given different *key* values and same plaintext inputs, if *n1* exhibits various power channel behaviours, then it shows that the program in Figure 1 exists side channel leakage, as there are *key*-induced different power consumptions. To verify if there are some *key*-induced differences in power consumption, we compare and apply different power analysis models in subsequent sections.

```
int leak(int i1, int i2, int key, int i3)
{
        int n1,n2,n3;
        n1 = i1 ^ i2;
        n2 = (n1 ^ 1) ^ key;
        n3 = n2 & i3;
        return n3;
}
```

Fig. 1: Example1 source code

B. No Leak under Hamming Weight (HW) Model

In Hamming Weight power leakage model, one can mathematically describe the power consumption according to the computed bits(number of 1s in the intermediate byte value), which means there is a correlation between the Hamming Weight of the intermediate variables and the power consumption traces. In the example Fig 1, if Hamming Weights of the intermediate variables are independent of the *key* value, then it can make sure that this program is power side channel leakage free.
According to the Table I, the value of intermediate variables *n1,n2* are not dependent on the *key* value, which means that the number of 1-bit is constant when *key* differs. Under Hamming Weight model, we assume that the power leakage of the device correlates to the intermediate variables involved in the sensitive operations. All the intermediate variables *n1,n2* have immutable number of 1-bit with different *key* values, hence the power consumption won't change either. This program will be regarded as power side channel free as the resource usage won't differ as the *key* value varies.

C. No Leak under Hamming Distance (HD) Model

Under Hamming Distance(HD) model, the data leakage through the power side-channel depends on the number of bits switching from one state to the other. In this case, the leakage current consumed is related to the number of 1-0 transit or 0-1 transit. In Hamming Distance model, there is a reference

key	i1	i2	i3	n1	n2	n3	n1⊕n2	n2⊕n3
0	0	0	0	0	1	0	1	1
0	0	0	1	0	1	1	1	0
0	0	1	0	1	0	0	1	0
0	0	1	1	1	0	0	1	0
0	1	0	0	1	0	0	1	0
0	1	0	1	1	0	0	1	0
0	1	1	0	0	1	0	1	1
0	1	1	1	0	1	1	1	0
1	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0
1	0	1	0	1	1	0	0	1
1	0	1	1	1	1	1	0	0
1	1	0	0	1	1	0	0	1
1	1	0	1	1	1	1	0	0
1	1	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0

TABLE I: Truth Table of Example1

state to decide which the bits are switched. We assume this reference state is a constant value, *R*. It will always be the same if the same data manipulation occurs at the same time. In the HD leakage model, the number of flipping bits to go from R to D is described by $HD(D,R)$ = $HW(D \oplus R)$ (The variable value $D = \sum_{j=0}^{m-1} d_j 2^j$ with the bit values $d_j = 0$ or 1, $H(D) = \sum_{j=0}^{m-1} d_j$. Here we assume one bit, then m = 1). Actually, the Hamming Weight model is a specific case for HD model which assumes that R = 0. The assumption also propose a linear correlation between the current consumption and the $HD(D,R)$.
As we can see from this example, for intermediate variable *n1*, assuming R is 1, hence the average value of $HD(n1,R)$ is 0.5 under both cases that *key* equals to 1 or 0. While R is 0, the *HD* value is also same under different *key* values, which means that the Hamming Distance between intermediate variable *n1* and R is constant as *key* varies. Hence, the constant HD values represent the constant power consumption under different *key* values. In sum, the source code in Figure 1 is also power leakage free under Hamming Distance model.

D. Leak after Register Allocation

As previous research shows, the dependency of the leakage current to the register's content is clear. In Section A and B, we all assume that all intermediate variables are allocated to distinct registers. If the register value or value transition inside register have no correlation with the *key*, which means the power consumptions have no relation with *key*, then it's side channel leakage free. However, in specific target platform such as X86-64, diverse intermediate variables may reuse the same register. Under HW model, all intermediate values have no correlation with *key* in program of Figure 1. If the intermediate values are stored in same register, which means sum of 1-bit the register owns equals to the sum of 1-bit from all the intermediate variables. As the register value is independent of *key* as the intermediate variables, sum of the register values are also irrelevant to *key*. Hence, the power consumption won't depend on the *key* either.
Nevertheless, under HD model, there are some leakage issues

after the register allocation. Figure 2 is the assembly code from this example in X86-64 system. *i1,n1,n1⊕key, n2* are stored in the same register %edi. Most critically, two consecutive states of %edi register will stores the value of *n1*, *n1⊕key* as line 11 shows in Figure 2. We assume $\%edx_1$ represents *n1*, $\%edx_2$ represents *n1⊕key*. In HD model, HD($\%edx_1, \%edx_2$) = HW($\%edx_1 \oplus \%edx_2$) = HD(*key*). Hence the amount of the 0-1 or 1-0 transitions between two states of the register %edx will depend on *key*, which means the power consumption will also have relation with the *key* value. Therefore, this program still have power side-channel leakage after the register allocation. Meanwhile, we can also see from Table I, the value of *n1* $\oplus$ *n2* representing the number of transition bits between two successive states of the register keeps 1 when *key* equals to 0, vice versa. Hence, as the number of transition bits inside a register depending on *key* have correlation with power consumption, the *key* value will definitely relate with the power consumption, which results in the side channel leakage. In other platforms such as ARM or MIPS, there are some intermediate variables sharing the same register as well. Therefore, under the Hamming Distance power analysis model and the register allocation scenario, the power channel leakage problem generally exists in various platforms.

```
1  leak:
   # @leak
2          .cfi_startproc
3  # BB#0:
   # %entry
4          pushq   %rbx
5  .Ltmp2:
6          .cfi_def_cfa_offset 16
7  .Ltmp3:
8          .cfi_offset %rbx, -16
9          movl    %edi, %ebx
10         xorl    %esi, %ebx
11         xorl    %edx, %ebx
12         xorl    $1, %ebx
13         andl    %ecx, %ebx
14         movl    $.L.str, %edi
15         xorl    %eax, %eax
16         movl    %ebx, %esi
17         callq   printf
18         movl    %ebx, %eax
19         popq    %rbx
20         ret
```

Fig. 2: Example1 assembly code on X86-64

III. Solution

In previous section, we have shown a power channel leakage program, the power consumption of which depends on *key*. But the existing method [3] will regard it as leak-free, and fail to mask it. In this paper, we propose a more precise leakage model to check whether the given program exists power channel leakage and design a mitigation method for eliminating the potentially leakage point and return a leak-free program. We will utilize LLVM IR to implement the detection pass and build the register reallocation backend for generating leak-free program. .

A. Leakage Detection

Taken both Hamming Distance model and register allocation into consideration, we aim to develop a more accurate detection pass to check whether the program has power channel leakage. As power consumption is relevant to the value transition in register, it's necessary to check whether the sum of state transitions will differ with respect to different *key*. To begin with, the variable information in LLVM IR is not enough for analysis as there are infinite registers allocated in *IR*. Hence, we can't acquire the critical information about which variables sharing the same register only from *IR*. Hence, we propose a detection method which utilize both the information from *IR* and the register allocation process. Figure 3 gives an overview about how to employ both the source information and the register allocation information to detect whether the source program exist leakage.

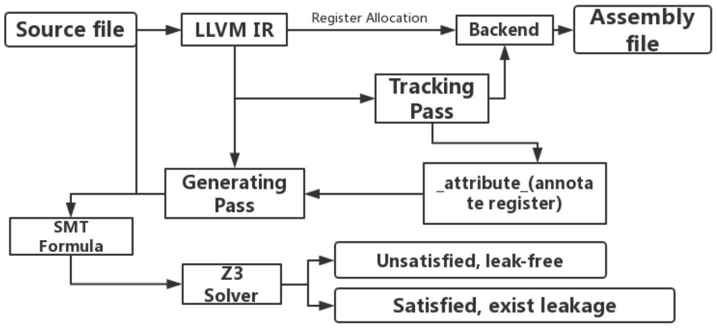

Fig. 3: Overview of Detection Process (Zoom in for detail)

Firstly, the Tracking pass will monitor the register allocation process in the backend, and gather such information. For instance, if intermediate variables *v1* and *v2* are not showing up in the same live-variable set of each instruction, then *v1* and *v2* can be distributed into same register. Tracking pass will annotate such variables sharing the same register, and transmit this annotated information to another Generating pass. Generating pass will accept the input from *IR* file and the annotated information from the Tracking pass, and return the SMT formula. The SMT formula for verifying whether the power consumption of program differs with respect to *key* are illustrated as below.

$$\varphi_{diff} = \exists k_1, k_2, k_1 \neq k_2 \wedge (\sum_{x \in P} I(x,k_1) \neq \sum_{x \in P} I(x,k_2) \vee \varphi_{RegReuse})$$

φ_{diff} represents whether the formula that the power consumption of program will differ according to the different *key*. Given various *key*, it's necessary to check both the sum of all possible intermediate values and the sum of transitions of all the registers are different. $\sum_{x \in P} I(x,k_1)$ represents the sum of all variable values under possible permutations of the input x and k_1. While $\varphi_{RegReuse}$ means that if the variables have reused the same register, then if the value transitions of the same register depend on *key*, which can also imply that the source program exists *key*-induced difference of power consumption. $f(I)$ equals to true if intermediate variable I shares the same register with other variables, vice versa.

$$\varphi_{RegReuse} = (f(I) \wedge \sum_{x \in P} \Delta I(x, k_1) \neq \sum_{x \in P} \Delta I(x, k_2))$$

Algorithm 1 Leakage Detection

```
 1: procedure DETECTION
 2:     M ← number of intermediate variables
 3:     N ← number of possible permutations I
 4:     i ← iterator of instructions inside the function
 5:     Sum, $Transition_{sum}$ ← Output
 6: Initialize:
 7:     Reg ← $Map_{VR}(var)$
 8:     Var ← $Map_{RV}(reg)$
 9: loop:
10:     if inst_iterator i ≠ inst_end() then
11:         if I is annotated then
12:             f(I) ← True
13:         goto loop.
14:         close;
15:     $cnt\_n = 0, cnt\_m = 0$
16:     do
17:         $cnt_m += 1$
18:         do
19:             $cnt_n += 1$
20:             $Ivar = Intermediate_{var}(cnt\_n)$
21:             $Sum(cnt\_m) += Ivar$
22:             $VarNext = Map_{RV}(++Map_{VR}(Ivar))$
23:             $delta = f(I) ? VarNext : 0$
24:             $Transition_{Sum}(cnt\_m) += delta$
25:         while $cnt\_n \neq N$
26:     while $cnt\_m \neq M$
```

B. New Countermeasure

The initial state and state transition in the same register is related to power consumption. The new-found leakage arises from the fact that different *key* values can induce different state transitions, thus power consumption also changes. As we can see from Figure 2, in line 11, %edx initially stores the *key* value. Under HW model, the number of 1-bit in register will influence the power consumption, hence, different *key* values will induce the difference in power consumption of register %edx. In Line 10-11, state transition of the register %edi is dependent on *key*. Under HD model, the number of 1-0 transitions in register correlates to the power consumption, therefore, *key*-induced state transition in register %edi is also related to the power consumption.

For solving those leakages in power channel, we must ensure that the *key* is independent of the power consumption. Based on the HW and HD model, we've proposed a mitigation method which spills the *key* value and intermediate result directly related to *key* into memory. There are two reasons for this solution: Firstly, in X86 instructions, the operands of the operation can be a register or memory location. If we put the *key* in one register, it's obvious the power consumption of this register is dependent on *key*. Hence, we store *key* in memory before calculation. Secondly, in Section D, we propose that due to the fact that multiple intermediate variables are allocated in the same register, hence the state transition is related to *key*. If we allocate distinct registers to intermediate variables, the problems still exist. Because most arithmetic instructions are 2-operand design, which means that it's possible one register representing both the source operand and the result operand. Hence, for one instruction, the state transition from the source value to result value is related to another value stored in other locations. As the *key* has been stored in memory, if we put the *key*-related calculation result into another register such as *R*, then the transition of *R* is dependent on *key*, which still owns the power channel leak. Hence, we also put the *key*-related result into memory. (The solution is based on the assumption that the value or value transitions of the register correlate with the power consumption while the memory does not.)

Hence, after modifying the register allocation, the secure

```
 1  leak:
    # @leak
 2          .cfi_startproc
 3  # BB#0:
    # %entry
 4          pushq   %rbx
 5  .Ltmp2:
 6          .cfi_def_cfa_offset 16
 7  .Ltmp3:
 8          .cfi_offset %rbx, -16
 9          movl    %edi, %ebx
10          xorl    %esi, %ebx
11          xorl    $1, %ebx
12          xorl    %ebx, 24(%esp)
13          movl    %ecx, %ebx
14          andl    24(%esp), %ebx
15          movl    $.L.str, %edi
16          xorl    %eax, %eax
17          movl    %ebx, %esi
18          callq   printf
19          movl    %ebx, %eax
20          popq    %rbx
21          ret
```

Fig. 4: Mitigation of Example1 assembly code on X86-64

assembly code is as Figure 4. In Figure 4, we put *key* value in the stack(24(%esp)), and stores the *key*-directly-related result in the same memory position as *key* is out of the live range after line 16. Later, to avoid the state transition of register %ebx has some relation with *key*, we put another input variable *i3* in this register %ebx instead of the *key*-directly-related result. After that we execute Line 6 in Figure 1, load the intermediate variable *n2* stored in memory, and execute the *and* operation with the *i3* stored in register %ebx. Finally, the output result is in %ebx. In this way, both the initial state and the state transition of the register has no correlation with *key*.

IV. Conclusion

As previous methods didn't consider the register allocation scenarios under the Hamming Distance power analysis model, therefore they fail to mask the potential power-channel leakage program. This paper propose a more precise leakage model for verifying whether the given program exist power channel leakage after the register allocation phase. Meanwhile, we also propose a mitigation approach for generating the leak-free program by designing a secure register allocation phase, which put the public data into the register as usual and spill the sensitive data into memory to protect the attacker from inferring the *key* by some techniques such as differential power analysis.

References

[1] Jude Angelo Ambrose, Roshan G Ragel, et al. Rijid: random code injection to mask power analysis based side channel attacks. In *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pages 489–492. IEEE, 2007.

[2] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 16–29. Springer, 2004.

[3] Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In *International Conference on Computer Aided Verification*, pages 114–130. Springer, 2014.

[4] Amir Moradi. Side-channel leakage through static power. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 562–579. Springer, 2014.

[5] Sami Saab, Pankaj Rohatgi, and Craig Hampel. Side-channel protections for cryptographic instruction set extensions. *IACR Cryptology ePrint Archive*, 2016:700, 2016.