Constraint Based Program Synthesis for Embedded Software

Hassan Eldib

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Chao Wang, Chair
Patrick Schaumont
Michael Hsiao
Sandeep Shukla
Eli Tilevich

May 5, 2015
Blacksburg, Virginia

Keywords: Program Synthesis, Formal Verification, Embedded Software, Security, Cryptography,
Side-Channel Attacks and Countermeasures

Constraint Based Program Synthesis for Embedded Software

Hassan Eldib

(ABSTRACT)

In the world that we live in today, we greatly rely on software in nearly every aspect of our lives. In many critical applications, such as in transportation and medical systems, catastrophic consequences could occur in case of buggy software. As the computational power and storage capacity of computer hardware keep increasing, so are the size and complexity of the software. This makes testing and verification increasingly challenging in practice, and consequentially creates a chance for software with critical bugs to find their way into the consumer market.

In this dissertation, I present a set of innovative new methods for automatically verifying, as well as synthesizing, critical software and hardware in embedded computing applications. Based on a set of rigorous formal analysis techniques, my methods can guarantee that the resulting software are efficient and secure as well as provably correct.

Acknowledgment

First and foremost, I would like to express my deepest appreciation to my Ph.D. advisor, Professor Chao Wang, from whom I have learned incredibly a lot. His continuous guidance, dedication and encouragement has helped me become a more mature researcher and shape my future goals. Professor Wang will always remain a source of inspiration for me, even after graduation.

I would also like to express my gratitude to Professor Patrick Schaumont, Professor Michael Hsiao, Professor Sandeep Shukla, and Professor Eli Tilevich for serving on my committee. It has been a great honor for me to work closely with them.

I am grateful to my parents for their never-ending encouragement and support throughout my life. I am truly fortunate to have such parents.

Finally, I want to thank my wife and my daughter for their endless love and support throughout these years.

Hassan Eldib

Contents

List of Figures

List of Tables

List of Abbreviations

AES Advanced Encryption Standard

AST Abstract Syntax Tree

BFS Breadth-First Search

DAC Directed Acyclic Graph

DC Don’t Care

DFS Depth-First Search

DPA Differential Power Attack

DSP Digital Signal Processing

EDA Electronic Design Automation

FPGA Field Programmable Gate Array

FSA Fault Sensitivity Analysis

HD Hamming Distance

HW Hamming Weight

LSB Least Significant Bit

LTL Linear Temporal Logic

IR Intermediate Representation

MAC Message Authentication Code

MO Memory Out

NIST National Institute of Standards and Technology

OS Operating System

RSA Rivest, Shamir and Adleman

SAT Satisfiability

SHA Secure Hash Algorithm

SMT Satisfiability Modulo Theories

TO Time Out

QMS Quantitative Masking Strength

Chapter 1

Introduction

Embedded devices are found in many critical systems, such as in communication, transportation, medical and military. Errors in such systems could have devastating consequences economically or even worse, loss of life. Skillful and experienced engineers are needed to design these systems to create robust products, but unfortunately there is an increasing shortage in these engineers, according to the Bureau of Labor statistics. An effective solution is developing innovative tools to support software/hardware engineers in increasing their productivity and the efficiency of their code to keep the pace needed for development.

Recent research has led to progress in inductive program synthesis methods which have the potential to help software developers in programming [78, 40, 44, 38, 39, 41, 67, 75, 4, 26, 28, 27]. A developer could use inductive program synthesis to generate a program code by specifying some input-output relation examples and the grammar required in the new code. This method has advantage over the traditional time-consuming programming methods. In the traditional method, the software developer writes the complete program then carefully analyzes the code to optimize it. In some case, the developer would even need to get some expertise in sophisticated fields to complete the task. Inductive program synthesis enhances the developer's capabilities by generating a complete or partial program that satisfies a specified input-output relation and optional added optimization constraints.

An example for using inductive program synthesis is shown by Harris *et al.* [41, 76], who implemented *Flash Fill*, an autocomplete method for Microsoft Excel Spreadsheet. An empty spreadsheet column that is function of another non-empty column is automatically filled by just having the user fill a few cells and without the need to define the relation function between both columns.

Another example for a usage of inductive program synthesis is shown by Solar-Lezama *et al.* [77, 78], who presented a *Sketching* method to aid the programmer to write code for stencil computations. The programmer would write a partial code implementation of the desired program while leaving the hard to code fragments empty, then the missing code fragments are completed by the inductive synthesis tool.

There is large potential for using inductive program synthesis in novel applications to aid software developers in programming reliable code. Two prospective areas with good potential for research are embedded control and cryptography software.

The main obstacle for practically using program inductive synthesis in different fields is the unscalability of the method. The number of possible programs to search from for a satisfying program has an exponential relation to the size of the program to be synthesized. Currently, the state of the art program inductive synthesis tools still cannot synthesis practical programs of large size.

1.1 Background

1.1.1 Embedded Control Software

Hardware of embedded devices has improved in the previous years and has become more complex to the extent that some of the embedded devices are now comparable to general purpose computer systems. These devices are now running more sophisticated programs than before.

In recent years, we have seen an increasingly large amount of software code developed for embedded systems. These software codes are often responsible for controlling many transportation,

medical, industrial machinery and weapon systems.

It is important to optimize the embedded software and adaptthem to the device hardware for efficiency. In addition, embedded software must be carefully tested to be error free in order to avoid possible damaging failure consequences. This is a challenging task as designers have to reason about many factors under which the device is run.

For example, embedded systems often execute fixed-point arithmetic computations [88]. Due to limitations in the bitwidth of the embedded device registers, the software computation results may have values that are larger than the maximum register size, which causes overflow and underflow errors. In practice, programmers often mitigate this problem by reordering the code instructions, or adding instructions to estimate the resulting value range and then add some branching statements to limit the value to the default maximum or minimum. Both of these methods reduce the accuracy and reliability of the computation.

If available, tools could be used by the software developers to simplify problems like this and avoid degradation in the code execution. Tools could assist the developer in adapting complex computational algorithms to the targeted embedded hardware, in addition to reducing the time and effort spent for this task.

1.1.2 Cryptographic Software

Sensitive information is usually encrypted to avoid adversaries from accessing it. Cryptographic software such as encryption are run on many embedded devices. A successful attack on an embedded device would make sensitive information available to the attacker which could have many damaging effects.

Embedded devices that implement cryptographic algorithms are increasingly susceptible to power and fault analysis-based side-channel attacks [50, 51, 15]. Side-channel attacks may arise when computers and microchips leak sensitive information about the software code that they execute, e.g. through power and heat dissipation or response to faults injected to the hardware. Such information

leaks have been exploited in many commercial systems in the embedded space.

A common strategy for designing countermeasures against power side-channel attacks is using randomization techniques to remove the statistical dependency between the sensitive data and the side-channel leakage [20, 7, 68]. However, this process is both labor intensive and error prone.

Currently, there is a lack of automated tools to formally assess if a countermeasure really is secure. Furthermore, there is no formal method to quantify the actual strength of a countermeasure. Security design errors may therefore go undetected until the physical product is produced and evaluated.

Another closely related problem is that side-channel countermeasures are difficult to design and implement. Although it would be desirable to have a design automation tool that can automatically generate provably secure countermeasures, in reality, robust implementations of such tools do not yet exist.

1.2 Contribution

In this dissertation, I address the challenges facing software developers in producing reliable and secure code for embedded systems. I propose a set of techniques for automatically verifying, as well as automatically synthesizing, software with a focus on embedded control and cryptography software. I also propose innovative new techniques to scale up the proposed program synthesis methods to software code of realistic size and complexity. The program synthesis methods I present are capable of synthesizing code that is more efficient than the code experts in their fields would write. The challenges I explore in this dissertation can be summarized as follows:

- How to automatically tailor existing critical code to embedded systems without degradation in its execution?
- How to automatically fully verify embedded software for reliability and security issues?

- How to synthesize code for embedded devices that is more optimized than that written by experts?
- How to overcome the main drawback of program synthesis and formal methods of unscalability, and synthesize programs that are of practical size?

Altering code to fit a targeted embedded device is not a trivial task, and in most cases needs to be done by an expert. Directly incorporating an algorithm code implementation, that was not written for the targeted platform, into the embedded program could likely lead to errors during the code execution. A serious source of error is overflow/underflow which occurs due to the limited fixed-point processor's register size. These errors are critically important to address because no warnings are given during compilation of the code, and this most probably make a developer believe that the embedded program is safe while in reality it is not. I present in this dissertation an automated method to avoid these errors.

Specifically, I propose a method that automatically modifies a given arithmetic computational code in order to suit the smaller targeted bitwidth of a fixed-point embedded device [26, 27]. To avoid overflow/underflow errors, the method verifies which code instructions are susceptible to the errors then synthesizes another functionally equivalent code implementation that is not vulnerable. The synthesized code would then replace the vulnerable code. By using formal verification it is guaranteed that the new synthesized code is equivalent to the original code but will not cause overflow/underflow errors. By iteratively repeating this process, an implementation that requires the minimum processor bitwidth to execute the code, could be found. The instructions requiring the largest register bitwidth are repeatedly searched for, then equivalent instructions that require a reduced bitwidth are synthesized to replace them. Since the dynamic range is proportional to the bitwidth, the method I present has another advantage of increasing the code dynamic range. This is done by reducing the minimum bitwidth needed to execute the code bellow the actual embedded device available bitwidth.

Using a formal program synthesis method guarantees always finding an optimized code if it exists. The proposed method searches for a valid solution from all possible codes that could be generated

with the specified grammar. Unscalability is a downside for this because the size of the search space is exponentially proportional to the size of the code to be synthesized. I present a solution to this problem by performing static analysis techniques to partition the code into smaller regions. Each region could have a new optimized region synthesized to replace it. The applied static analysis techniques allow maintaining useful information from the complete code and then make use from it during the synthesis of an optimized region. This method overcomes the unscalability disadvantage of program synthesis tools and makes it possible to synthesize practical programs of large size and in reasonable time.

In this dissertation, I also apply program synthesis techniques to other new fields such as embedded cryptography. Securing the sensitive information in cryptographic implementations against malicious attacks has always been a tedious task assigned to cryptographic experts and not to design engineers due to its complications. Although this, still there has been many cases where vulnerabilities were discovered in the cryptographic implementations [42, 16, 10].

Power side-channel attacks is one of the most carried out attacks on embedded devices. By monitoring the device power consumption, an attacker could guess the secret key from a correlation between the secret key value and the power consumption pattern. As a countermeasure against such attacks, the secret key is masked with random generated numbers to reduce the correlation between the secret key and the power consumption pattern. This process is manually done by cryptography experts.

I propose an automated method to verify if the countermeasures implemented are perfectly masked or still have vulnerabilities [31, 30]. In many cases my method finds vulnerabilities in cryptographic code thought to be secure against power side-channel attacks. To improve scalability of this method and to reduce the verification problem time, an incremental procedure is proposed by which the cryptographic code is gradually verified.

The previous state of the art automated verification method proposed by Bayrak *et al.* [10] could only detect if instructions were masked but could not distinguish if the masking was vulnerable or not. I show that the proposed verification procedure [31, 30] is more accurate and faster than the

previous state of the art method.

I also propose a new notion, called Quantitative Masking Strength (QMS), to quantify the resistance of masking countermeasures against attacks [32, 33]. If an instruction result in the cryptographic code is strongly dependent on the value of a sensitive data bit then it is considered weakly masked, while if it is strongly dependent on a generated random number bit then it is considered strongly masked. By performing static analysis on the cryptographic source code, a value between 0 and 1 is returned to describe the degree of the masking strength of the implemented countermeasure. A formula is proposed to relate the computed QMS value to the difficulty of preforming a successfully power side-channel attack.

Furthermore, I propose a new method for synthesizing provably secure masking countermeasures [28]. For the cryptographic implementations that are vulnerable to power side-channel analysis. The method synthesizes a cryptographic code functionally equivalent to the vulnerable code and eliminates all information leakage via power side-channels. It is based on formal program synthesis methods and so guarantees to find a countermeasure implementation if it exists within the specified search space. Results show that synthesized countermeasures can be more optimized than those proposed by cryptographic experts. Furthermore, a partitioned method that combines static analysis with the program synthesis method is proposed to improve scalability.

Fault sensitivity analysis (FSA) is another type of side-channel attack on cryptographic hardware. To launch an attack, a fault is injected during the execution of the cryptographic algorithm, for example by increasing the embedded device clock frequency. To guess the secret key, the attacker exploits the dependency between the intensity of the injected fault at which an execution error starts to occur and the value of the secret key. For example, at a certain increased clock frequency, if the chance for an error to occur during execution is dependent on the values of the secret key bits, then the cryptographic implementation is vulnerable to FSA attacks. The main contributor to this dependency is the difference in paths' propagation delay from the sensitive bits to the output.

The latest proposed method by Ghalaty *et al.* [35] for generating countermeasures against FSA attacks is based on adding delay elements in the paths of the secret key bits to equate all propagation

delays. This method has a drawback of an exponential relation between the number of added gates and the longest path for a sensitive bit. In contrast, I propose a novel method to create countermeasures based on inductive program synthesis [29]. A completely new circuit is generated that is functionally equivalent to the original vulnerable circuit, but constrained to ensure that all delay paths in the implementation are independent of the sensitive data. This method reduces the needed gates, which greatly reduces the synthesized countermeasure circuit size. A partitioned method incorporating static analysis is also proposed to improve the scalability of the proposed method.

To summarize, I propose in this dissertation a set of new innovative techniques to automate the verification and synthesis of embedded systems software and hardware [26, 27, 31, 30, 32, 33, 28, 29]. The presented methods are either first to be proposed in their fields or if previous methods were proposed, our methods surpass the performance of the state of the art methods. Partitioning methods are proposed to overcome the unscalability problems of the formal program synthesis tools. All proposed methods are shown to be effective by applying them on practical benchmarks.

1.3 Organization

This dissertation is organized as follows. First, I begin by introducing the background and notation needed to read this dissertation in Chapter 2. Then, from Chapter 3 to Chapter 7, I present the main research contributions in detail. Specifically, in Chapter 3, I introduce the new method for optimizing fixed-point arithmetic computation software. The method reduces the embedded device required minimum bitwidth to execute an arithmetic computational program. In Chapter 4, I present the new method for verifying whether a cryptographic software is vulnerable to power side-channel attacks. Static analysis and formal verification methods are combined to speed up the verification process. In Chapter 5, I address the importance of having power side-channel countermeasures perfectly masked. I introduce the new notion QMS to quantify the masking strength of an implemented countermeasure. The method statically computes the QMS from the cryptographic

software source code, and reflects the resistance of the implemented countermeasure against an attack. In Chapter 6, I present the new method for synthesizing masking countermeasures for cryptographic software code against power side-channel attacks. A code partitioning method is also introduced to improve the scalability and synthesize masking countermeasures for practical large cryptography programs. In Chapter 7, I present the new method for synthesizing countermeasures against FSA attacks for embedded cryptography hardware. The method applies constraint-based program synthesis methods to synthesize compact circuit countermeasures. Finally, in Chapter 8, I conclude the dissertation by summarizing the contributions and outlining the potential future work.

Chapter 2

Background

In this chapter, we intruduce the needed background to support this dissertation.

2.0.1 Inductive Synthesis

In inductive synthesis, we are concerned with a set x of inputs, a theory T, and a grammar G, which collectively define the design space. The synthesis problem is defined as constructing a function f such that it satisfies a correctness specification P under all possible input conditions. In this context, the theory T and the grammar G are used to restrict the search space. Let y be the set of auxiliary variables that control how the function f is chosen from the pool of candidates in the design space. That is, each valuation of y corresponds to a candidate function $P(x, y)$. At the high level, the synthesis problem can be expressed as a constraint solving problem as follows:

$$\exists y. \forall x.\ \ P(x, y)$$

That is, we want to find a configuration of y such that, for all possible x, the correctness condition $P(x, y)$ holds.

However, since directly solving a quantified logical formula with alternation depth 2 is difficult, the

more pragmatic solution is to use an iterative, counterexample-guided, inductive synthesis. First, we solve the simpler problem $\exists y.\ P(X, y)$. That is, we want to find a candidate configuration c of y such that, at least for some input values X, the correctness condition $P(X, c)$ holds. If we cannot find any solution, then we know there is no solution for the original logical formula. But if we find a candidate solution, the next step is to verify that the solution c is valid under all possible input values. That is, $\forall x.\ P(x, c)$. If this verification step passes, we are done. Otherwise, we need to block this bad solution c, and compute a new candidate c' such that $P(X', c')$ holds for at least some input values X'.

Therefore, the inductive synthesis procedure consists of two subroutines: the synthesis subroutine and the verification subroutine. The synthesis subroutine takes a set of test examples as input, solves $\exists y.P(X, y)$, and returns a candidate function f as output. The verification subroutine takes the candidate function f as input and formally verifies $\forall x.P(x, c)$, i.e., whether it is valid for all possible test examples. Initially, the set of test examples can be empty or consist of some randomly generated input-output pairs. Whenever the candidate function is proved to be invalid, new test examples can be deduced from the counterexamples generated by the verification subroutine. Adding these new test examples back to the synthesis subroutine can guarantee to eliminate the bad candidate in the future.

2.1 Intermediate Representation

The standard C language cannot explicitly represent fixed-point arithmetic operations, so we use an intermediate representation (IR) for the fixed-point programs. We use a combination of the integer C program representation and a separate configuration file, which defines the fixed-point types of all program variables. More specifically, we scale each fixed-point constant (other than the ones used in *shift* operations) to an integer by using the scaling factor 2^m, where m is the number of bits representing the fractional part. For example, a fixed-point with two fractional bits, the representation of a constant with the value of 2.5 will be represented as 10, where $m = 2$.

After each multiplication, a *shift-right* is added to normalize the result so as to match the fixed-point type for the result. For example, $x = c \times z$, where variables x and z and constant c all have the fixed-point type $\langle 1, 8, 3 \rangle$, would be represented as $x = (c \times z) >> 3$.

For each multiplication, we also assign an *accumulate flag*, which can be set by the user to indicate whether the microcontroller has the capability of temporally storing the multiplication result into two registers, which effectively doubles the bit-width of the registers. Many real-world microcontrollers have been designed in this way. Continuing with the same example $x = (c \times z) >> 3$. If the *accumulate flag* is set to 1 by the user, the multiplication node will not be checked for overflow and underflow. Only after the right-shift, will the final result be checked for overflow and underflow.

For all the other operations (`+, -, >>, <<`), we do not rewrite the default IR representation and do not allow the user to set the *accumulate flag*, because most of the microcontrollers do not have double sized registers to temporally store the results of these operations.

In general, the class of programs that we consider here do not have input-dependent control flow, meaning that we can easily remove all the loops and function calls from the code using standard loop unrolling and function inlining techniques. Furthermore, the program can be transformed into a branch-free representation, where the if-else branches are merged. Finally, since all variables are bounded integers, we can convert the program to a purely Boolean program through bit-blasting.

2.2 Side Channel Attacks

Following the notation used by Blömer *et al.* [16], we assume that a sensitive computation $c \leftarrow enc(x, k)$ takes a plaintext x and a secret key k as input and returns a ciphertext c as output. The implementation of function $enc(x, k)$ consists of a sequence of intermediate operations. Each intermediate operation is referred to as function $I_i(x, k)$, where i is the index of that operation.

We assume that the plaintext x and the ciphertext c may be observed by an adversary, whereas the

secret key k is hidden in the computing device. The goal of the adversary is to deduce k based on observing x, c, and the power leakage of the device. Based on the widely used Hamming Weight (HW) and Hamming Distance (HD) models, we assume that the power leakage of the device correlates to the values involved in the sensitive operations $I_1(x,k)\ldots I_n(x,k)$. For example, for two different key values k and k', the power consumption of $k \oplus x$ and $k' \oplus x$ may differ. Such information leaks may be exploited by techniques such as differential power analysis (DPA [51]).

2.3 Leakage Model

A leakage model specifies the amount of information observable during program execution through side channels, such as timing variation, power consumption, and electromagnetic radiation. In power analysis, a simple but effective leakage model, for a single instruction, is the *Hamming Weight (HW)* of the operand. An equally effective leakage model, for two consecrative instructions, is the *Hamming Distance (HD)* of the two operands.

2.4 Masking

To resist power side-channel attacks on cryptographic software, a countermeasure called *masking* can be implemented to eliminate side-channel leaks. It randomizes the instantaneous power consumption to make it statistically independent from the secret data.

For example, when the computation $f(z)$ is a linear function of sensitive variable z in the $\oplus$ domain, meaning that $f(z_1 \oplus z_2) = f(z_1) \oplus f(z_2)$, masking requires no modification of the original implementation of function $f(z)$.

$$f(z \oplus r) \oplus f(r) = f(z) \oplus f(r) \oplus f(r) = f(z)\ .$$

Here, the random bit r is generated internally on the cryptographic device so the adversary cannot

access its value. Due to commutativity of the XOR operation, we can mask z with r before the computation and demask with $f(r)$ afterward.

However, when $f(z)$ is a non-linear function, the implementation of $f(z)$ often needs to be completely redesigned. Depending on the order of attacks to be mitigated, for instance, z may have to be divided into n chunks by using XOR operations with n random bits $r_1 \dots r_n$. Then, each chunk is fed to a newly designed cryptographic function $f'_i(z \oplus r_i, r_i)$, where $1 \leq i \leq n$. At the end, these results are combined to reconstruct $f(z)$ by using XOR operations with another function $f''_i(z \oplus r_i, r_i)$. Consider n=1 as an example, we require the newly designed functions $f'()$ and $f''()$ to satisfy the following constraint:

$$f'(z \oplus r, r) \oplus f''(z \oplus r, r) = f(z) \ .$$

However, the design of such cryptographic functions f' and f'' is a highly creative manual process currently undertaken by experts – it is labor intensive and error prone. Furthermore, even if the masking algorithm is provably secure, bugs introduced during the software coding process may still cause information leaks.

2.5 Perfect Masking

For a pair (x, k) of plaintext and key for the function $enc(x, k)$ and d intermediate computation results $I_1(x, k, r), \dots, I_d(x, k, r)$, where r is an s-bit random variable uniformly distributed in the domain $R = \{0, 1\}^s$, we use $D_{x,k}(R)$ to denote the joint distribution of $I_1, \dots, I_d$. In side channel analysis, d is assumed to be the maximal number of leakage channels accessible to an adversary. If $D_{x,k}(R)$ is statistically independent from k, we say that the function is *order-d* perfectly masked [16]. Otherwise, the function has side channel leaks.

Definition 1. *Given an implementation of function $enc(x, k)$ and a set of its intermediate results*

c1 = x⊕ k∧(r1∧r2)
c2 = x⊕ k∨(r1∧r2)
c3 = x⊕ k⊕(r1∧r2)
c4 = x⊕ k⊕(r1⊕r2)

x	k	r1	r2	c1	c2	c3	c4
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	0
0	1	1	1	1	1	0	1

x	k	r1	r2	c1	c2	c3	c4
1	0	0	0	0	1	1	1
1	0	0	1	0	1	1	0
1	0	1	0	0	1	1	0
1	0	1	1	1	1	0	1
1	1	0	0	0	0	0	0
1	1	0	1	0	0	0	1
1	1	1	0	0	0	0	1
1	1	1	1	0	1	1	0

Figure 2.1: Example of difference between masking and perfect masking.

$\{I_i(x,k,r)\}$, *we say that the function is order-*d *perfectly masked if for all* d*-tuples* $\langle I_1,\ldots,I_d\rangle$,

$$D_{x,k}(R) = D_{x,k'}(R) \quad \textit{for any two pairs } (x,k) \textit{ and } (x,k') \ .$$

As an example, consider Fig. 2.1 where ciphertexts `c1,c2,c3,c4` are results of four different masking schemes for plaintext bit `x` and key bit `k` using random bits `r1` and `r2`. According to the truth tables on the right-hand side, all of these four outputs are logically dependent on `r1,r2`. However, this does not imply statistical independence from the secret `k`. Indeed, `c1,c2,c3` all leak sensitive information. Specifically, for `x` is logical 0, and when `c1` is 1, we know for sure that the secret `k` is also 1, regardless of the values of the random variables. Similarly, when `c2` is logical 0, we know for sure that `k` is also 0. When `c3` is logical 1 (or 0), there is a 75% chance that `k` is logical 1 (or 0). In contrast, `c4` is the only leak-free output because it is statistically independent of `k`. When `k` is logical 1 (or 0), there is 50% chance that `c4` is logical 1 (or 0).

To check for violations of *perfect masking*, we need to decide whether there exists a d-tuple $\langle I_1,\ldots,I_d\rangle$ such that $D_{x,k}(R) \neq D_{x',k'}(R)$ for some (x,k) and (x',k'). Here, the main challenge is to compute $D_{x,k}(R)$.

2.6 Insensitivity

A necessary condition for power side-channel resistance is for all the intermediate computation results of a function to be *insensitive*, as in Bayrak *et al.* [10]. An intermediate result I_i is *sensitive*

o1 = k∧(r1∧r2)
o2 = k∨(r1∧r2)
o3 = k⊕(r1∧r2)
o4 = k⊕(r1⊕r2)

k	r1	r2	o1	o2	o3	o4
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
1	0	1	0	1	1	0
1	1	0	0	1	1	0
1	1	1	1	1	0	1

Figure 2.2: Examples of masking and leakage of secret information.

if it depends on the secret/plaintext and, at the same time, it does not depend on any random variable. According to [10], this dependency analysis is equivalent to computing *don't cares (DCs)* in logic synthesis: If random bit r is a *don't care* of I_i, then I_i does not depend on r. Recall that r is a *don't care* if I_i remains unchanged whether r is set to logical 0 or 1. However, even an *insensitive* I_i may still leak secret information, because *depending on a random bit* does not mean that I_i is statistically independent from the secret.

Figure 2.2 shows an example, where `k` is the secret, `r1` and `r2` are the random variables, and `o1, o2, o3`, and `o4` are the results of four masking schemes. According to the truth table on the right-hand side, all four outputs depend on `r1,r2` and therefore are *insensitive* [10], but three of them still leak secret information. When `o1` is logical 1, we know for sure that the secret `k` is also 1, regardless of the values of the random variables. Similarly, when `o2` is logical 0, we know for sure that `k` is also 0. When `o3` is logical 1 (or 0), there is a 75% chance that `k` is logical 1 (or 0). In contrast, `o4` is the only side-channel resistant output because it statistically independent of k. When `k` is logical 1 (or 0), there is 50% chance that `o4` is logical 1 (or 0).

2.6.1 Fault Sensitivity Analysis (FSA)

Fault attacks are typically conducted by changing the physical environment of the circuit executing a cryptographic algorithm, to introduce logical errors in the otherwise normal computation. These in turn lead to information leaks that are leveraged to deduce the sensitive data. Various fault injection techniques have been used in practice, including varying the voltage of the power supply, the clock frequency, and the temperature of the execution environment. Here we focus on faults

injected by disturbing the external clock, more specifically by aggressively increasing the clock frequency beyond its normal range.

In a digital circuit, the time taken by a signal to change from logical 1 to logical 0 (or vice versa) in response to the inputs often depends on both the circuit structure and the values of its signals. In general, the arrival time of the output signal is determined by the delay of the paths connecting the input signals to this output. In addition, the delay of these paths may depend on the values of the internal signals of the circuit. For the same circuit, but with different values of the internal signals, the delay of these input-to-output paths may be different. This is important because it means the impact of the same injected fault, but under different internal states of the circuit, may be significantly different.

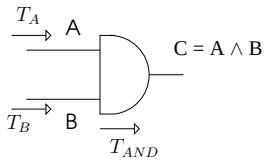


Figure 2.3: The fault sensitivity of an AND gate.

Consider the example AND gate in Fig. 2.3, which has two input signals A and B and an output signal C. Let T_A and T_B be the arrival time of signals A and B, respectively, and T_{AND} be the propagation delay of the AND gate. Consider a scenario where $T_A < T_B$, i.e., signal B has a longer arrival time than signal A. In this case, the time it takes for signal C to stabilize (T_c) depends on the value of signal A. Specifically,

- when A is logical 0, we have $T_C = T_A + T_{AND}$; and
- when A is logical 1, we have $T_C = T_B + T_{AND}$.

In other words, even if we do not know the value of signal A, by observing the difference in the arrival time T_c, we can reliably deduce the sensitive information based on our knowledge of the circuit structure.

It is worth pointing out that such dependency relation between the gate delay and the values of input signals is not unique for the AND gate; other types of logic gates have similar dependency relations.

In general, for a large digital circuit with a wide range of logic gates, the delay along its various input-to-output paths may depend on the values of the signals — that is the main source of vulnerability for Fault sensitivity analysis (FSA) attacks. To successfully launch fault attacks, merely injecting faults into a circuit is not enough. In addition, these faults must be propagated through the circuit to become observable at the output. In practice, the susceptibility of developing a faulty output often depends on the intensity of the faults injected, for example, the degree of over-clocking, since the more intense the faults are, the more likely they will be propagated to the output. The least intensity level under which injected faults becomes observable at the output is referred to as the *critical level*. Due to the dependency between gate propagation delay and values of the signals in the circuit, in general, the *critical level* will be different for different values of the signals.

FSA attacks, in particular, rely on exploiting a dependency between the *critical level* and values of the sensitive signals. Since the time taken by a signal to propagate through the circuit depends on both the circuit structure and values of its internal signals, the attacker needs to have knowledge of the circuit under attack. This is a realistic assumption, since most cryptographic algorithms, along with their implementations, are publicly available.

An FSA attack typically consists of three steps:

1. The attacker injects faults and then measures the critical level of a circuit for a set of N randomly generated plaintexts (test inputs);

2. The attacker computes, using computer simulation, the critical level for each of the N selected plaintexts, together with each possible sensitive data value combination;

3. The attacker performs a correlation analysis between the measured critical level and the simulated critical level for each possible sensitive data value combination.

At the end of the third step, the sensitive data value combination that results in the highest correlation coefficient will be identified. This information will then be used to deduce the sensitive data value from the circuit.

2.6.2 FSA Countermeasures

The necessary condition for carrying out a successful FSA attack is having easily distinguishable fault sensitivity *critical levels* for the different sensitive data value combinations. Among output signals whose arrive time depends on the sensitive data, the greater the difference in their arrival times, the more distinguishable the critical levels, and consequently, the higher the chance attackers will have in successfully deducing the sensitive data. Therefore, the goal of an FSA countermeasure is to disable the aforementioned condition.

All previously proposed countermeasures against FSA attacks [34, 35] rely on adding delay to certain parts of the circuit to make the arrival time of the output signals independent from the sensitive data. Such solution often adds an unnecessarily large number of buffers, which results in larger area and higher power consumption.

Chapter 3

Optimizing Arithmetic Computation in Embedded Software Code

Analyzing and optimizing the fixed-point arithmetic computations in embedded control software is crucial to avoid overflow and underflow errors and minimize truncation errors within the designated input range. Implementation errors such as overflow, underflow, and truncation can lead to degradation of the computation results, which in turn may destabilize the entire system. The conventional solution is to carefully estimate the minimum bit-width required by the software code to run in the error-free mode and then choose a microcontroller that matches the minimum bit-width. However, this can be expensive or even infeasible, e.g., when the microcontroller at hand has 16 bits but the code requires 17 bits.

In many cases, it is possible for the developer to manually reorder the arithmetic computation operations and optimize the code structure to avoid the overflow and underflow errors and to minimize truncation errors. However, the process is labor intensive and error prone. In this chapter, we present a new compiler assisted code transformation method to automate the process. More specifically, we apply inductive synthesis incrementally to optimize the arithmetic computations so that the code can be safely executed on microcontrollers with a smaller bit-width.

The content of this chapter is originally published in [26]

For example, consider the code in Fig. 3.1, where all input parameters are assumed to be in the range $[0, 9000]$. A quick analysis of this program shows that, to avoid overflow, the program must be executed on a microcontroller with at least 32 bits. If it were to run on a 16-bit microcontroller, many of the arithmetic operations, e.g., the subtraction in Line 13, would overflow. In this case, a naive solution is to scale down the bit-widths of the overflowing operations by eliminating some of their least significant bits (LSBs). However, this would decrease the dynamic range, ultimately leading to a large accumulative error in the output.

Our method, in contrast, can reduce the minimum bit-width required to run this piece of fixed-point arithmetic computation code without any loss in accuracy. Our method would take the original C code in Fig. 3.1 and the user-specified ranges of its input parameters, and returns the optimized C code in Fig. 3.2 as output. Our method guarantees that the two programs are mathematically equivalent – if all program variables represent unbounded integers – but the one in Fig. 3.2 requires a smaller bit-width to achieve the same dynamic range. More specifically, the new code can run on a 16-bit microcontroller. Furthermore, our method ensures that the new code does not introduce any additional truncation error. In other words, the new code is always more accurate than the original one.

The optimization in our method is carried out by an SMT solver based *inductive synthesis* procedure, which is customized specifically for efficient handling of fixed-point arithmetic computations. Recent years of have seen a renewed interest in applying inductive synthesis techniques to a wide variety of applications (e.g., [79, 78, 44, 38, 39, 41, 67, 75]). However, a naive application of these techniques would not work due to their limited scalability and large computational overhead. For example, our experience with the Sketch tool [79] shows that, when being applied to synthesizing arbitrary fixed-point arithmetic computations, it does not scale beyond programs with 3-4 lines of code.

The main contribution here is our proposal of an *incremental inductive synthesis* algorithm, where the SMT solver based analysis is carried out only on small code regions of bounded size, one at a time, as opposed to the entire program. This incremental optimization approach allows our method

to scale up to programs of practical size and complexity.

Our new method differs from most of the existing methods for optimizing arithmetic computations in embedded software code. These existing methods, including the recent ones [45, 71], focus primarily on computing the optimal (smallest) bit-widths for all program variables. Instead, our method focuses on re-ordering the arithmetic operations and re-structuring the code, which in turn may lead to reduction in the minimum bit-width. In other words, we are not merely *finding* the minimum bit-width, but also *reducing* the minimum bit-width through code transformation. Due to the use of an SMT solver based search, our method can find the best implementation solution within a bounded search space. This is in contrast to standard compiler optimizations, which are based on matching simple syntactic patterns.

We have implemented our new method in a software tool based on the popular Clang/LLVM compiler framework [21] and the Yices SMT solver [25]. We have evaluated the performance of our tool on a representative set of public domain benchmarks collected from embedded control applications and digital signal processing (DSP) applications. Our results show that the new method can achieve a significant reduction in the minimum bit-width required by the program, and alternatively, a significant increase in the dynamic range.

To sum up, the main contributions here are:

- We propose the first method for incrementally optimizing the linear fixed-point arithmetic computations in C/C++ code via inductive synthesis to reduce the minimum bit-width and increase the error-free dynamic range.
- We implement the new method in a practical software tool based on Clang/LLVM and the Yices SMT solver and demonstrate its effectiveness and scalability on a set of representative embedded control and DSP applications.

The remainder of this Chapter is organized as follows. In Section 3.1, we illustrate our new method by using an example. Then we present the overall algorithm in Section 3.3. We present our inductive synthesis procedure in Section 3.4. The implementation details and experimental results

```
1: int comp(int A,int B,int H,int E,int D,int F,int K) {
2:  int t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12;
3:  t12 = 3 * A;
4:  t10 = t12 + B;
5:  t11 = H << 2;
6:  t9  = t10 + t11;
7:  t6  = t9 >> 3;
8:  t8  = 3 * E;
9:  t7  = t8 + D;
10: t5  = t7 - 16469;
11: t3  = t5 + t6;
12: t4  = 12 * F;
13: t2  = t3 - t4;
14: t1  = t2 >> 2;
15: t0  = t1 + K;
16: return t0;
17:}
```

Figure 3.1: The original C program for implementing an embedded controller.

are given in Section 3.5 and Section 3.6, respectively. We review related work in Section 3.7, and finally give a summary in Section 3.8.

3.1 Motivating Example

We illustrate the overall flow of our method by using the example in Fig. 3.1. Our method takes this program and a configuration file that defines the value ranges of all parameter variables as input, and returns the program in Fig. 3.2 as output. It starts by parsing the original program and constructing an abstract syntax tree (AST). Each program variable in Fig. 3.1 corresponds to a node in the AST. The root node of the AST is the return value of the program. The leave nodes are the input parameters.

The AST is first traversed *forwardly*, from the parameters to the return value, to compute the value ranges. Each value range is a (min, max) pair for representing the minimum and maximum values of the node. They are computed by using a symbolic range analysis [70]. Then, the AST is traversed *backwardly*, from the return statement to the entry statement, to identify the list of AST nodes that may overflow or underflow when using a reduced bit-width. The first overflowing node in Fig. 3.1 is the subtraction in Line 13. Although both `t3` and `t4` can be represented in 16 bits,

```
1: int comp(int A,int B,int H,int E,int D,int F,int K) {
2:  int t0,t1,t3,t4,t5,t6,t8,t12;
3:  int N1,N2,N3,N4,N5,N6,N7,N9,N10;
4:  t12 = 3 * A;
5:  N6  = H;
6:  N10 = t12 - B;
7:  N9  = N10 >> 1;
8:  N7  = B + N9;
9:  N5  = N7 >> 1;
10: N4  = N5 + N6;
11: t6  = N4 >> 1;
12: t8  = 3 * E;
13: N3  = t8 - 16469;
14: t5  = N3 + D;
15: t3  = t5 + t6;
16: t4  = 12 * F;
17: N2  = t4 >> 2;
18: N1  = t3 >> 2;
19: t1  = N1 - N2;
20: t0  = t1 + K;
21: return t0;
22:}
```

Figure 3.2: Optimized C code for implementing the same embedded controller.

the subtraction may produce a value that requires more bits.

For each AST node that may overflow or underflow, we carve out some neighboring nodes to form a *region for optimization*. The region should include the node, its parent node, its child nodes, and optionally, the transitive fan-in and fan-out nodes up to a bounded depth. The region size is limited by the capacity of the subsequent inductive synthesis procedure. For the subtraction in Line 13, if we bound the region size to 2 AST node levels, the extracted region would include the right-shift in Line 14 (parent node).

The extracted region is then subjected to an inductive synthesis procedure, which generates a new region that is mathematically equivalent to the extracted region but *overflow/underflow free*. For Line 13 in Fig. 3.1, the extracted region and the new region are shown side by side as follows:

```
t2 = t3 - t4;                   N2 = t4 >> 2;
t1 = t2 >> 2;          -->      N1 = t3 >> 2;
...                             t1 = N1 - N2;
```

That is, instead of applying right-shift to the operands after subtraction, we apply right-shift first. Because of this, the new region needs a smaller bit-width to avoid overflow.

However, it is important to note that the new region is not always better, because sometimes it may introduce additional truncation errors. Consider `t3 = 2, t4 = -2` as a test case. We have `(t3 - t4)>>2 = 1` and `(t3>>2 - t4>>2) = 0`. The new region may lose precision if the two least significant bits (LSBs) of `t3,t4` are not zero. Therefore, an integral part of our new optimization method is to synthesize a new region only when it does not introduce additional truncation errors that affect the final output. For this reason, we perform a truncation error margin analysis to identify, for each AST node, the number of LSBs that are immaterial in deciding the final output. For Line 13, the two LSBs of both `t3` and `t4` can be ignored. Therefore, the truncation error introduced above will not affect the final output.

Since the new region is strictly better, the original AST is updated by replacing the extracted region with the new region. After that, our method continues to identify the next node that may overflow or underflow. The entire procedure terminates when it is no longer possible to optimize any further. In the remainder of this section, we provide a more detailed description of the subsequent optimization steps.

After optimizing the subtraction in Line 13, the next AST node that may overflow is in Line 10. The extracted region and the new region are shown side by side as follows:

```
t7 = t8 + D;                    N3 = t8 - 16469;
t5 = t7 - 16469;      -->       t5 = N3 + D;
```

Our analysis shows that variables `t8, D` and constant `16469` all have zero truncation error margins. The new region does not introduce any additional truncation error. Therefore, the original AST is updated with the new region.

The next AST node that may overflow is in Line 6. The extracted region and the new region are shown as follows:

```
t9 = t10 + t11;                 N6 = t11 >> 2;
t6 = t9 >> 3;                   N5 = t10 >> 2;
...                   -->       N4 = N5 + N6;
...                             t6 = N4 >> 1;
```

The truncation error margins are 2 for `t10` and 2 for `t11`. Therefore, the truncation error margin for `t9` is 2, meaning that the two LSBs may be ignored. Since the new region is strictly more accurate, the original AST is again updated with the new region.

The next AST node that may overflow is in Line 4. The extracted region and the new region are shown as follows:

```
t10 = t12 + B;                  N10 = t12 - B;
N5  = t10 >>2;                  N9  = N10 >> 1;
...                    -->      N7  = B + N9;
...                             N5  = N7 >> 1;
```

Notice that this extracted region consists of a node that is the result of a previous optimization step. The truncation error margins are 0 for `t12` and 0 for `B`. The new code region does not suffer from the same truncation error that would be introduced by `N5 = (B>>2 + t12>>2)`, because the truncation error is not amplified while being propagated to the final result. Instead, it is compensated by the addition of `B`.

The last node that may overflow is in Line 5 of Fig. 3.1. The extracted region and the new region are shown as follows:

```
t11 = H << 2;
N6  = t11 >> 2;        -->      N6 = H;
```

By now, all arithmetic operations that may overflow are optimized. The new program in Fig. 3.2 can run on a 16-bit microcontroller while still maintaining the same accuracy as the original program running on a 32-bit microcontroller. Another way to look at it is that, if the optimized code were to be executed on the original 32-bit microcontroller, it would have a significantly larger dynamic range.

3.2 Fixed-point Notations

We follow [88] to represent the fixed-point type by a tuple $\langle s, N, m \rangle$, where s is the sign bit (1 for signed and 0 for unsigned), N is the total number of bits or the *bit-width*, and m is the number of bits representing the fractional part. The number of bits representing the integer part is $n = (N - m)$. Different variables and constants in the original program are allowed to have different bit representations, but all of them should have the same bit-width N.

Signed numbers are represented in the standard *two's complement* form. For an N-bit number α, which is represented by bit-vector $x_{N-1}\ x_{N-1}$ ... x_0, its value is defined as follows:

$$\alpha = \frac{1}{2^m} \times \left(-2^{N-1} x_{N-1} + \sum_{i=0}^{N-2} 2^i x_i \right) ,$$

where x_i is the value of the i^{th} bit. The value of α lies in the range $[-2^n, 2^n - 2^{-m}]$. If a number to be represented exceeds the maximum value, there will be an *overflow*. If a number to be represented is less than the minimum value, there will be an *underflow*. If the number to be represented requires more designated fractional bits than m, there will be a *truncation error*. The maximum error caused by truncation is 2^{-m}.

We define the *step* of a variable or a constant as the number of consecutive LSBs that always have the value zero. For example, the number 1024 has a *step* 9, meaning that nine of the LSBs are zero. On the other hand, the number 3 has a *step* 0.

3.3 Overall Algorithm

The overall flow of our method in shown in Algorithm 1. The input includes the original program and the value ranges of all the parameter variables. First, we invoke COMPUTERANGES to compute the value ranges of all non-leave AST nodes. Then, we invoke COMPUTEIGNOREBITS to compute the truncation error margins (LSBs whose values can be ignored) for all AST nodes. Finally, we

compute the bit-width ($bw1$) required by the original program to run within the given input range.

Algorithm 1 Optimizing the program within its input range.

```
 1: OPTIMIZEPROGRAM (prog, p_ranges) {
 2:   ranges ← COMPUTERANGES(prog, p_ranges);
 3:   ig_bits ← COMPUTEIGNOREBITS(prog);
 4:   bw1 ← COMPUTMINBITWIDTH(prog, ranges);
 5:   while (true) {
 6:     bw2 ← bw1 − 1;
 7:     for each (Node n ∈ prog that may overflow or underflow) {
 8:       reg ← EXTRACTREGION(prog, n);
 9:       new_reg ← SYNTHESIZE(reg, bw1, bw2, ranges, ig_bits);
10:       if (new_reg does not exist) break;
11:       REPLACEREGION(prog, reg, new_reg);
12:     }
13:     bw1 ← bw2;
14:   }
15:   return prog;
16: }
```

After the bit-width of the original program ($bw1$) is determined, we enter the while-loop to iteratively optimize the program. In each iteration, we try to reduce the bit-width from $bw1$ to $bw2$. The loop terminates as soon as a call to the inductive synthesis procedure fails to return the new region.

Within each loop iteration, we search for all nodes that may overflow or underflow when the new bit-width ($bw2$) is used. We process these nodes in a breadth-first search (BFS) order, i.e., from the return value of the program to the parameter variables. For each node, we invoke EXTRACTREGION to extract a neighboring region and then invoke the inductive synthesis procedure. If successful, the inductive synthesis procedure would return a new region, which is mathematically equivalent to the extracted region but would not overflow or underflow. It also ensures that the new region would not introduce additional truncation error. After the new region is found, we use it to replace the extracted region in the program.

3.3.1 Region for Optimization

The size of the extracted *region* affects both the effectiveness and the computational overhead of the inductive synthesis procedure. The minimum extracted region should include the erroneous node and its parent node. Since we follow the BFS order, the parent node must have no overflow or underflow since it is already tested negative or optimized. Since in the original program, the parent operation restores the overflowed value created in the overflowing node back to the normal operation range, when the parent node is included in the region, it is more likely to find an alternative implementation that is more accurate than the extracted region.

In general, a larger extracted region allows for more opportunity to find a suitable new region. The maximum extracted region – if it were not for the limited capability of the SMT solver – would be the entire input program. This is equivalent to applying inductive synthesis tools such as Sketch [79, 78] to the entire program, provided that the fixed-point arithmetic optimization problem is modeled in the Sketch input language. In practice, however, such a monolithic optimization approach seldom works. Indeed, our experience with the Sketch tool shows that it cannot scale beyond arbitrary fixed-point arithmetic computation code of 2-3 lines.

Therefore, in addition to implementing our customized inductive synthesis procedure, which can efficiently handle fixed-point arithmetic computations, we also bound the size of the extracted region so that inductive synthesis is applied only in the context of incremental optimization. More specifically, the extracted region is bounded to an AST with at most 5 node levels, which represents up to 63 AST nodes.

3.3.2 Truncation Error Margin

We compute the *step* and the *ignore bits* for all AST nodes recursively. During the optimization process, the calculated *step* will be used to compute the truncation error margin (the LSBs whose values can be ignored). Our method will leverage the truncation error margins to obtain the best possible optimization results.

First, we determine the *step* of each leave node based on the definition in Chapter 2. In general, the *step* may originate from a *shift-left* operation, a *step* in a parameter variable, or a *step* in a constant.

We compute the step of each internal AST node as follows:

- $step(x * y) = step(x) + step(y)$;
- $step(x + y) = min(step(x), step(y))$;
- $step(x - y) = min(step(x), step(y))$;
- $step(x << c) = step(x) + c$;
- $step(x >> c) = max(step(x) - c, 0)$.

The *ignore bits* are those consecutive LSBs that can be ignored during the optimization process. If these bits are truncated in the new region, for example, no error will occur in its output. By taking into account these bits in the optimization process, we are able to synthesis more compact new regions.

To clarify this, consider the example in Fig. 3.3, where the extracted region is shown inside the dotted box. We start by analyzing the AST to determine the *step* of each node. For the purpose of optimizing the extracted region, we need to know the *step* of the region's inputs, which are the nodes labeled as a and b. Due to the shift-left operations, the *step* of a is 4, while the *step* of b is 3. Considering these *step* values, we determine that, when optimizing the extracted region, we have a "credit" of 3 bits to ignore. In other words, we have the freedom to truncate up to 3 consecutive LSBs of the two inputs (a and b) without decreasing the accuracy of the result. Because of this, we are able to synthesize the new region as shown in Fig. 3.4.

Notice that, even if we do not consider the ignore bits, our method can still synthesize a new region to remove the overflowing node in the above example. However, in such case, the extracted region would have to be larger. That is, the extracted region would need to include all the AST nodes in Fig. 3.3. The synthesized new region would include all the AST nodes in Fig. 3.4. However, this would also lead to a significantly longer synthesis time.

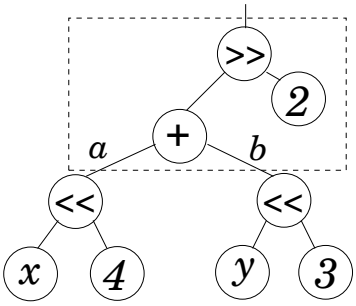


Figure 3.3: The extracted region.

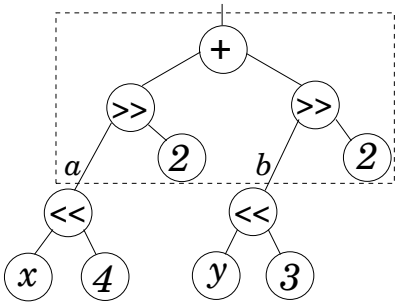


Figure 3.4: The synthesized region.

3.4 The Inductive Synthesis Procedure

At the high level, our inductive synthesis procedure consists of two steps: (1) run a set of test cases on the extracted region, and based on the results, generate a new region that is equivalent to the extracted region at least for the set of test cases; (2) check if the two regions are equivalent in the full input range. If they are not equivalent, block this region (bad solution) and try again.

Algorithm 2 shows the pseudo code of our synthesis procedure, which computes a new region (new_reg) of bit-width $bw2$, such that it is equivalent to the original region (reg) of bit-width $bw1$, under the value ranges specified in $ranges$ while considering the truncation error margins specified in ig_bits. The procedure starts by initializing *blockedRegions* and *testSet* to empty sets, where *testSet* consists of the test cases used for inductively generating (guessing) a new region, and *blockedRegions* consists of the previously explored regions that fail the equivalence check. The procedure initializes the *size* of the new region to 1, and then enters the while-loop to iteratively search for a new region of increasingly larger size. When *size* exceeds a predetermined bound, we have proved that no solution exists in this search space.

Subroutine GENREGION uses an SMT solver to inductively generate a new region, based on the test examples in *testSet* and the already explored regions in *blockedRegions*. Subroutine COMPDIFF formally checks the equivalence of the extracted region (reg) and the new region (new_r), and returns a concrete test if they are not equivalent.

Algorithm 2 Inductively synthesizing the new code region.

```
1: SYNTHESIZE (reg, bw1, bw2, ranges, ig_bits) {
2:   blockedRegions ← { };
3:   testSet ← { };
4:   size ← 1;
5:   while (size < MAX_REGION_SIZE) {
6:     new_r ← GENREGION(reg, bw1, bw2, size,blockedRegions, testSet);
7:     if (new_r exists) {
8:       test ← COMPDIFF(reg, new_r, bw1, bw2, ranges, ig_bits);
9:       if (test exists) {
10:        blockedRegions ← blockedRegions ∪{new_reg};
11:        testSet ← testSet ∪{test};
12:      }
13:      else
14:        return new_r;
15:    }
16:    else
17:      size ← size + 1;
18:  }
19:  return no_solution;
20: }
```

3.4.1 Constructing the New Region Skeleton

First, we generate a *skeleton* of the new region, which is a generalized AST capable of representing any linear arithmetic equation up to a bounded size. In this AST, each leave node is either a constant or any of the set of input variables of the extracted *region*. Each internal node is any of the linear arithmetic operations (`*, +, -, >>, <<`). The root node is the result of the arithmetic computation and should compute the same result as the output node in the extracted region. Fig. 3.5 shows an example *skeleton* of 7 AST nodes. Here, *Op* represents any binary arithmetic operator and $V|C$ represents a leave node (either a variable or a constant).

For each AST node in the *skeleton*, we assign an auxiliary variable called the *selector*, whose value determines the node type. For example, a leave node (`LNode1`), which may be variable `V1`, variable `V2`, or constant `C1`, is represented as follows:

```
((LNode1 == V1) && (sel1 == 0) ||
 (LNode1 == V2) && (sel1 == 1) ||
 (LNode1 == C1) && (sel1 == 2))
```

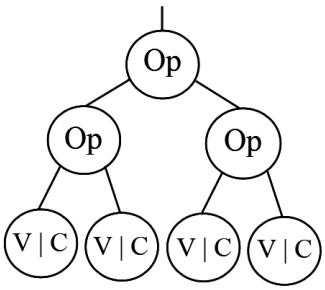


Figure 3.5: Skeleton of 7 AST nodes.

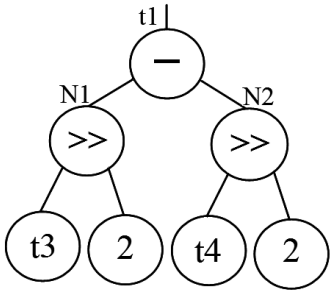


Figure 3.6: Synthesized new region.

where the integer value of selector variable `sel1` ranges from 0 to 2. Similarly, a generalized internal node (`INode3`), which may be an addition or a subtraction of `LNode1` and `LNode2`, is represented as follows:

```
((INode3 == LNode1+LNode2) && (sel2 == 0) ||
 (INode3 == LNode1-LNode2) && (sel2 == 1))
```

where the integer value of selector variable `sel2` ranges from 0 to 1. The actual node types in the *skeleton* are determined later, when we encode the skeleton into an SMT formula, and then call the SMT solver to find a set of suitable values for all these selector variables.

3.4.2 Inductively Generating the New Region

To generate the new region, we need a representative set of test cases for the extracted region. These are test values for the input variables of the region, and should include both the corner cases and the intermediate values. Since the arithmetic computations are linear, we construct the corner cases by including the minimum and maximum values of all input variables as defined in *ranges*. Additional test values are generated by taking semi-equidistant intermediate values between values in the corner cases.

We create an SMT formula Φ such that Φ is satisfiable iff the resulting new region – induced by a satisfying assignment to all *selector* variables – is mathematically equivalent to the extracted region, but does not overflow or underflow.

$$\Phi = \Phi_{reg} \wedge \Phi_{skel} \wedge \Phi_{sameI} \wedge \Phi_{sameO} \wedge \Phi_{tests} \wedge \Phi_{blocked},$$

where the subformulas are defined as follows:

- Extracted region (Φ_{reg}): It encodes the transition relation of the extracted region by using bit-vector arithmetic, where the bit-width is *bw1*.
- New region skeleton (Φ_{skel}): It encodes the transition relation of the skeleton by using bit-vector arithmetic, where the bit-width is *bw2*.
- Same input values (Φ_{sameI}): It asserts that the input variables of the two regions must share the same values.
- Same output value (Φ_{sameO}): It asserts that the output variables of the two regions must have the same value.
- Test cases (Φ_{tests}): It asserts that the input variables must adopt concrete values from the given test cases.
- Blocked solutions ($\Phi_{blocked}$): It asserts that the *selector* variables should not take values that represent any previously explored (bad) solution.

If Φ is unsatisfiable, no solution exists in the bounded search space. In this case, we need to increase the *size* of the *skeleton* and try again. If Φ is satisfiable, we have computed a candidate new region. As an example, consider the first extracted region in Section 3.1. The new region generated from the skeleton in Fig. 3.5 is shown in Fig. 3.6.

3.4.3 Checking the Equivalence of the Regions

The candidate new region is guaranteed to be equivalent to the extracted region over the given set of test cases. However, they may not be equivalent over the full input range. Therefore, the next step is to formally verify their equivalence over the full input range. Toward this end, we create another SMT formula Ψ, which is satisfiable iff the two regions are *not* equivalent; that is, if there exists a test case that can differentiate them. Formula Ψ is defined as follows:

$$\Psi = \Phi_{reg} \wedge \Phi_{new\_reg} \wedge \Phi_{sameI} \wedge \Phi_{diffO} \wedge \Phi_{ranges} \wedge \Phi_{ig\_bits},$$

where the subformulas are defined as follows:

- New region (Φ_{new_reg}): It encodes the transition relation of the candidate new region in bit-vector arithmetic, where the bit-width is *bw2*.
- Different output values (Φ_{diffO}): It asserts that the output variables of the two regions have different values.
- Value ranges (Φ_{ranges}): It asserts that all input variables should stay within their pre-computed value ranges. We are not interested in checking the equivalence of the two regions outside the designated value ranges.
- Ignore bits (Φ_{ig_bits}): It asserts that the LSBs as specified in the ignore bits should all be set to zero. This allows us to ignore the differences between the two regions for LSBs within the truncation error margins.

If Ψ is unsatisfiable, it means that the two regions are mathematically equivalent within the given input range and under the consideration of the ignore bits.

If Ψ is satisfiable, the candidate new region is not correct. In this case, we add it to the *blockedRegions*, and then try again. The blocking of an incorrect solution follows the counter-example guided inductive synthesis algorithm [79, 80], where the blocked solutions are encoded as an additional

constraint in the SMT formula, by adding an extra pair of extracted *region* and new region *skeleton* with the blocked assignment to *selector* variables. It ensures that, when the SMT solver is invoked again to find a candidate new region, the same solution will not be returned.

3.5 Implementation

We have implemented our new method in a software tool for optimizing the C/C++ code of embedded control and DSP applications based on the Clang/LLVM compiler framework [21] and the Yices SMT solver [25]. Our tool has two modes: the whole-program optimization mode and the incremental optimization mode. The two modes differ only in the size bound imposed on the extracted region.

When the bound is set to an arbitrarily large number, our tool runs in the whole-program optimization mode. This makes it somewhat comparable to the popular inductive synthesis tool called Sketch [79, 80], provided that our new region *skeleton* is carefully modeled in the Sketch input language, with the *selector* variables defining the "integer holes" for Sketch to fill. Before implementing our own inductive synthesis procedure, we have explored this approach. However, it turns out to be not scalable: synthesizing a new region with a size bound of more than 2 would cause Sketch to quickly run out of the 4 GB memory. We believe that there are two reasons for this. First, the performance of Sketch is not optimized for handling arbitrary combinations of linear fixed-point arithmetic computations. Second, inductive synthesis, in general, may not be able to scale up to arbitrarily large arithmetic computation programs.

Due to the scalability problem encountered by using Sketch, we have implemented our own inductive synthesis procedure directly using the Yices SMT solver, which is designed for efficient handling of fixed-point arithmetic operations, e.g., by designing SMT encoding schemes for exploiting the AST structures encountered in this type of applications. Our experimental evaluation shows that the new procedure is significantly more efficient than Sketch. Instead of a size bound of 2, it now can routinely optimize the *skeleton* with a size bound of 5 (representing up to 63 AST

nodes). Nevertheless, this improvement alone is not sufficient for supporting the whole-program optimization.

Instead, we propose an incremental optimization method that applies inductive synthesis only to individual regions of a bounded size. More specifically, we have set the maximum bound for *shift-right* and *shift-left* operations to 4, and the maximum level of AST nodes in the new region skeleton to 5. By incrementally optimizing one extracted region at a time, our method is able to avoid the scalability bottleneck imposed by the SMT solver, and therefore can be applied to programs of practical size and complexity.

3.6 Experimental Results

We have evaluated our tool on a set of public domain benchmark examples. The experiments are designed to answer the following three questions:

- How much can our method reduce the minimum bit-width required for the program to run in the given input range?
- How much can our method increase the dynamic range of the program for the given bit-width?
- If both the original and the optimized programs are forced to run with a reduced bit-width, what is the difference between their fixed-point specific implementation errors?

3.6.1 Benchmarks

Our benchmark includes a set of public domain C programs for embedded control and DSP applications. They come from various sources including papers, textbooks, and the output of code generation tools. The sizes of the programs range from 21 lines of code (LoC) to 131 lines, with an average LoC of 79. The number of fixed-point arithmetic operations on average is 58. For the

kind of cyber-physical systems (CPS) software targeted by our new method, these are programs of realistic size and complexity.

Table 3.1 shows the statistics of each benchmark example, including the name, the LoC, and the number of arithmetic operations.

Table 3.1: Statistics of the benchmark C programs.

Name of the Benchmark	Line of Code	Arithmetic Operations
Sobel Image filter (3x3)	42	28
Bicycle controller	37	27
Locomotive controller	42	38
IDCT (N=8)	131	114
Control. Impl.	21	8
Diff. image filter (5x5)	131	77
FFT (N=8) (no DC component)	112	82
IFFT (N=8)	112	90

The first test case, taken from [69], is a 3x3 Sobel digital filter that is widely used in image processing applications. The second test case, taken from [71], is a bicycle controller optimally synthesized for a custom-designed microprocessor with double-sized internal registers. The third test case is a locomotive controller generated by using Fixed Point Advisor and Real Time Workshop of the Matlab toolkit [57]. The fourth test case, taken from [49], is an inverse discrete cosine transform (IDCT), which is widely used in mobile communication and image compression applications. The fifth test case is the fixed-point version of a control rule implementation from [57]. The sixth test case is a 5x5 kernel sized difference image filter [18]. The seventh test case is a fast Fourier transform (FFT) implementation, where the floating-point version was taken from [86] and then converted to fixed-point, by changing all `double` variables into `int` variables without modifying or reordering any of its instructions. The eighth test case is the inverse fast Fourier transform (IFFT) for test case 7. None of the benchmarks was modified from their original forms in any way to give performance advantage to our method.

All experiments were conducted on a machine with a 3.4 GHz Intel i7-2600 CPU, 3.3GB of RAM, and 32-bit Linux.

Table 3.2: Increase in the overflow/underflow free input range.

benchmark	bit	original	optimized	%
Sobel Image	32	[0, 16320]	[-65536, 49152]	602
Bicycle	32	[-3.4*10^8, 3.4*10^8]	[-1.0*10^9, 1.0*10^9]	194
Locomotive	64	[-8.7*10^{18}, 8.7*10^{18}]	[-9.2*10^{18}, 9.2*10^{18}]	5
IDCT	32	[0, 1.5*10^6]	[0, 2.1*10^6]	40
Controller	32	In1 [0, 5.0*10^8]	In1 [-0, 6.6*10^8]	32
		In2 [-5.0*10^8, 0]	In2 [-6.6*10^8, 0]	32
		In3 [-5.0*10^8, 0]	In3 [-6.6*10^8, 0]	32
Diff. Image	32	[0, 1.3*10^8]	[0, 2.1*10^9]	1515
FFT (N=8)	32	[0, 32736]	[0, 32736]	0
IFFT (N=8)	32	[0, 2.6*10^8]	[0, 5.3*10^8]	103

Table 3.3: Increase in the overflow/underflow free output range.

benchmark	bit	original	optimized	%
Sobel Image	32	[0, 16320]	[-49184, 65504]	602
Bicycle	32	[-5.3*10^8, 5.3*1^8]	[-1.5*10^9, 1.5*10^9]	194
Locomotive	64	[-3.6*10^{18}, 5.0*10^{18}]	[-3.9*10^{18}, 5.2*10^{18}]	5
IDCT	32	[-1.4*10^6, 2.9*10^8]	[-1.9*10^7, 3.9*10^8]	40
Controller	32	[0, 1.0*10^9]	[0, 1.4*10^9]	32
Diff. Image	32	[0, 1.3*10^8]	[-1.0*10^9, 1.1*10^9]	1515
FFT (N=8)	32	[25600, 25600]	[25600, 25600]	0
IFFT (N=8)	32	[-1.3*10^8, 2.6*10^8]	[-2.6*10^8, 5.3*10^8]	103

3.6.2 Results

First, we show that there is a significant increase in the input/output range from the original program to the optimized program, when they both use the original bit-width. Tables 3.2 and 3.3 show the results. Column 1 shows the name of the benchmark. Columns 2 and 3 show the input (output) ranges of the original program and the optimized program, respectively. Column 4 shows the percentage of the range increase. The increase in input (output) range spans from 0% to 1515%, with an average increase of 307%. The increase is due to the removal of the overflowing and underflowing nodes in the original program. As a result, the output range is also increased. Together, they lead to a significant increase in the dynamic range of the entire application.

Second, we show that there is a significant decrease in the minimum bit-width required for the program to run without overflow/underflow errors for the given input range. The experimental

Table 3.4: Increase in the minimum and average bit-widths.

Name of Benchmark	Original (bit-width)		Optimized (bit-width)	
	Minimum	Average	Minimum	Average
Sobel image filter (3x3)	17	10.26	15	6.67
Bicycle controller	18	14.47	16	14.16
Locomotive controller	33	29.41	32	29.32
IDCT (N=8)	20	16.29	19	16.38
Control. Impl.	17	15	16	14.67
Diff. image filter (5x5)	17	11.11	13	8.09
FFT (N=8)	18	7.32	16	6.95
IFFT (N=8)	17	7.11	16	7.26

results are shown in Table 3.4. Column 1 is the name of the benchmark. Column 2 is the minimum bit-width of the original program to avoid overflow and underflow, and Column 3 is the average bit-width for all program variables. Column 4 is the minimum bit-width of the new program to avoid overflow and underflow, and Column 5 is the average bit-width for all program variables.

Our results show that the bit-width reduction spans from 1 bit to 4 bits. Consider the Sobel Image filter as an example. The minimum bit-width required to run the original program is 17 bits. After optimization, it is reduced to 15 bits. This is significant, because now the code can be executed on a 16-bit microcontroller instead of a 32-bit microcontroller, which is often significantly cheaper.

To further illustrate the benefit of our new method, consider the maximum error bound in a scaled-down version of the original program in order to downgrade the hardware from 32-bit to 16-bit, or from 64-bit to 32-bit. Table 3.5 shows the comparison between the optimized program and a scaled-down version of the original program. Column 1 is the name of the benchmark. Column 2 is the scaling level. Columns 3 and 4 are the maximum relative errors of the original program and the optimized program, respectively. Our results show that the optimized programs have smaller errors in all test cases.

We also show, in Table 3.6, the statistics of running our optimization method. Column 1 is the name of the benchmark. Column 2 is the number of lines optimized by the incremental inductive synthesis procedure in the original program. Column 3 is the total execution time by our method. The data show that, by using incremental synthesis, we have kept the overall runtime down. In

Table 3.5: Decrease in the maximum relative error.

Benchmark	Scaling	Error original	Error optimized
Sobel Image filter (3x3)	32-b $\rightarrow$ 16-b	$3.1 * 10^{-2}$	0.0
Bicycle controller	32-b $\rightarrow$ 16-b	$3.5 * 10^{-4}$	$2.0 * 10^{-4}$
Locomotive controller	64-b $\rightarrow$ 32-b	$2.9 * 10^{-8}$	$1.5 * 10^{-9}$
IDCT (N=8)	32-b $\rightarrow$ 16-b	$9.2 * 10^{-3}$	$1.8 * 10^{-5}$
Control. Impl.	32-b $\rightarrow$ 16-b	$5.2 * 10^{-4}$	$2.9 * 10^{-4}$
Diff. image filter (5x5)	32-b $\rightarrow$ 16-b	$1.2 * 10^{-2}$	$2.5 * 10^{-3}$
FFT (N=8)	32-b $\rightarrow$ 16-b	$8.1 * 10^{-2}$	$4.4 * 10^{-3}$
IFFT (N=8)	32-b $\rightarrow$ 16-b	$8.4 * 10^{-2}$	$3.2 * 10^{-}2$

Table 3.6: Statistics of the incremental optimization process.

Name of the Benchmark	Num. Optimized Lines	Total Time
Sobel Image filter (3x3)	22	2s
Bicycle controller	2	5s
Locomotive controller	1	5m 41s
IDCT (N=8)	3	2.7s
Control. Impl.	1	46s
Diff. image filter (5x5)	23	10s
FFT (N=8)	14	1m9s
IFFT (N=8)	1	4s

fact, it is no longer directly dependent on the program size, but more on the number of extracted regions and the time spent on optimizing each region.

3.7 Related Work

Our new method incrementally optimizes the fixed-point arithmetic computations in an embedded software program with the objective of reducing the minimum bit-width through code transformation, without changing the computational accuracy. The core synthesis routine in our method follows the same counter-example guided inductive program synthesis paradigm pioneered by Sketch [79, 78]. However, our method is significantly different in that it has an implementation that is designed for more efficiently handle linear fixed-point arithmetic computations. Furthermore, we apply inductive synthesis incrementally to regions of a bounded size, one at a time, as opposed to the entire program.

Gulwani *et al.* [39] propose a method for synthesizing bit-vector programs from a linear reference code by leveraging a set of user defined library functions. Their method does not use incremental inductive synthesis, and the largest synthesized code reported in their paper has 16 lines of code, for which their tool takes over 45 minutes. Jha *et al.* [44] use the same symbolic encoding as in [39] but replace the logical specification of the desired program by an input-output oracle.

The SCIDUCTION tool implemented by Jha [45] can automatically synthesize a fixed-point arithmetic program from the floating-point arithmetic code. However, the focus of this tool is solely on *finding* the smallest possible bit-width and *choosing* the best fixed-point representation for each program variable. They have not attempted to change the code structure or synthesize completely new code for the purpose of *reducing* the minimum bit-width.

Another closely related work is the linear fixed-point optimization method proposed in [71], which relies on using a Mixed Integer Linear Programming (MILP) solver to minimize the error bound by changing the fixed-point representation of the program. Again, their method can only optimize the bit-vector representations of the program variables, but do not change the structure of the original code or synthesize new completely new code in order to reduce the bit-width.

Darulova *et al.* [23] proposed a method for compiling real-valued arithmetic expressions to fixed-point arithmetic programs to minimize the discrepancy between the fixed-point values and the real values. Their method uses genetic programming, which mutates the order of the original arithmetic expressions to find better fixed-point representations. The method differs from ours in three aspects. First, their method takes a real-valued expression in Matlab format as input and returns a fixed-point arithmetic program as output whereas our method transforms an existing fixed-point C program into another fixed-point C program – this also makes experimental comparison of the two approaches difficult to conduct. Second, their method relies on genetic programming, which consists of random mutation and filtering of the mutants, whereas our methods relies on exhaustive search via an SMT solver. Third, their method does not employ incremental inductive analysis, which is one of the main contributions of our work.

Our new method is also related to the various superoptimization techniques that are becoming

popular in compilers in recent years [46, 8, 74]. Superoptimizers are more powerful than conventional compiler based optimizations that rely on matching known code patterns and then applying predetermined transformation rules. In contrast, superoptimizers often perform a more involved search in the implementation space of a set of valid instruction sequences, for example, to optimize performance-critical inner loops. However, to the best of our knowledge, there has not been any existing superoptimizer that can be used to increase the error free dynamic range, or to minimize the minimum bit-width, of fixed-point arithmetic computations in embedded C programs.

3.8 Summary

We have presented a new method for incrementally optimizing the linear fixed-point arithmetic computations of an embedded software program via code transformation to reduce the required bit-width and to increase the dynamic range. Our method is based on judicious application of an SMT solver based inductive synthesis procedure to code regions of bounded size. We have implemented our method in a software tool and evaluated it on a set of representative embedded programs. Our results show that the new method can significantly reduce the bit-width and handle programs of realistic size and complexity.

Chapter 4

Detecting Power Side-Channel Leaks in Cryptographic Software

Security analysis of the hardware and software systems implemented in embedded devices is becoming increasingly important, since an adversary may have physical access to such devices and therefore can launch a whole new class of side-channel attacks, which utilize secondary information resulting from the execution of sensitive algorithms on these devices. For example, the power consumption of a typical embedded device executing the instruction `tmp=text`$\oplus$`key` depends on the value of the secret `key` [56]. This value can be reliably deduced using a statistical method known as *differential power analysis* (DPA [51, 81]). In recent years, many commercial systems in the embedded space have shown weaknesses against such attacks [66, 59, 7].

A common mitigation strategy against such attacks is using randomization techniques to remove the statistical dependency between the sensitive data and the side-channel information. This can be done in multiple ways. Boolean masking, for example, uses an XOR operation of a random number r with a sensitive variable a to obtain a masked (randomized) variable: $a_m = a \oplus r$ [7, 68]. Later, the sensitive variable can be restored by a second XOR operation with the same random number: $a_m \oplus r = a$. Other randomization based countermeasures have used additive masking ($a_m = a + r \ mod \ n$), multiplicative masking ($a_m = a * r \ mod \ n$), and application-specific code

The content of this chapter is originally published in [31]

transformations such as RSA blinding ($a_m = ar^e \ mod \ N$).

However, designing and implementing such side-channel countermeasures are labor intensive and error prone, and currently, there is a lack of formal verification tools to evaluate how secure a countermeasure really is. Software countermeasures are particularly challenging to design, since the source of the information leakage is not the cryptographic software but the microprocessor hardware that executes the software. From the perspective of average software developers – who may not know all the architectural details of the device – it is difficult to predict the myriad possible ways in which side-channel information may be leaked. Furthermore, bugs in implementation can also break an otherwise secure countermeasure.

In this chapter, we propose a new method for verifying the security of randomization based countermeasures against side-channel attacks. Our method uses an SMT solver to check if any intermediate computation result of a software statistically depends on the sensitive data. Since the security of the countermeasure against power analysis attacks is a statistical property, the problem cannot be solved by conventional techniques such symbolic model checking based on Binary Decision Diagrams (BDDs) and satisfiability (SAT) solvers [22, 54, 43, 82, 84, 87]. Although in the literature, there exists some work on tackling the problem using type-based information flow analysis techniques [1, 72], these methods are often overly conservative, leading to the classification of countermeasures as secure when they are not. In contrast, our method always returns the precise result. Although Bayrak *et al.* [10] also used a constraint solver in their CHES13 method, its analysis is significantly less precise than ours. They only check whether a variable is *masked* by some random variable, but do not check whether it is *perfectly masked*, i.e., whether the probability distribution is dependent on the sensitive data. To the best of our knowledge, our method is the first automated verification method that checks for *perfect masking*. This is important because with *order-d* perfect masking, an implementation is provably secure against any type of *order-d* (and lower-order) power analysis attack [47].

We have implemented our new method in an automated verification tool based on the Clang/LLVM compiler [21] and the Yices SMT solver [25]. We encode the verification problem into a series of

quantifier-free first-order logic formulas, whose satisfiability can be decided by Yices. Our SMT encoding scheme is significantly different from the ones used by standard verification methods, because the *perfect masking* property checked by our tool is statistical in nature. For comparison, we also implemented the CHES13 method [10] in our tool. We have conducted experiments on a large set of recently proposed countermeasures, including the ones applied to AES and the MAC-Keccak reference code submitted to Round 3 of NIST's SHA-3 competition. Our results show that the new method is effective in detecting flaws in the masking implementation. Furthermore, the method is scalable enough to handle programs of practical size and complexity.

The remainder of this section is organized as follows. We present our SMT based verification algorithm in Section 4.1. Then, we illustrate the entire verification process using an example in Section 4.2. We present our incremental verification method in Section 4.3, which further improves the scalability of our SMT-based method. We present our experimental results in Section 4.4, We review related work in Section 4.5, and finally provide a summary in Section 4.6.

4.1 SMT-based Method for Verification of Perfect Masking

We first illustrate the overall flow of our verification method using the program in Fig. 4.1. The program is a masked version of $c \leftarrow (k1 \wedge k2)$, where $k1$ and $k2$ are two secret keys, $r1$ and $r2$ are random variables with independent and uniform distribution in $\{0, 1\}$, and c is the computation result. The objective of masking is to make the power consumption of the device executing this code independent from the values of the secret keys. This masking scheme originated from Blömer et al. [16]. The return value c is logically equivalent to $(k1 \wedge k2) \oplus (r1 \wedge r2)$. The corresponding demasking function, which is not shown in the figure, is $c \oplus (r1 \wedge r2)$. Therefore, demasking would produce a result that is logically equivalent to the desired value $(k1 \wedge k2)$.

Our method will determine if all the intermediate variables of the program are perfectly masked. We use the Clang/LLVM compiler to parse the input Boolean program and construct the data-flow graph, where the root represents the output and the leaf nodes represent the input bits. Each

```
1 :   compute(bool k1, bool k2, bool r1, bool r2){
2 :     bool n1, n2, n3, n4, n5, n6, n7, n8, c;
3 :     n1 = k1 ⊕ r1;
4 :     n2 = k2 ⊕ r2;
5 :     n3 = n1 ∧ n2;
6 :     n4 = k2 ⊕ r2;
7 :     n5 = r1 ∧ n4;
8 :     n6 = k1 ⊕ r1;
9 :     n7 = r2 ∧ n6;
10 :    n8 = n5 ⊕ n7;
11 :    c = n3 ⊕ n8;
12 :    return c;
13 :  }
```

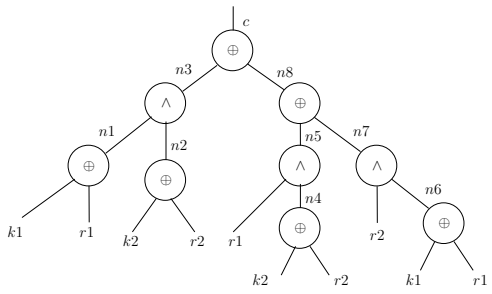


Figure 4.1: Example: the program under verification (left) and its graphic representation (right).

internal node represents the result of a Boolean operation of one of the following types: AND, OR, NOT, and XOR. For the example in Fig. 4.1, our method starts by parsing the program and creating a graph representation. This is followed by traversing the graph in a topological order, from the program inputs (leaf nodes) to the return value (root node). For each internal node, which represents an intermediate result, we check whether it is perfectly masked. The order in which we check the internal nodes is as follows: $n1, n2, n3, n4, n5, n6, n7, n8$, and finally, c.

4.1.1 The Theory

As the starting point of the verification process, we mark all the plaintext bits in x as public, the key bits in k as secret, and the mask bits in r as random. Then, for each intermediate computation result $I(x, k, r)$ of the program, we check whether it is perfectly masked. Following Definition 1, we formulate this check as a satisfiability problem as follows:

$$\exists x. \exists k, k' \ . \ \left(\Sigma_{r\in\{0,1\}^s} I(x, k, r) \neq \Sigma_{r\in\{0,1\}^s} I(x, k', r)\right)$$

Here, x represents the plaintext bits, k and k' represent two different valuations of the key bits, and r is the random number uniformly distributed in the domain $\{0, 1\}^s$, where s is the number of random bits. For any fixed (x, k, k'),

- $\Sigma_{r\in\{0,1\}^s} I(x,k,r)$ is the number of satisfying assignments for $I(x,k,r)$, and
- $\Sigma_{r\in\{0,1\}^s} I(x,k',r)$ is the number of satisfying assignment for $I(x,k',r)$.

Assume that r is uniformly distributed in the domain $\{0,1\}^s$, the above summations can be used to indicate the probabilities of I being logical 1 under two different key values k and k'.

If the above formula is satisfiable, there exists a plaintext x and two different keys (k,k') such that the distribution of $I(x,k,r)$ differs from the distribution of $I(x,k',r)$. In other words, some information of the secret key is leaked through I, and therefore we say that I is not perfectly masked. If the above formula is unsatisfiable, then such information leakage is not possible, and therefore we say that I is perfectly masked.

Another way to understand the above satisfiability problem is to look at the negation. Instead of checking the *satisfiability* of the formula above, we check the *validity* of the formula below:

$$\forall x.\forall k,k'.\ \left(\Sigma_{r\in\{0,1\}^s} I(x,k,r) = \Sigma_{r\in\{0,1\}^s} I(x,k',r)\right)$$

If this formula is valid – meaning that it holds for all valuations of x, k and k' – then we say that I is perfectly masked.

4.1.2 The Encoding

Let Φ denote the SMT formula to be created for checking intermediate result $I(x,k,r)$. Let s be the number of random bits in r. Our encoding method ensures that Φ is satisfiable if and only if I is not perfectly masked. We define Φ as follows:

$$\Phi := \left(\bigwedge_{r=0}^{2^s-1} \Psi_k^r\right) \wedge \left(\bigwedge_{r=0}^{2^s-1} \Psi_{k'}^r\right) \wedge \Psi_{b2i} \wedge \Psi_{sum} \wedge \Psi_{diff}\ ,$$

where the subformulas are defined as follows:

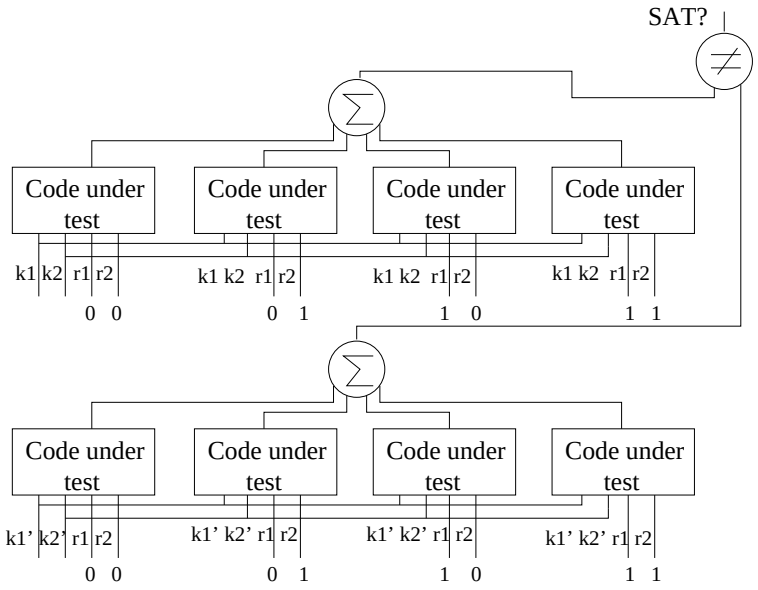


Figure 4.2: SMT encoding for checking the statistical dependence on secret data $(k1, k2)$.

- ***Program logic (Ψ_k^r)***: Each subformula Ψ_k^r encodes a copy of the functionality of $I(x, k, r)$, with the random variable r set to a concrete value in $\{0, \ldots, 2^s - 1\}$ and the key set to value k or k'. All copies share the same plaintext variable x.

- ***Boolean-to-int (Ψ_{b2i})***: It encodes the conversion of the Boolean valued output of $I(x, k, r)$ to an integer (true becomes 1 and false becomes 0), so that the integer values can be summed up later to compute $\Sigma_{r=1}^{2^s} I(x, k, r)$.

- ***Sum-up-the-1s (Ψ_{sum})***: It encodes the two summations of the logical 1s in the outputs of the 2^s program logic copies, one for $I(x, k, r)$ and the other for $I(x, k', r)$.

- ***Different sums (Ψ_{diff})***: It asserts that the two summations should have different results.

Fig. 4.2 is a pictorial illustration of our SMT encoding for an intermediate result $I(k1, k2, r1, r2)$, where $k1, k2$ are the secret key bits and $r1, r2$ are two random bits. Here, the first four boxes, encoding $\Psi_k^0, \ldots, \Psi_k^3$, are the four copies of the program logic for key bits ($k1k2$), with the random bits set to 00, 01, 10, and 11, respectively. The other four boxes, encoding $\Psi_{k'}^0, \ldots, \Psi_{k'}^3$, are the four copies of the program logic for key bits ($k1'k2'$), with the random bits set to 00, 01, 10, and

11, respectively. The formula checks for security against first-order DPA attacks – whether there exists two sets of keys (*k1 k2* and *k1' k2'*) under which the distributions of I are different.

4.1.3 An Example

Consider node $n8$ in Fig. 4.1 as the node under verification. The function is defined as `n8 = (r1 & (k2 xor r2)) xor (r2 & (k1 xor r1))`. The SMT formula that our method generates – by instantiating `r1r2` to 00, 01, 10, and 11 – is the conjunction of all of the formulas listed below:

```
n8_1 = (0 & (k2 xor 0)) xor (0 & (k1 xor 0))         // four copies of I(k, r)
n8_2 = (0 & (k2 xor 1)) xor (1 & (k1 xor 0))
n8_3 = (1 & (k2 xor 0)) xor (0 & (k1 xor 1))
n8_4 = (1 & (k2 xor 1)) xor (1 & (k1 xor 1))
n8_1' = (0 & (k2' xor 0)) xor (0 & (k1' xor 0))      // four copies of I(k',r)
n8_2' = (0 & (k2' xor 1)) xor (1 & (k1' xor 0))
n8_3' = (1 & (k2' xor 0)) xor (0 & (k1' xor 1))
n8_4' = (1 & (k2' xor 1)) xor (1 & (k1' xor 1))
(( num1 = 1 ) & n8_1 ) | ((num1=0) & not n8_1 )      // convert bool to integer
(( num2 = 1 ) & n8_2 ) | ((num2=0) & not n8_2 )
(( num3 = 1 ) & n8_3 ) | ((num3=0) & not n8_3 )
(( num4 = 1 ) & n8_4 ) | ((num4=0) & not n8_4 )
(( num1' = 1 ) & n8_1') | ((num1'=0) & not n8_1')    // convert bool to integer
(( num2' = 1 ) & n8_2') | ((num2'=0) & not n8_2')
(( num3' = 1 ) & n8_3') | ((num3'=0) & not n8_3')
(( num4' = 1 ) & n8_4') | ((num4'=0) & not n8_4')
(num1 + num2 + num3 + num4) != (num1' + num2' + num3' + num4')   // the check
```

We solve the conjunction of the above formulas using an off-the-shelf SMT solver called Yices [25]. In this particular example, the formula is satisfiable. For example, one of the satisfying assignments is `k1k2`=00 and `k1'k2'`=01. We shall show in the next section that, when the key bits are 00, the probability for $n8$ to be logical 1 is 0%; but when the key bits are 01, the probability is 50%. This makes it vulnerable to first-order DPA attacks. Therefore, $n8$ is not perfectly masked.

High-Order Attacks

For a masked code to be resistant to *first-order* differential power analysis (DPA) attacks, all the intermediate results must be perfectly masked. However, even if each intermediate result is perfectly masked, it is still not sufficient to resist *high-order* DPA attacks, where an adversary can simultaneously observe more than one intermediate computation results. For a masking scheme to be resistant to *order-d* DPA attacks, we need to ensure that the joint distribution of any d intermediate results (where $d = 2, 3, \ldots$) is independent of the secret key. That is, for any d intermediate results $I_1, \ldots, I_d$, we check the satisfiability of the following formula:

$$\exists x. \exists k, k' \; . \; \left(\Sigma_{r \in \{0,1\}^s} \Sigma_{i=1}^{d} I_i(x, k, r) \neq \Sigma_{r \in \{0,1\}^s} \Sigma_{i=1}^{d} I_i(x, k', r)\right)$$

Our aforementioned encoding algorithm can be easily extended to implement this new check. In practice, most countermeasures assume that the adversary has access to the side-channel leakage of either one or two intermediate results, which corresponds to first-order and second-order attacks. In our actual implementation, we handle both first-order and second-order attacks. In our experiments, we also evaluate our new method on verifying countermeasures against both first-order and second-order attacks (where $d = 1$ or 2).

4.2 The Running Example

Consider the automated verification of our running example in Fig. 4.1. For each internal node I, we first identify all the transitive fan-in nodes of I in the program to form a *code region* for the subsequent SMT solver based analysis. In the worst case, the extracted code region should start from the instruction (node) to be verified, and cover all the transitive fan-in nodes on which it depends. Then, the extracted code region is given to our SMT based verification procedure, whose goal is to prove (or disprove) that the node is statistically independent of the secret key.

Following a topological order, our method starts with node $n1$, which is defined in Line 3 of the

k1	k2	r1	r2	n3
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

k1	k2	r1	r2	n8
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

k1	k2	r1	r2	c
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Figure 4.3: The truth-tables for internal nodes $n3$, $n8$, and c of the example program in Fig. 4.1.

program in Fig. 4.1. The extracted code region consists of $n1 = k1 \oplus r1$ itself. Since it involves only one key and one random variable in the XOR operation, a simple static analysis can prove that it is perfectly masked. Therefore, although we could have verified it using SMT, we skip it for efficiency reasons. Such simple static analysis is able to prove that $n2, n4$ and $n6$ are also perfectly masked.

Next, we try to prove that node $n3$ is perfectly masked. The truth table of $n3$ is shown in Fig. 4.3 (left). In all four valuations of $k1$ and $k2$, the probability of $n3$ being logical 1 is 25%. Therefore, $n3$ is perfectly masked. When we apply our SMT based method, the solver is not able to find any satisfying assignment for $k1$ and $k2$ under which the probability distributions of $n3$ are different. Note that our method does not check the probability of the output being logical 0, since having an equal probability distribution of logical 1 is equivalent to having an equal probability distribution for logical 0.

The verification steps for nodes $n5$ and $n7$ are similar to that of $n3$ – all of them are perfectly masked.

Next, we try to prove that node $n8$ is perfectly masked. However, the proof would fail because, as shown in the truth table in Fig. 4.3 (middle), the probability for $n8$ to be logical 1 is not the same under different valuations of the keys. For example, if the keys are 00, then $n8$ would be 0 regardless of the values of the random variables. Recall that we have shown the detailed SMT

encoding for $n8$ in Section 4.1.3. Using our method, the solver can quickly find two configurations of the key bits (for example, 00 and 11) under which the probabilities of $n8$ being logical 1 are different. Therefore, $n8$ is not perfectly masked.

The remaining node is c, whose truth table is shown in Fig. 4.3 (right). Similar to $n8$, our SMT based method will be able to show that it is not perfectly masked.

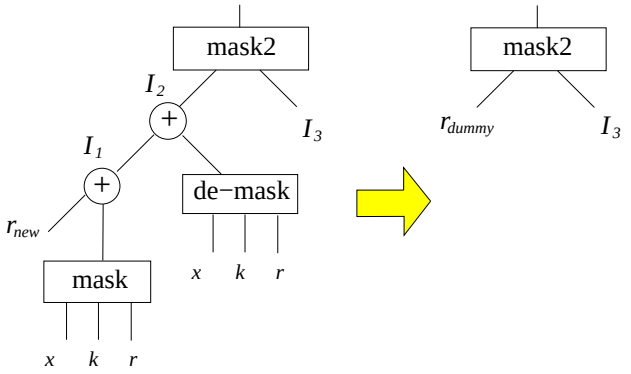
It is worth pointing out that the result of applying the CHES13 method [10] would have been different. Although $n8$ and c are clearly vulnerable to first-order DPA attacks, the CHES13 method, based on the notion of *sensitivity*, would have classified them as "securely masked." This demonstrates a major advantage of our new method over the CHES13 method.

4.3 The Incremental Verification Algorithm

It is worth pointing out that the size of the formula created by our SMT encoding is linear in the size of the program and exponential in the number of random variables – for s random bits, we need to make 2^{s+1} copies of the program logic. This is the main bottleneck for applying our method to large programs. In this section, we propose an incremental verification algorithm, which applies SMT solver based analysis only to small code regions – one at a time – as opposed to the entire fan-in cone of the node under verification. This is crucial for scaling the method up to programs of practical size and complexity.

4.3.1 Extracting the Verification Region

In practice, a common strategy in implementing randomization based countermeasures is to have a chain of modules, where the inputs of each module are masked before executing its logic, and are demasked afterward. To avoid having an unmasked intermediate value, the inputs to the successor module are masked with fresh random variables, before they are demasked from the random variables of the previous module. This can be illustrated by the example in Fig. 4.4, where



$$\begin{aligned} I_2 &:= I_1 \oplus \textit{de-mask}(x,k,r) \\ &:= r_{new} \oplus \textit{mask}(x,k,r) \oplus \textit{de-mask}(x,k,r) \\ &:= r_{new} \oplus (\ldots) \\ &:= r_{dummy} \end{aligned}$$

Before verifying *mask2*, if we have already proved that I_2 is *perfectly masked*, and r_{new} is a new random variable not used elsewhere, then for the purpose of checking *mask2* only, we can substitute I_2 with r_{new} while verifying *mask2*.

Figure 4.4: Applying the SMT based analysis to a small fan-in region only.

the output of *mask(x,k,r)* is masked with the new random variable r_{new} before it is demasked from the old random variable r.

Due to *associativity* of the $\oplus$ operator, reordering the masking and demasking operations would not change the logical result. For example, in Fig. 4.4, the instruction being verified is in *mask2()*. Since the newly added random variable r_{new} is not used inside *mask()* or *de-mask()*, or in the support of I_3, we can replace the entire fan-in cone of I_2 by a new random variable r_{dummy} (or even r_{new} itself) while verifying *mask2()*. We shall see in the experimental results section that such opportunities are abundant in real-world applications. Therefore, in this subsection, we present a sound algorithm for extract a small code region from the fan-in cone of the node under verification.

Our algorithm relies on some auxiliary data structures associated with the current node i under verification: *supportV[i]*, *uniqueM[i]* and *perfectM[i]*.

- *supportV[i]* is the set of inputs in the support of the function of node i.
- *uniqueM[i]* is the set of random inputs that each reaches i along only one path.
- *perfectM[i]* is a subset of *uniqueM[i]* where each random variable, by itself, guarantees that node i is perfectly masked.

These tables can be computed by a traversal of the program nodes as described in Algorithm 3.

For example, for node I_1 in Fig. 4.4, *supportV[*I_1*]*$= \{x, k, r, r_{new}\}$, *uniqueM[*I_1*]*$= \{r, r_{new}\}$, and *perfectM[*I_1*]*$= \{r_{new}\}$, assuming r is not repeated in the mask block. For node I_2, we have *supportV[*I_2*]*$= \{x, k, r, r_{new}\}$, *uniqueM[*I_2*]*$= \{r_{new}\}$, since r reaches I_2 twice and so may have been de-masked, and *perfectM[*I_2*]*$= \{r_{new}\}$.

Algorithm 3 Computing the auxiliary tables for all internal nodes of the program.

```
 1: supportV[i] ← { v } for each input node i with variable v
 2: uniqueM[i] ← { v } for each input node i with random mask variable v
 3: perfectM[i] ← { v } for each input node i with random mask variable v
 4: for each (internal node i in a leaf-to-root topological order) {
 5:    L ← LEFTCHILD(i)
 6:    R ← RIGHTCHILD(i)
 7:    supportV[i] ← supportV[L] ∪ supportV[R]
 8:    uniqueM ← (uniqueM[L] ∪ uniqueM[R]) \ (supportV[L] ∩ supportV[R])
 9:    if (i is an XOR node)
10:       perfectM[i] ← uniqueM[i] ∩ (perfectM[L]∪perfectM[R])
11:    else
12:       perfectM[i] ← { }
13: }
```

Algorithm 4 Extracting a code region for node i for the subsequent SMT based analysis.

```
 1: GETREGION (n, uniqueMATi) {
 2:    if (n is an input node with variable v)
 3:       region.add ← (n, v)
 4:    else if (∃ random variable r ∈ perfectM[n] ∩ uniqueMATi)
 5:       region.add ← (n, r)
 6:    else
 7:       region.add ← (n, {})
 8:       region.add ← GETREGION(n.Left,  uniqueMATi)
 9:       region.add ← GETREGION(n.Right, uniqueMATi)
10:    return region
11: }
```

Our idea of extracting a small code region for SMT based analysis is formalized in Algorithm 4. Given the node i under verification, and *uniqueM[i]* as the set of random variables that each reaches i along only one path, we call GETREGION(*i,uniqueM[i]*) to compute the region. Inside GETREGION, *uniqueM[i]* is renamed to *freshMasksATi*. More specifically, we start by checking each transitive fan-in node n of the current node i. If n is a leaf node (Line 2), then we add n and the input variable v to the region. If n is not a leaf node, we check if there is a random variable $r \in$*uniqueMATi* that, by itself, can perfectly mask node n (Line 4). In Fig. 4.4, for example, r_{new},

by itself, can uniformly mask node I_2. If such random variable r exists, then we add pair (n, r) to the region and return – skipping the entire fan-in cone of n. Otherwise, we recursively invoke GETREGION to traverse the two child nodes of n.

4.3.2 The Overall Algorithm

Algorithm 5 shows the overall flow of our incremental verification method. Given the program and the lists of secret, random and plaintext variables, our method systematically scan through all the internal nodes from the inputs to the return value. For each node i, our method first extracts a small code region (Line 4). Then, we invoke the SMT based analysis. If the node is not perfectly masked, we add it to the list of *bad* nodes.

Algorithm 5 Incremental verification of perfect masking.

```
 1: VERIFYPERFECTMASKING (Prog, keys, rands, plains) {
 2:    badNodes ← { }
 3:    for each (internal node i ∈ Prog in a topological order ) {
 4:          region ← GETREGION(i, uniqueM[i])
 5:          notPerfect ← CHECKMASKINGBYSMT (i, region, keys, rands, plains )
 6:          if (notPerfect)
 7:             badNodes.add( i )
 8:    }
 9:    return badNodes
10: }
```

To optimize the performance of Algorithm 5, we conduct a simple static analysis between Line 4 and Line 5 to quickly check whether it is fruitful to invoke the SMT solver. The first one checks if the region contains any secret keys, if not then the solver is not invoked and the instruction is perfectly masked. The second analysis checks some syntactic conditions – if all of these conditions are satisfied, the current node i is guaranteed to be perfectly masked. In such case, we also avoid invoking the SMT solver. The implemented syntactic conditions are listed as follows:

- The instruction has no secret input as its child. This guarantees that when a secret variable is introduced, its masking operation will be verified.

- None of the random variables appears in both operand's $supportV$ tables. This guarantees that no perfectly masking of a secret variable in any of the operands may be affected.
- Both operands are perfectly masked. This guarantees to find all the resultant imperfect masked instructions due to an initial imperfectly masked instruction.

To further optimize the performance of Algorithm 5, we implement a method for identifying random variables that are *don't cares* for the node i under verification, and use the information to reduce the cost of the SMT based analysis. Prior to the SMT encoding, for each random variable $r \in$*supportV[i]*, we check if the value of r can ever affect the output of i. If the answer is no, then r is a *don't care*. During our SMT encoding, we will set r to logical 0 rather than treat r as a random variable, to to reduce the size of the SMT formula. This can lead to a significant performance improvement since the formula size is exponential in the number of relevant random variables.

We check whether $r \in$ *support[i]* is a *don't care* for node i by constructing a Boolean SAT formula and solving it using the SMT solver. The SAT formula is defined as follows:

$$\Psi_{region}^{r=0} \wedge \Psi_{region}^{r=1} \wedge \Psi_{diffO} \ ,$$

where $\Psi_{region}^{r=0}$ encodes the program logic of the region, with the random bit r set to 0, $\Psi_{region}^{r=1}$ encodes the program logic of the region, with the random bit r set to 1, and Φ_{diffO} asserts that the outputs of these two copies differ. If the above formula is unsatisfiable, then r is a *don't care* for node i.

4.4 Experimental Results

We have implemented our new method in an automated verification tool based on the Clang/LLVM compiler [21] and the Yices SMT solver [25]. Our tool runs in two modes: the monolithic verification mode and the incremental verification mode. The monolithic verification mode applies our SMT based encoding to the entire fan-in cone of each node in the program, whereas the

incremental verification method tries to restrict the SMT encoding to a localized region. In addition to our new method, we also implemented the CHES13 method [10] for the purpose of experimental comparison. The main difference is that our method not only checks whether an node is masked (as in the CHES13 method), but also checks whether it is perfectly masked, i.e. it is statistically independent of the secret key.

We have evaluated our verification tool on a set of recently proposed side-channel countermeasures. Our experimental evaluation was designed to answer the following research questions:

- How effective is our new method? We know that in theory, the new method is more accurate than the CHES13 method. But does it has a significant advantage over the CHES13 method in practice?
- How scalable is our new method, especially in verifying applications of realistic code size and complexity? We have extended our SMT based method with incremental verification. Is it effective in practice?

4.4.1 Benchmarks

Table 4.1 shows the statistics of the benchmarks used in our experimental evaluation. Column 1 shows the name of each benchmark example. Column 2 shows a short description of the implemented algorithm. Column 3 shows the number of lines of code – here, each instruction is a bit level operation. Column 4 shows the number of nodes that represent the intermediate computation results. Columns 5-7 show the number of input bits that are the secret key, the plaintext, and the random variable, respectively.

The benchmarks are classified into three groups. The first group of test cases (P1 to P5) are taken from the CHES13 benchmark [10], all of which contain intermediate variables that are not masked at all. More specifically, P1 is the masking key whitening code on Page 12 of the CHES13 paper. P2 is the AES8 example, a smart card implementation of AES resistant to power analysis, originated from Herbst *et al.* [42]. P3 is the code on Page 13 of the CHES13 paper, also

originated from Herbst *et al.* [42]. P4 is the code on Page 18 of the CHES13 paper, originated from Messerges [58]. P5 is the code on Page 18 of the CHES13 paper, originated from Goubin [36].

The second group of test cases (P6 to P11) are examples where most of the intermediate variables are masked, but none of the masking schemes is perfect. P6 and P7 are the two examples used by Blömer *et al.* [16] (on Page 7). P8 and P9 are the SHA3 MAC-Keccak computation reordered examples, originated from Bertoni *et al.* [14] (Eq. 5.2 on Page 46). P10 and P11 are two experimental masking schemes for the Chi function in SHA3, none of which is perfectly masked.

The third group of test cases (P12 to P17) come from the regeneration of MAC-Keccak reference code submission to NIST in the SHA-3 competition [64]. There are a total of 285k lines of Boolean operation code. The difference among these test cases is that they are protected by various countermeasures, some of which are perfectly masked (e.g. P12) whereas others are not.

Table 4.1: The benchmark description and statistics.

Name	Description	Code Size	Nodes	Keys	Plains	Rands
P1	CHES13 Masked Key Whitening	79	47	16	16	16
P2	CHES13 De-mask and then Mask	67	31	8	0	16
P3	CHES13 AES Shift Rows [2nd-order]	21	21	2	0	2
P4	CHES13 Messerges Boolean to Arithmetic (bit0) [2-order]	23	24	1	0	2
P5	CHES13 Goubin Boolean to Arithmetic (bit0) [2-order]	27	60	1	0	2
P6	Logic Design for AES S-Box (1st implementation)	32	9	2	0	2
P7	Logic Design for AES S-Box (2nd implementation)	40	6	2	0	3
P8	Masked Chi function MAC-Keccak (1st implementation)	59	19	3	0	4
P9	Masked Chi function MAC-Keccak (2nd implementation)	60	19	3	0	4
P10	Syn. Masked Chi func MAC-Keccak (1st implementation)	66	22	3	0	4
P11	Syn. Masked Chi func MAC-Keccak (2nd implementation)	66	22	3	0	4
P12	MAC-Keccak 512b Perfect masked	285k	128k	288	288	805
P13	MAC-Keccak 512b De-mask and then mask – compiler error	285k	128k	288	288	805
P14	MAC-Keccak 512b Not-perfect Masking of Chi function (v1)	285k	128k	288	288	805
P15	MAC-Keccak 512b Not-perfect Masking of Chi function (v2)	285k	152k	288	288	805
P16	MAC-Keccak 512b Not-perfect Masking of Chi function (v3)	285k	128k	288	288	805
P17	MAC-Keccak 512b Unmasking of Pi function	285k	131k	288	288	805

4.4.2 Results

Table 4.2 shows the experimental results run on a machine with a 3.4 GHz Intel i7-2600 CPU, 3.3 GB RAM, and a 32-bit Linux OS. We have compared the performance of three methods: CHES13, New (monolithic), and New (incremental). Here, CHES13 is the method proposed by Bayrak *et*

al. [10], while the other two are our own method. In this table, Column 1 shows the name of each test program. Columns 2-5 show the results of running CHES13, including whether the program passed the check, the number of nodes failed the check, and the total number of nodes checked. Columns 6-9 show the results of running our new monolithic method. Here, `mem-out` means that the method requires more than 4 GB of RAM. Columns 10-14 show the results of running our new incremental method. Here, we also show the number of SMT based masking checks made, which is often much smaller than the number of nodes checked, because many of them are resolved by our static analysis.

Table 4.2: The experimental results: comparing our new method with the CHES13 method [10].

Name	**CHES13**				**New (monolithic)**				**New (incremental)**				
	masked	nodes failed	nodes checked	time	masked perfect	nodes failed	nodes checked	time	masked perfect	nodes failed	nodes checked	SMT mask	time
P1	No	16	47	0.16s	No	16	47	0.22s	No	16	47	16	0.09s
P2	No	8	31	0.21s	No	8	31	0.20s	No	8	31	8	0.09s
P3	No	9	21	1.17s	No	9	21	1.27s	No	9	21	18	0.46s
P4	No	2	24	0.58s	No	2	24	0.65s	No	2	24	8	0.57s
P5	No	2	60	1.19s	No	2	60	1.40s	No	2	60	20	1.12s
P6	Yes	0	9	0.06s	No	2	9	0.10s	No	2	9	2	0.08s
P7	Yes	0	6	0.04s	No	1	6	0.07s	No	1	6	1	0.03s
P8	No	1	19	0.15s	No	3	19	0.26s	No	3	19	3	0.11s
P9	Yes	0	19	0.13s	No	2	19	0.27s	No	2	19	2	0.10s
P10	Yes	0	22	0.18s	No	1	22	0.32s	No	1	22	2	0.14s
P11	Yes	0	22	0.20s	No	1	22	0.37s	No	1	22	3	0.18s
P12	Yes	0	128k	91m53s	-	0	34	mem-out	Yes	0	128K	0	10m48s
P13	No	2560	128k	92m59s	No	1	46	mem-out	No	2560	128K	2560	14m10s
P14	Yes	0	128k	97m38s	-	0	31	mem-out	No	1024	128K	1024	18m20s
P15	Yes	0	152k	132m10s	-	0	32	mem-out	No	512	152K	1024	37m37s
P16	No	512	128k	113m12s	-	0	40	mem-out	No	1536	128K	1536	17m24s
P17	No	4096	131k	103m56s	-	0	34	mem-out	No	4096	131K	4096	17m35s

First, the experimental results show that our new algorithm is more accurate than CHES13 in deciding whether a node is securely masked. Every node that failed the security check of CHES13 would also fail the security check of our new method. However, there are many nodes that passed the check of CHES13, but failed the check of our new method. These are the nodes that are masked, but their probability distributions are still dependent on the sensitive inputs – in other words, they are not perfectly masked.

Second, the experimental results show that our new incremental method is significantly more scalable than the monolithic method. On the first two groups of test cases, where the programs are relatively small, both methods can complete the verification process, and the difference in run

time is small. However, on large programs such as the Keccak reference code, the monolithic method could not finish since it quickly ran out of the 4GB RAM, whereas the incremental method can finish in a reasonable amount of time. Moreover, although the CHES13 method implements a significantly simpler (and hence weaker) check, it is also based on a monolithic verification approach. Our results in Table 4.2 show that, on large examples, our incremental method is significantly faster than the CHES13 method.

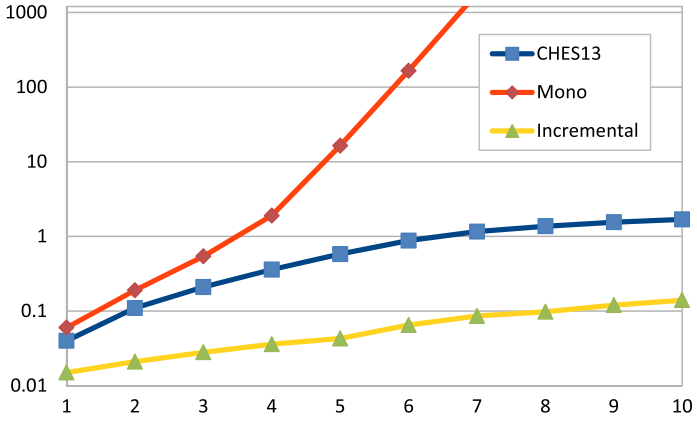


Figure 4.5: Scalability curves.

As a measurement of the scalability of the algorithms, we have conducted experiments on a 1-bit version of test program P1 for 1 to 10 encryption rounds. In each parameterized version, the input for each round is the output from the previous round. We ran the experiment twice, once with an unmasked instruction in each round, and once with all instructions perfectly masked. The results of the two experiments are almost identical, and therefore, we only plot the result for the perfectly masked version. In Figure 4.5, the x-axis shows the program size, and the y-axis shows the verification time in seconds. Among the three methods, our incremental method is the most scalable.

4.5 Related Work

Perfect Masking The notion of *perfect masking* was first introduced by Blömer [16] and subsequently applied to various countermeasures for AES [19]. Later, Goubin [36] proposed a sound method for switching between Boolean masking and arithmetic masking. In addition, there is a large body of work on side-channel analysis resistant designs of AES [65, 42, 58, 60]. However, to the best of our knowledge, there does not exist a method for quantifying the strength of a software countermeasure implementation.

Verification Tools Tools that can formally verify the security of a software implementation are severely lacking. To the best of our knowledge, the only existing tool that can check whether the intermediate computation results of a software implementation are masked is *Sleuth* [10]. However, it only checks whether the intermediate results are masked, i.e. their values depends on some random bits, but does not check the quality of the masking, e.g. whether the intermediate results are statistically independent from the sensitive data. As we have shown in previous sections, there is a big difference between *mathmetically dependent on some random bits* and *statistical independent from sensitive data*, as we have shown in the previous sections. Our new notion of QMS has been proposed specifically to address this problem.

Other Side Channels Beside power side channels, sensitive information may be leaked through many other side channels, such as the execution time [50, 52], faults [15], and cache side channels [37]. Various leak detection and mitigation techniques have also been proposed for these types of side channels. For example, Köpf *et al.* proposed methods for conducting quantitative information flow analysis [53, 6]. Doychev *et al.* [24] developed a static analysis tool for detecting information leaks through cache side channels. Barthe *et al.* [9] proposed a mitigation method designed for defending against concurrent cache attacks. Since these methods focus on other types of side channels, they are orthogonal to the new verification method proposed in this work.

4.6 Summary

We have presented the first fully automated method for formally verifying whether a software implementation is *perfectly masked* by uniformly random inputs, and therefore is secure against power analysis based side-channel attacks. Our new method relies on translating the verification problem into a set of constraint solving problems, which can be decided by off-the-shelf solvers such as Yices. We have also presented an incremental checking procedure to drastically improve the scalability of the SMT based algorithm. We have conducted experiments on a large set of recently proposed countermeasures. Our results show that the new method is not only more precise than existing methods, but also scalable for practical use.

Chapter 5

Quantifying the Masking Strength against Side-Channel Attacks

In recent years, many commercial systems in the embedded space have shown weaknesses against side-channel attacks [66, 59, 7], where an adversary can utilize secondary information such as timing and power consumption resulting from the execution of sensitive algorithms on these devices. For example, the power consumption of an embedded device executing instruction `a=t`$\oplus$`k` may depend on the value of the secret `k` [56] and as a result, `k` can be reliably deduced using a statistical method known as *differential power analysis* (DPA [51]).

Masking, which is a randomization technique for removing the statistical dependency between sensitive data and the side-channel information, is a commonly used mitigation strategy. For example, Boolean masking uses an XOR operation of a random bit r with a variable a to obtain a masked variable: $a_m = a \oplus r$ [7, 68]. Later, the original variable can be restored by a second XOR operation: $a_m \oplus r = a$. Other similar countermeasures have used additive masking ($a_m = a + r \ mod \ n$), multiplicative masking ($a_m = a * r \ mod \ n$), as well as application-specific masking such as RSA blinding ($a_m = ar^e \ mod \ N$).

However, side-channel countermeasures are difficult to design and implement because the process

The content of this chapter is originally published in [32]

is both labor intensive and error prone. There is also no formal method to quantify how secure a software countermeasure really is. This is a problem in practice because the source of the information leakage is not the cryptographic software but the microprocessor hardware that executes the software. For average software developers, who often do not know all the architectural details of the device, it can be difficult to understand when side-channel information may be leaked.

In this chapter, we solve the problem by introducing the notion of *quantitative masking strength (QMS)* to estimate the side-channel resistance of a software implementation. To demonstrate the effectiveness of QMS in quantifying the side-channel resistance, we conduct experiments on a set of cryptographic software on real devices while launching DPA attacks. For each software implementation, we record the number of traces required to successfully break the countermeasure. Our experimental results show that the number of traces, which correlates to the difficulty in breaking the countermeasure, matches the QMS.

We also develop a design automation tool, which leverages a new static code analysis method to compute the QMS of a given C program. The tool can be used as a formal verification procedure as well, to decide whether a program satisfies a given QMS requirement. In case that some intermediate computation results of the program do not satisfy the QMS requirement, our method can produce a side-channel attack scenario, consisting of a combination of the plaintext and the relevant code region that leaks an excessive amount of information about the secret.

Our static code analysis tool builds upon the popular LLVM compiler [21] and the Yices SMT solver [25]. We encode the verification problem into a series of quantifier-free first-order logic formulas, whose satisfiability can be decided by the SMT solver. Although in the literature there exists some work on statically checking the security of mask software code, e.g. using type-based information flow analysis [72], they are significantly less accurate and therefore may generate many false positives. Bayrak *et al.* [10] have used SAT solvers to check if the software is *masked*, but they cannot quantitatively check the masking strength. To the best of our knowledge, our method is the first fully automated static analysis method for checking the strength of masking quantitatively.

We have conducted experiments on a set of cryptographic software implementations to evaluate the performance of our static analysis tool. The benchmarks include several recent countermeasures for AES as well as MAC-Keccak, a MAC based on the new SHA-3 standard. Our experimental results show that the new method is effective in detecting vulnerabilities in the software code and is scalable enough to handle cryptographic software of practical size.

To sum up, this chapter contains the following contributions:

- We propose the new notion of *quantitative masking strength (QMS)* as a way to estimate the side-channel resistance of a masked software implementation in practice.
- We conduct DPA attack experiments on real devices to confirm that the QMS is indeed a good indicator of the side-channel resistance of the masked software.
- We propose a static code analysis method for computing the QMS of a given software program. It can also formally verify that a program satisfies a given QMS requirement.
- When a program fails to satisfy the QMS requirement, our tool will produce an attack scenario, consisting of the plaintext and the code region with excessive information leakage.

The remainder of this chapter is organized as follows. We define the QMS in Section 5.1. We present our static code analysis method in Section 5.2, and describe our DPA attack experiments in Section 5.3. We present our experimental results in Section 5.4, and finally, give a summary in Section 5.5.

5.1 Quantitative Masking Strength (QMS)

Given a pair (x, k) of plaintext and secret key for the function $enc(x, k)$, an s-bit random number r uniformly distributed in the domain $R = \{0, 1\}^s$, and d intermediate results $I_1, \ldots, I_d$, we use $D_{x,k}(R)$ to denote the joint distribution of $I_1, \ldots, I_d$. If $D_{x,k}(R)$ is statistically independent of the secret k, we say that the function is *order-d* perfectly masked [16]. Otherwise, the function is vulnerable to side-channel attacks, and we would like to quantify the bias of $D_{x,k}(R)$, denoted

Δ_{qms}, with respect to x and k.

Definition 2. *Given an implementation of function $enc(x, k)$ and a set of intermediate computation results $\{I_i(x, k, r)\}$, we define the quantitative masking strength (QMS) as the minimal value of $(1 - \Delta_{qms})$ such that, for all d-tuple $\langle I_1, \ldots, I_d\rangle$,*

$$|D_{x,k}(R) - D_{x',k'}(R)| \leq \Delta_{qms} \quad \textit{for any } (x, k) \textit{ and } (x', k') \ .$$

In this sense, the *perfect masking* criterion introduced by Blömer *et al.* [16] is an extreme where $\Delta_{qms} = 0$. The *sensitivity* criterion introduced by Bayrak *et al.* [10] is another extreme where $\Delta_{qms} = 1$. They represent two extreme cases of the spectrum, whereas QMS allows us to quantify the side-channel resistance of the vast number of design choices in between. As an example, consider the four masking schemes in Figure 2.2. In the context of *order-1* side-channel attacks, we have

$$\begin{array}{ll} \Delta_{qms}(o1) = 1/4 - 0/4 = 0.25 & \Delta_{qms}(\overline{o1}) = 4/4 - 3/4 = 0.25 \\ \Delta_{qms}(o2) = 4/4 - 1/4 = 0.75 & \Delta_{qms}(\overline{o2}) = 3/4 - 0/4 = 0.75 \\ \Delta_{qms}(o3) = 3/4 - 1/4 = 0.50 & \Delta_{qms}(\overline{o3}) = 3/4 - 1/4 = 0.50 \\ \Delta_{qms}(o4) = 2/4 - 2/4 = 0.00 & \Delta_{qms}(\overline{o4}) = 2/4 - 2/4 = 0.00 \end{array}$$

All four outputs are *insensitive* according to [10] because of their logical dependence on the random bits, but only o4 is statistically independent of the secret k.

To check if a function satisfies the given QMS requirement, we need to decide whether there exists a d-tuple $\langle I_1, \ldots, I_d\rangle$ such that $|D_{x,k}(R) - D_{x',k'}(R)| > \Delta_{qms}$ for some (x, k) and (x', k'). The function $enc(x, k)$ satisfies the QMS requirement if and only if no such d-tuple exists for the given Δ_{qms} and the given d. Note that $d = 1, 2, \ldots, t$ specifies the order of the side-channel attack. In an order-d attack, we assume that an adversary can measure the leakage of d intermediate computation results simultaneously.

The main challenge for static code analysis – whether to compute the QMS of a given program or to verify that the program satisfies the given QMS requirement – is to compute $D_{x,k}(R)$. As the starting point, we mark all the plaintext bits in x as public, the key bits in k as secret, and the mask bits in r as random. Then, for each $I(x, k, r)$, we check whether it satisfies the QMS requirement. Following Definition 2, we can formulate the *order-1* QMS check as a satisfiability problem as

follows:

$$\exists x, k, k' \ . \ (\Sigma_{r \in R} I(x, k, r) - \Sigma_{r \in R} I(x, k', r)) > \Delta_{qms}$$

Here, x is the plaintext, k and k' are two different values of the secret key, and r is the s-bit random number in domain $R = \{0, 1\}^s$. For any fixed (x, k, k'), the summation $\Sigma_{r \in R} I(x, k, r)$ represents the number of satisfying assignments of $I(x, k, r)$, and the summation $\Sigma_{r \in R} I(x, k', r)$ represents the number of satisfying assignment of $I(x, k', r)$. Assume that r is uniformly distributed in domain $R = \{0, 1\}^s$, the summations represent the probabilities of I being logical 1 under key values k and k', respectively.

If the above formula is satisfiable, there exist x and two keys (k, k') such that the distribution of $I(x, k, r)$ differs from the distribution of $I(x, k', r)$ by more than Δ_{qms}. In other words, the secret values of k and k' are leaked, and the amount of information leakage is more than expected. On the other hand, if the above formula is unsatisfiable, then I satisfies the given QMS requirement.

5.2 Static Code Analysis to Compute the QMS

In this section, we first present our verification procedure, which takes a program and a QMS as input and checks whether the program satisfies the QMS requirement. Then, we present our algorithm for estimating the QMS of a given program, which uses the aforementioned verification procedure as a subroutine.

5.2.1 Checking a Program against a QMS Requirement

Our method is based on translating the verification problem into a set of quantifier-free first-order logic (FOL) formulas and then deciding the formulas using an SMT solver. For each intermediate computation result $I(x, k, r)$, we construct the formula Φ that is satisfiable if and only if there exist a plaintext x and two key values k and k' such that the probability for $I(x, k, r)$ to be logical 1 differs from the probability for $I(x, k', r)$ to be logical 1 by more than Δ_{qms}. Although

```
1 : compute(bool k1, bool k2, bool r1, bool r2){
2 :   bool n1, n2, n3, n4, n5, n6, n7, n8, c;
3 :   n1 = k1 ⊕ r1;
4 :   n2 = k2 ⊕ r2;
5 :   n3 = n1 & n2;
6 :   n4 = k2 ⊕ r2;
7 :   n5 = r1 & n4;
8 :   n6 = k1 ⊕ r1;
9 :   n7 = r2 & n6;
10 :  n8 = n5 ⊕ n7;
11 :  c = n3 ⊕ n8;
12 :  return c;
13 : }
```

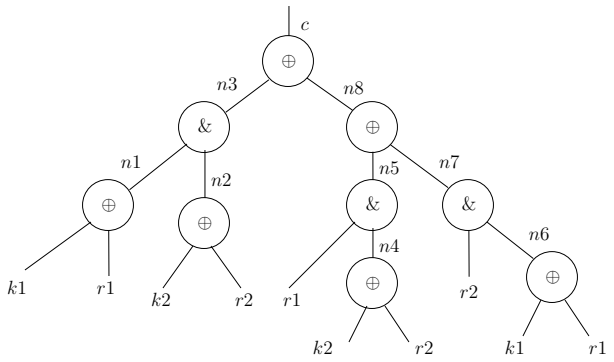


Figure 5.1: A program and the abstract syntax tree (AST) nodes.

satisfiability (SAT) based verification techniques have been widely used in EDA for checking functional correctness properties, our method is significantly different from them because QMS is a quantitative property and is statistical in nature.

Given a Boolean program as input, we first construct a data-flow graph, where the root represents the return value and the leaf nodes represent the inputs. Each internal node represents the result of a Boolean operation of one of the following types: AND, OR, NOT, and XOR. For the example in Figure 5.1, our method starts by parsing the program and creating a graph representation. This is followed by traversing the graph in a topological order, from the program inputs (leaf nodes) to the return value (root node). For each internal node, which represents an intermediate computation result, we check whether it satisfies the given QMS requirement. The order in which we check the internal nodes is as follows: $n1, n2, n3, n4, n5, n6, n7, n8$, and finally, c.

Notice that the program in Figure 5.1 is a masked version of $c \leftarrow (k1 \& k2)$, where $k1$ and $k2$ are secret keys, $r1$ and $r2$ are random variables, and c is the computation result. The return value c is logically equivalent to $(k1 \& k2) \oplus (r1 \& r2)$. This masking scheme (from [16]) is used to make the power consumption independent from the values of $k1$ and $k2$. The corresponding demasking function (not shown in the figure) is $c \oplus (r1 \& r2)$. Therefore, demasking would produce the desired value $(k1 \& k2)$.

Our method will determine if all intermediate variables of the program have a masking strengthen

higher than Δ_{qms}. Let Φ denote the SMT formula to be created for checking the intermediate result $I(x, k, r)$. Let s be the number of random bits in r. Our encoding method ensures that Φ is satisfiable if and only if I violates the QMS requirement. Therefore, we define Φ as follows:

$$\Phi := \left(\bigwedge_{r=0}^{2^s-1} \Psi_k^r\right) \wedge \left(\bigwedge_{r=0}^{2^s-1} \Psi_{k'}^r\right) \wedge \Psi_{b2i} \wedge \Psi_{sum} \wedge \Psi_{diff} \ ,$$

where the subformulas are defined as follows:

- ***Program logic (Ψ_k^r)***: Each subformula Ψ_k^r encodes a copy of the functionality of $I(x, k, r)$, with the random variable r set to a concrete value in $\{0, \ldots, 2^s - 1\}$ and the key set to value k or k'. All copies share the same plaintext value x.
- ***Boolean-to-int (Ψ_{b2i})***: It encodes the conversion of the output of $I(x, k, r)$ from Boolean to integer (true becomes 1 and false becomes 0), so that the integer values can be summed up later to compute $\Sigma_{r \in R} I(x, k, r)$.
- ***Sum-up-the-1s (Ψ_{sum})***: It encodes the two summations of the logical 1s in the outputs of the 2^s copies of program logic, one for $I(x, k, r)$ and the other for $I(x, k', r)$.
- ***Different sums (Ψ_{diff})***: It asserts that the difference between the two summations is bigger than the required Δ_{qms}.

Figure 5.2 is a pictorial illustration of the SMT encoding for output $I(k1, k2, r1, r2)$, where $k1, k2$ are the secret bits and $r1, r2$ are two random bits. The first four boxes, encoding $\Psi_k^0, \ldots, \Psi_k^3$, are copies of the program logic for key bits ($k1k2$) with random bits set to 00, 01, 10, and 11, respectively. The other four boxes, encoding $\Psi_{k'}^0, \ldots, \Psi_{k'}^3$, are copies of the program logic for key bits ($k1'k2'$) with random bits set to 00, 01, 10, and 11, respectively. The formula checks for security against first-order DPA attacks – whether there exist two sets of keys (*k1 k2* and *k1' k2'*) under which the distributions of I differs from each other by more than Δ_{qms}.

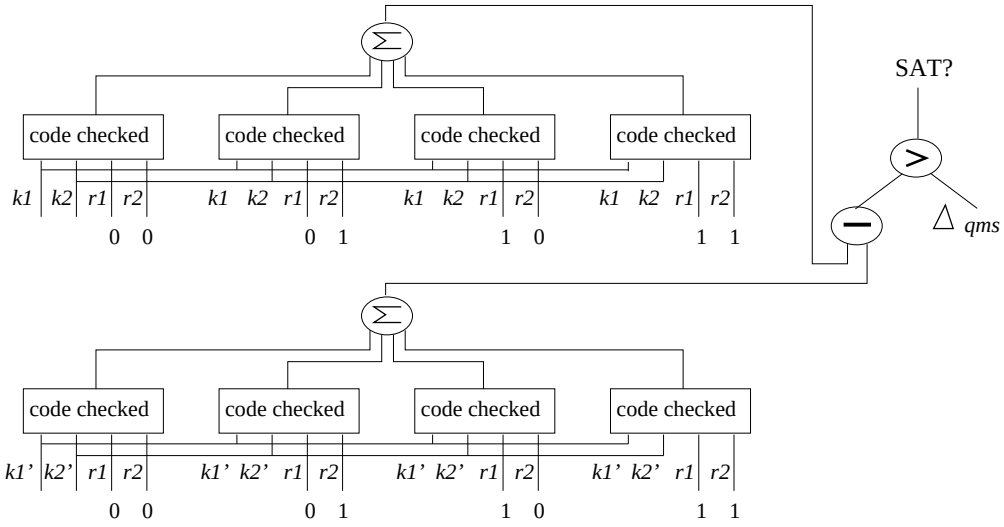


Figure 5.2: SMT encoding to verify the QMS w.r.t. $(k1, k2)$.

5.2.2 Checking the Fan-in AST Nodes Incrementally

Since the SMT formula size is linear in the size of the program but exponential in the number of random variables, it may become a bottleneck if the program uses a large number s of random bits. To avoid the potential performance problem, we propose an incremental algorithm, which applies the SMT based analysis only to small code regions of the program as opposed to the entire fan-in cone of each intermediate computation result. This is crucial for scaling our method to code of practical complexity.

Our incremental algorithm can be illustrated by Figure 5.3, where the output of *mask(x,k,r)* is masked again with the new random variable r_{new} before it is demasked from the old random variable r. Before verifying *mask2*, if we have already proved that I_2 is *perfectly masked*, and r_{new} is a new random variable not used elsewhere (not in computing I_3), then for the purpose of checking *mask2*, we can substitute I_2 with a new random variable r_{dummy} while verifying *mask2*.

Due to *associativity* of the $\oplus$ operator, reordering the masking and demasking operations would not change the logical result. For example, in Figure 5.3, the instruction being analyzed is in *mask2()*. Since random variable r_{new} is not used inside *mask()* or *de-mask()*, or in the support of I_3, we can

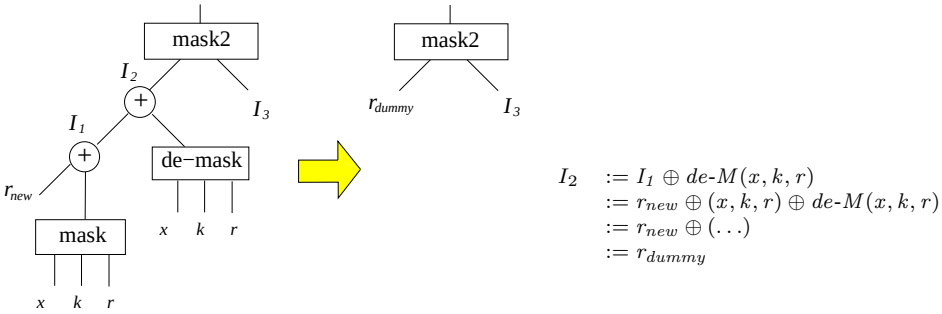


Figure 5.3: Incremental applying the SMT based analysis only to small fan-in region.

replace the entire fan-in cone of I_2 by a new random variable r_{dummy} while verifying *mask2()*.

The effectiveness of our incremental algorithm relies on the following observation. In practice, a common used strategy for implementing randomization based countermeasures is to have a chain of modules, where the inputs of each module are masked before executing its logic, and are demasked afterward. To avoid having an unmasked intermediate value, the inputs to the successor module are masked with fresh random variables, before they are demasked from the random variables of the previous module. We shall see in the experimental results section that such optimization opportunities are abundant in real applications.

5.2.3 Estimating the QMS of a Given Program

Given a program, we can estimate the QMS of all the intermediate computation results by iteratively invoking our SMT based verification procedure as a subroutine. We start with $\Delta_{qms} = 1.0$, and check whether the program satisfies this QMS requirement. If the answer is no, then we decrease Δ_{qms} and check again. We stop as soon as the program satisfies the QMS requirement. At that moment, the value for Δ_{qms} is the estimated QMS of the given program. Algorithm 6 shows the overall flow of our iterative procedure. To make it efficient, we have used the binary search.

It is worth pointing out that in this work, we focus on verifying implementations of cryptographic algorithms, as opposed to arbitrary software applications. The program under verification typically does not have input-dependent control flow, meaning that we can easily remove all the loops

Algorithm 6 Iteratively computing the QMS of a given program.

```
1: COMPUTEQMS (Prog) {
2:    Δ_low ← 0.00
3:    Δ_high ← 1.00
4:    while ( Δ_low ≤ Δ_high ) {
5:         Δ_mid ← (Δ_low + Δ_high)/2.0
6:         if ( CHECKQMS( Prog, Δ_mid) = SAT )
7:            Δ_low ← Δ_mid + 0.01;
8:         else
9:            Δ_high ← Δ_high − 0.01;
10:   }
11:   return Δ_low
12: }
```

and function calls from the code using standard loop unrolling and function inlining techniques. Furthermore, the program can be transformed into a branch-free representation, where the if-else branches are merged. Finally, since all program variables are bounded integers, we can convert the program to a purely Boolean program through bit-blasting. Therefore, in this chapter, our static code analysis method is concerned with only the bit-level representation of a branch-free program.

5.3 Measurements on embedded Devices

To check if QMS reflects the masking strength of a software, we conducted a set of side-channel attacks on implementations of countermeasures for MAC-Keccak, AES, and a few other cryptographic algorithms. We ran all software code on a 32-bit Microblaze processor [85] built on a Xilinx Spartan-3e FPGA (Figure 5.4). To measure the power consumption of the processor core, we used a Tektronix DPO 3034 oscilloscope and a CT-2 current probe to sample the power consumption of the FPGA. The side-channel attack was conducted using differential power analysis (difference of means [51]). To limit the effect of measurement noise, we collected each *trace* after running the same software code 128 times and using the oscilloscope to calculate the average. Here, a trace refers to a set of samples taken during the execution of the software.

We used DPA to determine whether a key guess was correct. Recall that DPA relies on the observation that power consumption variations correlate to the values of the sensitive bits being manipulated. Using the same input vector stream of plaintext as in the measured traces, we

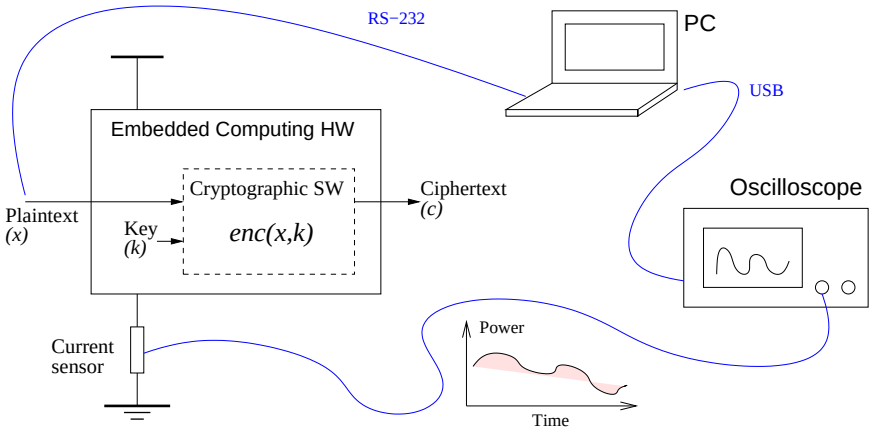


Figure 5.4: The side-channel attack measurement system setup.

compute the value of the sensitive variable assuming that the secret key was one of the key guesses. For an n-bit key, there would be 2^n key guesses. For each key guess, we divide the set of measurement traces into two bins, one for all the sensitive values of logic 0, and one for all the sensitive values of logic 1. Then we compute the difference of means between those two bins, for each key guess. We select the key guess that result in the maximum difference.

Table 5.1: The description and statistics of the masked software benchmarks.

Name	Description	LoC	Nodes	Keys	Plains	Rands
SHA3	A series of masked MAC-Keccak with varying levels of masking (biased random number generators from 0.01 to 0.5 to vary QMS from 0.0 to 1.0)	61	31	3	3	3
AES	A series of masked AES with varying levels of masking (biased random number generators from 0.01 to 0.5 to vary QMS from 0.5 to 1.0)	52	37	8	8	8
P1	CHES13 Masked Key Whitening	79	47	16	16	16
P2	CHES13 De-mask and then Mask	67	31	8	8	16
P3	CHES13 AES Shift Rows	21	21	2	2	2
P4	CHES13 Messerges Boolean to Arithmetic (bit0)	23	24	1	1	2
P5	CHES13 Goubin Boolean to Arithmetic (bit0)	27	60	1	1	2
P6	Logic Design for AES S-Box (1st implementation)	32	9	2	2	2
P7	Masked Chi function MAC-Keccak (1st implementation)	59	19	3	3	4
P8	Masked Chi function MAC-Keccak (2nd implementation)	60	19	3	3	4
P9	Syn. Masked Chi func MAC-Keccak (1st implementation)	66	22	3	3	4
P10	Syn. Masked Chi func MAC-Keccak (2nd implementation)	66	22	3	3	4

We have conducted three sets of experiments. Table 5.1 shows the statistics of the benchmarks, including the name of the program, a short description, the lines of code, the number of computation nodes, as well as the numbers of key bits, plaintext bits, and random bits. The first two sets consist of various versions of the MAC-Keccak and ASE implementations [14, 64, 81, 17]

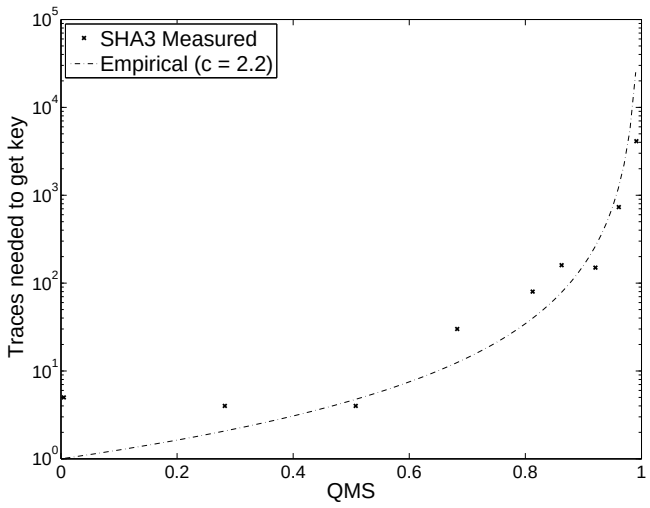


Figure 5.5: DPA attacks on SHA3: QMS vs. number of traces needed to determine the key.

with gradually degrading QMS values. We measured the average number of traces needed to determine the secret key. In the third set of experiments, we used a set of recently published software countermeasures [10, 42, 58, 36, 16], with fixed QMS values, and measured the average number of traces needed to determine the secret key.

Figure 5.5 shows our results on the SHA3 benchmark. The x-axis is the QMS value, while the y-axis is the measured average number of traces needed to determine the secret key. Notice that the y-axis is in logarithmic scale. In addition to the measured data, we have plotted an empirical approximation rule (dotted curve) to estimate the measured data. We can see that when the QMS value approaches 1.0, the number of traces needed to determine the secret key will approach infinity. This is as expected because QMS=1.0 means that the code is perfectly masked – since there is no information leakage, the implementation is provably secure. However, when the QMS value deviates from 1.0 slightly, the number of traces needed to determine the secret key drops drastically – QMS=0.90 corresponds to around 100 DPA traces. Overall, the side-channel resistance, as measured by the number of traces needed to determine the secret key, is exponentially dependent on QMS.

Figure 5.6 shows our results on the AES benchmark. Here, the measured data are similar to those in

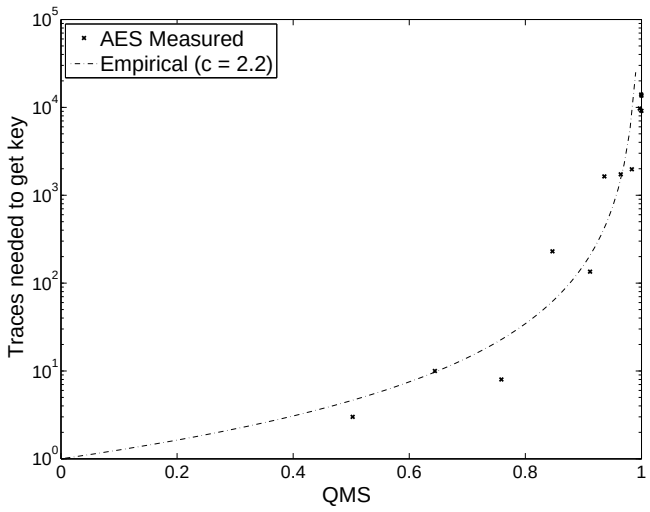


Figure 5.6: DPA attacks on AES: QMS vs. number of traces needed to determine the key.

Figure 5.5. Furthermore, we note that the approximate empirical formula computed to estimate the number of required DPA traces has the following relation with the QMS value: $N_{trace} = \frac{1}{(1-\text{QMS})^c}$, where $c \approx 2.2$ for these two sets of experiments. In general, c is an empirical constant that ultimately will be decided by the actual hardware and measurement set-up. We shall leave the investigation of the theoretical nature of this constant to future work. What is important is that, overall, the side-channel resistance is exponentially dependent on QMS.

Table 5.2 shows our results on the third set of benchmarks. Here, Columns 1 and 2 show the program name and the node to which we have applied the DPA attack. Column 3 shows the QMS value computed statically for the software code. Column 4 shows the number of traces needed to determine the secret key. T.O. means *timed out* after 100,000 traces are measured. It is worth pointing that we performed second order analysis on P3-P5. Overall, we have observed a similar exponential dependence between the number of measured traces and the QMS value. For example, when the QMS is 0.00 – meaning that the node is not masked at all – we have found that the secret key can be determined with merely a handful of DPA traces. When the QMS is 1.00 – meaning it is perfectly masked – the key cannot be determined within our time limit of 100,000 traces. When the QMS is between 0.00 and 1.00, the number of DPA traces closely follows the same empirical

Table 5.2: Relation between QMS and the number of traces needed to determine the key.

Name	Node	QMS	Trace	Name	Node	QMS	Trace
P1	n011	0.00	2	P1	n012	1.00	T.O.
P2	n21	0.00	3	P2	n 11	1.00	T.O.
P3	st10 $\oplus$ st2	0.00	2	P3	rx2 $\oplus$ st2	1.00	T.O.
P4	X $\oplus$ A3	0.00	2	P4	A1 $\oplus$ A3	1.00	T.O.
P5	X $\oplus$ R2	0.00	3	P5	T1 $\oplus$ R2	1.00	T.O.
P6	n09	0.50	936	P6	n07	1.00	T.O.
P7	n32	0.50	992	P7	n35	1.00	T.O.
P8	n02	0.50	587	P8	n23	1.00	T.O.
P9	n47	0.50	255	P9	n39	1.00	T.O.
P10	n47	0.50	426	P10	n48	1.00	T.O.

formula (exponential dependence on the QMS) that we have discovered earlier, but with a slightly different value for constant c.

5.4 Experimental Results

We have also evaluated the efficiency of our new static code analysis methods for QMS estimation and checking in the context of related work. Our experimental evaluation was designed to answer the following questions:

- Is it practical to compute the QMS of a C program through purely static code analysis?
- Does the new method offer significant advantages over existing methods such as *Slueth* [10]?

Our benchmarks included a set of recently published masking countermeasures [10, 17, 42, 58, 36, 16, 14, 64] whose statistics have been shown in Table 5.1. All our experiments were obtained on a desktop computer with a 3.4 GHz Intel i7-2600 CPU, 3.3 GB RAM, and a 32-bit Linux operating system.

Table 5.3 shows the results of applying our new method to compute the QMS of a given software. Column 1 shows the name of the software. Column 2 shows the number of internal nodes checked. Columns 3-6 show the QMS computed, including the minimal, maximal, local average, and global average. Columns 7 and 8 show the number of iterations and the total execution time. The number

of iterations is for the combination of checks on all internal nodes. Also, for P3-P5, we have applied second-order DPA following [10] as opposed to first-order DPA, so each node has been checked against every other node of the program. The results show that our iterative method converged quickly in all cases. Due to page limit, we omit the description of several pieces of useful information reported by our new method, e.g. which node in the program has the lowest QMS and therefore is the most vulnerable to side-channel attacks.

Table 5.3: Statically computing the QMS of the C programs.

Program		**QMS**				**Performance**	
Name	nodes	Min.	Max.	Local Avg.	Global Avg.	Iters	Time
P1	47	0.00	1.00	0.00	0.66	31	0.13s
P2	31	0.00	1.00	0.00	0.74	23	0.41s
P3	21	0.00	1.00	0.33	0.71	108	1.6s
P4	24	0.00	1.00	0.17	0.93	151	1.7s
P5	60	0.00	1.00	0.17	0.97	367	3.1s
P6	9	0.50	1.00	0.50	0.83	11	0.15s
P7	19	0.00	1.00	0.17	0.86	19	0.17s
P8	19	0.50	1.00	0.50	0.92	20	0.16s
P9	22	0.50	1.00	0.50	0.97	23	0.18s
P10	22	0.50	1.00	0.50	0.97	23	0.24s

Table 5.4 shows the results of applying our new method to check whether a program satisfies a given QMS requirement. For comparison, we have re-implemented and evaluated the *Sleuth* algorithm of Bayrak *et al.* [10] in our framework. Here, Columns 1 and 2 show the program name and the number of nodes checked. Columns 3-5 show the statistics of *Sleuth*, including whether it finds any unmasked node, the number of unmasked nodes, and the total execution time. Columns 6-8 show the statistics of our new method, including whether it finds any node that leaks side-channel information, the number of vulnerable nodes found, and the total execution time. In addition to the P1-P10 examples, we have experimented on a set of full-sized MAC-Keccak implementations [14] (P11-P16) in order to compare the scalability of the two methods.

From the results, we have observed several advantages of our new method over *Sleuth*. First, our new method can check for the quantitative masking strength – for any QMS value ranging from 0.00 to 1.00 – whereas *Sleuth* can only check whether a node is masked (whether the QMS is zero or non-zero). The results in Table 5.4 clearly show that there are many cases (e.g. in P6

Table 5.4: Verifying a C program against the QMS requirement.

Program		***Sleuth* [10]**			***New***		
name	nodes	masked	nodes failed	time	masked qms=1.0	nodes failed	time
P1	47	No	16	0.16s	No	16	0.09s
P2	31	No	8	0.21s	No	8	0.14s
P3	21	No	9	1.17s	No	9	1.14s
P4	24	No	2	0.58s	No	2	1.25s
P5	60	No	2	1.19s	No	2	2.53s
P6	9	Yes	0	0.06s	No	2	0.08s
P7	19	No	1	0.15s	No	3	0.12s
P8	19	Yes	0	0.13s	No	2	0.10s
P9	22	Yes	0	0.18s	No	1	0.16s
P10	22	Yes	0	0.20s	No	1	0.18s
P11	128k	Yes	0	91m53s	Yes	0	11m20s
P12	128k	No	2560	92m59s	No	2560	14m45s
P13	128k	Yes	0	97m38s	No	1024	19m26s
P14	152k	Yes	0	132m10s	No	512	37m17s
P15	128k	No	512	113m12s	No	1536	17m44s
P16	131k	No	4096	103m56s	No	4096	18m29s

and P8) where the nodes are masked by some random bits, but the masking is not perfect, and therefore the nodes can still leak sensitive information. Second, our new method is more scalable than *Sleuth*. Although the two methods have comparable run time on small programs, our new method is significantly faster than *Sleuth* on large programs, despite the fact that it is checking a more sophisticated quantitative property. This is due to the fact that we are using incremental SMT analysis as described in Section 5.2.2.

5.5 Summary

We have proposed the notion of quantitative masking strength (QMS), which can, for the first time, represent the side-channel resistance of a masking countermeasure numerically. We have confirmed through experiments that the QMS is a good indicator of the actual masking strength of the software. We have developed a new static analysis tool to compute the QMS of a C program. The method can also be used as a procedure to formally verify a program against a QMS requirement. Our experimental results show that the new static analysis method is effective

in detecting masking flaws and is scalable to handle cryptographic software code of practical size.

Chapter 6

Synthesizing Countermeasures against Power Side-Channel Attacks

When cryptographic algorithms are proved to be secure against thousands of years of brute force cryptanalysis attacks, the assumption is that sensitive information can be manipulated in a closed computing environment. Unfortunately, real computers and microchips leak information about the software code that they execute, e.g. through heat and power dissipation or electromagnetic radiation. For example, the power consumption of a typical embedded device executing instruction `a=t`$\oplus$`k` may depend on the value of the secret variable `k` [56]. Such information can be exploited by an adversary through statistical post-processing such as differential power analysis (DPA [51]), leading to successful attacks in linear time. In recent years, many commercial systems in the embedded space have shown weakness against such attacks [66, 59, 7].

In this chapter, we propose a new synthesis method, which takes an unprotected software program as input and returns a functionally equivalent but side channel leak free new program as output. By leveraging a new verification procedure that we developed recently, called *SC Sniffer* [31, 32], we can guarantee that the synthesized new program is secure by construction. That is, all intermediate computations of the program are *perfectly masked* [16] in that their computation results are statistically independent from the secret data. Masking is a popular and relatively low-

The content of this chapter is originally published in [28]

cost mitigation strategy for removing the statistical dependency between sensitive data and side channel emissions. For example, Boolean masking uses an XOR operation of a random bit r with variable a to obtain a masked variable: $a_m = a \oplus r$ [7, 68]. The original value can be restored by a second XOR operation: $a_m \oplus r = a$. Since a_m no longer depends on the sensitive data a statistically, subsequent computations based on a_m will not leak information about the value of a. Other similar countermeasures have used additive masking ($a_m = a + r \ mod \ n$), multiplicative masking ($a_m = a * r \ mod \ n$), as well as application-specific masking such as RSA blinding ($a_m = ar^e \ mod \ N$).

When a computation $f(z)$ is in the linear domain ($\oplus$ domain), with respect to the sensitive input z, masking can be implemented easily, e.g. as $f(z \oplus r) \oplus f(r)$ since it is equivalent to $f(z) \oplus f(r) \oplus f(r) = f(z)$. That is, we mask z using an XOR with random bit r before the computation and de-mask using an XOR with $f(r)$ afterward. However, when $f(z)$ is a non-linear function, the computation $f(z)$ often needs to be completely redesigned, e.g., by splitting $f()$ into $f'()$ and $f''()$ such that $f'(z \oplus r) \oplus f''(r) = f(z)$. Finding the proper $f'()$ and $f''()$ is a highly creative process currently performed by cryptographic experts. Indeed, designing a new masking countermeasure for algorithms such as AES and SHA-3 would be publishable work in top cryptographic venues.

Our new synthesis method relies on *inductive synthesis* and satisfiability modulo theory (SMT) solvers to search for masking countermeasures within a bounded design space. More specifically, given the software program to be masked, we use a set of quantifier-free first-order logic formulas to encode the two requirements of the synthesized new program – that it must be perfectly masked and that it must be functionally equivalent to the original program. The resulting formulas can be decided by an off-the-shelf SMT solver. Based on this formal analysis with SMT solvers, we can guarantee that the synthesized program is provably secure against power analysis attacks, even on devices with physical emissions.

In the past few years, there is a growing interest in using compilers to automate the application of side-channel countermeasures [2, 11, 12, 63]. However, these existing tools rely on matching known code patterns and applying predefined transformations. They do not employ SMT solver

based exhaustive search or the notion of *perfect masking*. They cannot guarantee to find the leakage free new program even if such program exists, or formally prove that the generated code is leakage free. Our new method provides both guarantees. Although inductive synthesis has enjoyed remarkable success (e.g. [78, 40, 4]), this is the first time that it is applied to mitigating power analysis attacks.

We have implemented our new method in a tool built on the LLVM compiler [21] and the Yices SMT solver [25]. We have conducted experiments on a set of cryptographic software benchmarks, including AES and MAC-Keccak. Our experiments show that the new method is both effective in eliminating side channel leaks and scalable for handling cryptographic software of practical size.

To sum up, we have made the following contributions:

- We propose a new method for synthesizing *masking* countermeasures to protect cryptographic software code against power analysis attacks.
- We implement the method in a software tool, which takes an unprotected C program as input and returns a perfectly masked new program as output.
- We conduct experiments on a set of cryptographic software benchmarks to demonstrate the effectiveness and scalability of the new method.

The remainder of this chapter is organized as follows. We will illustrate the overall flow of our method using an example in Section 6.1. We define the synthesis problem in Section 6.2. The detailed algorithms will be presented in Section 6.3, which includes inductively computing the candidate program, and formally verifying the candidate program. We will present a partitioned synthesis procedure in Section 6.4 to improve the run time performance. Our experimental results will be presented in Section 6.5. Finally, we will give a summary in Section 6.6.

6.1 Motivating Example

In this section, we illustrate the overall flow of our synthesis method using an example. Our example is part of the implementation of MAC-Keccak, the newly standardized SHA-3 cryptographic hashing algorithm [64], after three rounds of competitions by cryptographic experts worldwide. The MAC-Keccak code [14] consists of five main functions that are repeated for 24 rounds on the input bits (plaintext and key) in order to compute the output (ciphertext). The computation in a single round can be represented by $out = \iota.\chi.\pi.\rho.\theta(in)$, where $\iota(), \pi(), \rho()$ and $\theta()$ are linear functions in the domain of $\oplus$, consisting of operations such as XOR, SHIFT and ROTATE, whereas $\chi()$ is a nonlinear function, containing nonlinear operations such as AND.

Our synthesis procedure takes the MAC-Keccak code as input and returns a perfectly masked version of the code as output. It starts by transforming the original program into an intermediate representation (IR) using the LLVM compiler frontend. Since we focus on cryptographic software, not general purpose software, we can assume that all program variables are bounded integers and there is no input-dependent control flow. (Cryptographic software typically do not have input-dependent control flow because it is vulnerable to timing attacks.) Therefore, it is relatively straightforward to transform the input program into a Boolean program, e.g., by merging if-else conditions, unwinding loops, inlining functions, and bit-blasting the integer operations. Thus, from now on, we are only concerned with an IR where all instructions operate on bits. Focusing on the bit-level analysis allows us to detect leaks at the finest granularity possible.

The next step is traversing the abstract syntax tree (AST) nodes of the Boolean program in a topological order, starting at the input nodes and ending at the output node. For each internal node, we first check whether its function is linear or nonlinear in the domain of $\oplus$. As we have shown earlier, for a linear function $f(z)$, we can mask the input z with an XOR of a random bit r before the computation and demask with an XOR of $f(r)$ afterward. Furthermore, to make sure that all intermediate nodes stay masked, we need to chain the mask-demask segments together, by masking the output of a linear function with a new random variable before demasking it with the previous random variable.

```
1 : Chi(bool i1, bool i2, bool i3) {
2 :   bool n1, n2, n3;
3 :   n3 = ¬i2;
4 :   n2 = n3 ∧ i3;
5 :   n1 = n2 ⊕ i1;
6 :   return n1;
7 : }
```

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

```
1 : mChi(bool i1, bool i2, bool i3) {
2 :   bool r1, r2, r3;  //random bits added
3 :   bool b1, b2, b3, n1, n2, n3, n4, n5, n6, n7, n8, n9;
4 :   b1 = i1 ⊕ r1;
5 :   b2 = i2 ⊕ r2;
6 :   b3 = i3 ⊕ r3;
7 :   n9 = b3 ∧ r2;
8 :   n8 = r3 ∧ r2;
9 :   n7 = r3 ∨ b2;
10 :  n6 = r1 ⊕ n9;
11 :  n5 = n7 ⊕ n8;
12 :  n4 = b2 ∨ b3;
13 :  n3 = n5 ⊕ n6;
14 :  n2 = n4 ⊕ b1;
15 :  n1 = n2 ⊕ n3;
16 :  return n1;
17 : }
```

Figure 6.1: The original χ function, its truth table, and the synthesized χ function.

For nonlinear functions, such as the $\chi()$, there are no easy ways of generating the countermeasures. In this work, we rely on the use of iterative inductive synthesis and SMT solvers to search for valid countermeasures in a bounded design space. Given the $\chi()$ function in Fig. 6.1 (left), our method will produce the new code in Fig. 6.1 (right). Our method ensures that these two versions have the same input-output relation, and at the same time, all the intermediate computation results in the new program are perfectly masked with some random bits. Our method has two main advantages over the state of the art. First, it is more economical and sustainable than the manual mitigation approach, especially when considering the rapid increases in the application size and platform variety. Second, it eliminates both the design errors and the implementation errors while guaranteeing that the synthesized program is secure by construction. That is, assume that each of `r1,r2,r3` in Fig. 6.1 (right) is randomly distributed in the domain of $\{0, 1\}$, our method guarantees that the probability of each intermediate computation result being logical 1 (or 0) is statistically independent from the values of `i1,i2,i3`.

6.2 Inductive Synthesis of Masking Countermeasures

We propose using inductive synthesis to generate implementations of perfect masking countermeasures. We follow the iterative synthesis procedure shown in Fig. 6.2, which consists of three steps:

1. Given an unprotected program as input, we first compute a candidate new program that is masked and is functionally equivalent to the original program, at least for a small set of test inputs.
2. We try to prove that the candidate program is perfectly masked and is functionally equivalent to the original program under all possible test inputs.
3. If the verification succeeds, we are done. Otherwise, the candidate program is invalid. In the latter case, we block this solution, go back to Step 1, and try again.

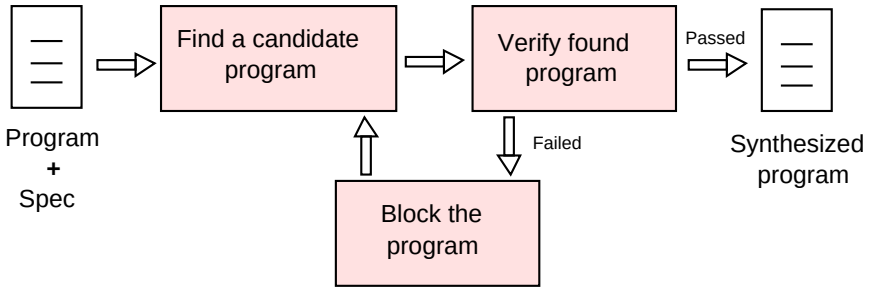


Figure 6.2: The iterative inductive synthesis procedure.

The reason why we choose not to generate, in one step, a candidate program that is valid for all possible test inputs is because of performance concerns. A candidate program valid for all possible test inputs would be prohibitively more expensive for an SMT solver to compute. By separating the synthesis task into three subtasks, namely the inductive synthesis of candidate programs, the formal verification of candidate programs, and the iterative refinement step, we can make all three substeps practically feasible to complete.

In this work, the verification step will consist of two substeps. First, we prove that the candidate program is functionally equivalent to the original program under all possible inputs. Second, we

prove that all intermediate computations in the candidate program are perfectly masked. Toward this end, we leverage a verification procedure that we developed recently, called *SC Sniffer* [31], which can check whether an intermediate computation result of the program is statistically dependent on the secret data.

Our method in *SC Sniffer* [31, 32] relies on translating the verification problem into a series of satisfiability (SAT) problems, each of which is encoded in a set of logical constraints. These constraints can be decided using an off-the-shelf SMT solver. More specifically, we start by making all the plaintext bits in x as public, the key bits in k as secret, and the mask bits in r as random. Then, we traverse the entire program and for each intermediate computation $I(x, k, r)$, check the satisfiability of the following formula:

$$\exists x, k, k' \; . \; \left(\sum_{r \in R} I(x, k, r) \neq \sum_{r \in R} I(x, k', r) \right)$$

Here, k and k' are two different values of the secret key and R is the domain of random variable r. For a fixed value combination (x, k, k'), the summation $\sum_{r \in R} I(x, k, r)$ represents the number of values of r that make $I(x, k, r)$ evaluate to logical 1, and the summation $\sum_{r \in R} I(x, k', r)$ represents the number of values of r that make $I(x, k', r)$ evaluate to logical 1. Assume that random variable r is uniformly distributed in the domain R, the above two summations represent the probabilities of I being logical 1 under key values k and k', respectively. If the above formula is satisfiable, then there exist a plaintext x and two values (k, k') such that the distributions of $I(x, k, r)$ and $I(x, k', r)$ differs – it means that the value of the secret key is leaked. In contrast, if the formula is unsatisfiable, it is a proof that I is perfectly masked. We will present the detailed SMT encoding in Section 6.3.2.

6.3 Synthesis Algorithm

In this section, we present our basic algorithm for iteratively synthesizing a masked version of the input Boolean program. We leave performance optimizations to the next section. The pseudocode is shown in Algorithm 7, where P is the original program, $inputs$ is the set of inputs, and $output$ is the output. The input variables also have annotations indicating whether they are plaintext bits, key bits, or random bits. The synthesis procedure returns a new program $newP$ whose input-output relation is equivalent to that of P. At the same time, all internal nodes of $newP$ are perfectly masked. New random bits may be added by the synthesis procedure gradually on a *need-to* basis.

Algorithm 7 Inductive synthesis of a masked version of the input program P.

```
1: SYNTHESIZEMASKING (P, inputs, output) {
2:   blocked ← { };
3:   testSet ← { };
4:   size ← 1;
5:   while (size < MAX_CODE_SIZE) {
6:     newP ← COMPUTECANDIDATE(P, inputs, output, size, blocked, testSet);
7:     if (newP does not exist)
8:       size ← size + 1;
9:     else {
10:      test1 ← CHECKEQUIVALENCE(newP, P);
11:      test2 ← CHECKINFOLEAKAGE(newP);
12:      if ( {test1, test2} == { } )
13:        return newP;
14:      blocked ← blocked ∪{newP};
15:      testSet ← testSet ∪{test1, test2};
16:    }
17:  }
18:  return no_solution;
19: }
```

The synthesis procedure iterates through three elementary steps: (1) compute a candidate program $newP$ which is functionally equivalent to the original program P, at least for a selected set of test inputs; (2) verify that $newP$ is functionally equivalent to P for all possible inputs and is perfectly masked; (3) if any of the two verification substeps fails, we block this solution, add the failure triggering inputs to *testSet*, and repeat. The synthesis procedure iteratively searches for a new candidate program with increasing code size, until the size reaches MAX_CODE_SIZE. We record the bad solutions in the set $blocked$ to avoid repeating them in the future. We record in $testSet$ all

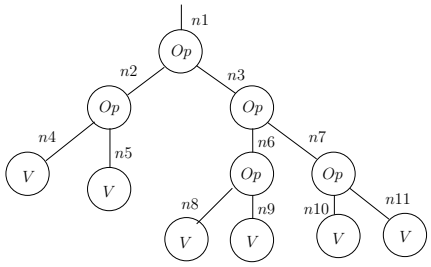


Figure 6.3: A candidate program skeleton consisting of 11 parameterized AST nodes.

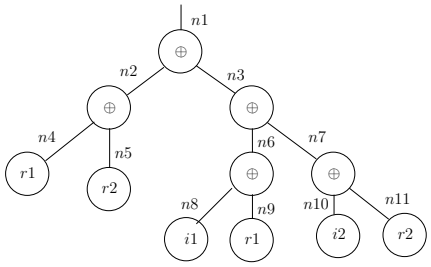


Figure 6.4: The synthesized candidate program with instantiated Boolean masking.

the test cases that led to failures at some previous verification steps.

In the remainder of this section, we present the detailed algorithms for two elementary steps: computing the candidate program and verifying the candidate program.

6.3.1 Computing the Candidate Program

The first step in computing $newP$ from P is to create a parameterized AST that captures all possible masked Boolean programs up to a bounded size. We refer to this AST as a $skeleton$. An example is shown in Fig. 6.3, which has 11 nodes. Each node is either an Op node or a V node. The internal node Op can be instantiated to any bit-level operation such as $\oplus, \&, |$, or !. The V node can be instantiated to any variable in the original program, or fresh random bit added by the synthesis procedure, or constant (logical 0 or 1). The instantiation of Op nodes and V nodes is controlled by a set of auxiliary variables, whose values will be assigned by the SMT solver.

As an example, consider node n_8 in Fig. 6.3. The corresponding logical constraint may be encoded as `((N8==V1)&&bV1)||(N8==V2)&&bV2)`, where `N8` denotes the output of n_8 and `V1` and `V2` are two variables in the input program. Auxiliary variables `bV1` and `bV2` are added to decide which of the node types are chosen – we would add another constraint saying that one and only one of `bV1` and `bV2` must be true. Based on which variable is set to true by the SMT solver, the output of node n_8 is determined. Similarly, for node n_1, the constraint may be `((N1==(N2&N3))&&bAND1) || ((N1==(N2|N3))&&bOR1)||((N1==(N2⊕N3))&&bXOR1)||((N1==(!N2))&&bNOT1` where `N1`,

`N2` and `N3` denote the output of node n_1, n_2, and n_3, respectively. Auxiliary variables `bAND1, bOR1, bXOR1`, and `bNOT1` are constrained such that one and only one of them must be true. Fig. 6.4 shows a masked candidate program synthesized by the SMT solver, which represents $n_1 = i_1 \oplus i_2$.

The next step is to build an SMT formula Φ which imposes two additional requirements: (1) the input-output relation of the candidate program $skeleton$ is equivalent to the original program P, and (2) the internal nodes of the candidate program are all masked by some random variables. More formally, the formula Φ is defined as follows:

$$\Phi = \Phi_P \wedge \Phi_{skel} \wedge \Phi_{iEqv} \wedge \Phi_{oEqv} \wedge \Phi_{masked} \wedge \Phi_{testSet} \wedge \Phi_{blocked},$$

where the subformulas are defined as follows:

- Φ_P encodes the program logic of P.
- Φ_{skel} encodes the program logic of the *skeleton*.
- Φ_{iEqv} asserts that the input variables of P and *skeleton* have the same values.
- Φ_{oEqv} asserts that the outputs of P and *skeleton* have the same value.
- Φ_{masked} asserts that all internal nodes are masked by some random bits – some random bit must appear in the support of the function of each node.
- $\Phi_{testSet}$ asserts that the input variables should take values only from $testSet$.
- $\Phi_{blocked}$ asserts that the previously failed solutions should not be selected.

If formula Φ is satisfiable, a candidate solution is found, and it will be verified for equivalence and perfect masking in the following step. Otherwise, the skeleton $size$ will be incremented and the SMT solver will be invoked again on the new formula.

6.3.2 Verifying the Candidate Program

Given a candidate program $newP$, which is computed by the SMT solver for a set of selected test inputs, we verify that it is a valid solution for all possible inputs. We formulate the verification problem into two satisfiability subproblems, where we look for counterexamples, or test inputs, under which either $newP$ is not equivalent to P, or some nodes in $newP$ are not perfectly masked.

Checking Functional Equivalence

We construct formula Ψ_1 such that it is satisfiable if and only if there exists a test input under which $newP$ and P have different outputs. The formula is defined as follows:

$$\Psi_1 = \Phi_P \wedge \Phi_{newP} \wedge \Phi_{iEqv} \wedge \Phi_{oDiff},$$

where Φ_P and Φ_{newP} encode the input-output relations of the two programs, Φ_{iEqv} asserts that they have the same input values, and Φ_{oDiff} asserts that they have different outputs. If Ψ_1 is satisfiable, we find a test case showing that $newP$ is a bad solution. If Ψ_1 is unsatisfiable, then $newP$ and P are functional equivalent.

Checking for Information Leakage

We construct formula Ψ_2 such that it is satisfiable if and only if there exists an intermediate node in $newP$ that leaks sensitive information. Toward this end, we leverage a verification procedure that we developed recently [31] to check, for each intermediate AST node $I(x, k, r)$, whether there exist a plaintext x and two key values k, k' such that $\sum_{r \in R} I(x, k, r) \neq \sum_{r \in R} I(x, k', r)$. As we have explained in Section 6.1, this inequality means that the probabilistic distributions of $I(x, k, r)$ and $I(x, k', r)$ differ for the two key values k and k'. The formula Ψ_2 is defined as follows:

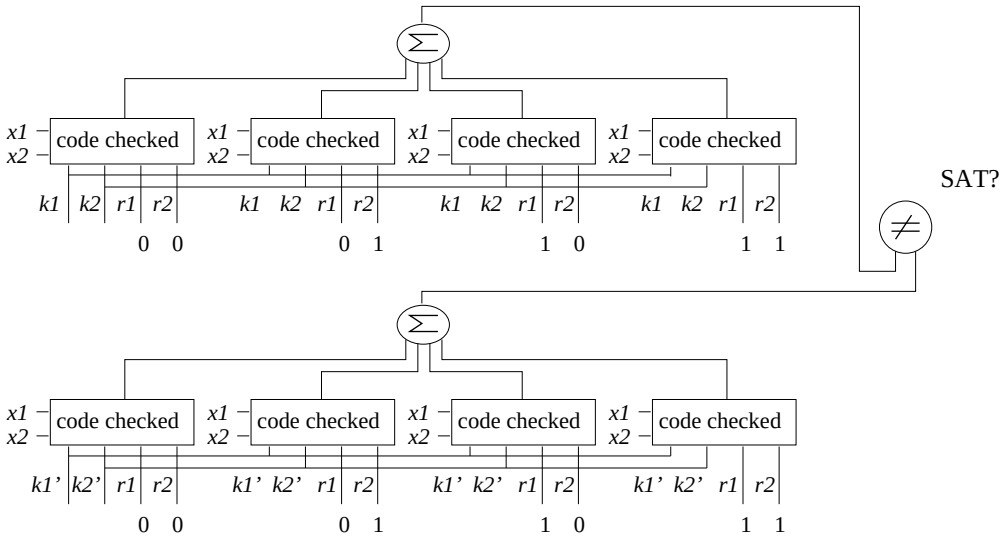


Figure 6.5: SMT encoding for checking the statistical dependence of an output on secret data.

$$\Psi_2 := \left(\bigwedge_{r \in R} \Phi_{I(x,k,r)} \right) \wedge \left(\bigwedge_{r \in R} \Phi_{I(x,k',r)} \right) \wedge \Phi_{b2i} \wedge \Phi_{sum} \wedge \Phi_{sumDiff} \ ,$$

where the subformulas are defined as follows:

- *Program logic ($\Phi_{I(x,k,r)}$)*: Each subformula $\Phi_{I(x,k,r)}$ encodes the input-output relation of $I(x, k, r)$ with a fixed value $r \in R$ and variable k. Each subformula $\Phi_{I(x,k',r)}$ encodes the input-output relation of $I(x, k', r)$ with a fixed value $r \in R$ and variable k'. All subformulas share the same plaintext variable x.
- *Boolean-to-int (Φ_{b2i})*: This subformula encodes the conversion of the bit output of $I(x, k, r)$ to an integer (true becomes 1 and false becomes 0), which will be summed up later to compute $\sum_{r \in R} I(x, k, r)$ and $\sum_{r \in R} I(x, k', r)$.
- *Sum-up-the-1s (Φ_{sum})*: This subformula encodes the two summations of the logical 1's in the outputs of the $|R|$ copies of $I(x, k, r)$ and the $|R|$ copies of $I(x, k', r)$.
- *Different sums ($\Phi_{sumDiff}$)*: It asserts that the two summations have different results.

If Ψ_2 is unsatisfiable, the intermediate result I is perfectly masked. If Ψ_2 is satisfiable, then I has information leakage.

Fig. 6.5 provides a pictorial illustration of our SMT encoding for an intermediate result $I(k1, k2, r1, r2)$, where $k1, k2$ are the key bits and $r1, r2$ are the random bits. The first four boxes encode the program logic of $\Phi_{I(x,k,0)} \ldots \Phi_{I(x,k,3)}$ for key bits ($k1k2$), with the random bits set to 00, 01, 10, and 11, respectively. The other four boxes encode the program logic of $\Phi_{I(x,k',0)} \ldots \Phi_{I(x,k',3)}$ for key bits ($k1'k2'$), with the random bits set to 00, 01, 10, and 11, respectively. The entire formula checks whether there exist two sets of key values (*k1 k2* and *k1' k2'*) under which the probabilities of I being logical 1 are different.

As a more concrete example, consider the computation $c2 = x \oplus k \vee (r1 \wedge r2)$ in Fig. 2.1. The SMT solver may return the solution x=0, k=0 and k'=1 because, according to the truth table in Fig. 2.1, $\sum_{r \in R}\ c2(0, 0, r) = 1$ whereas $\sum_{r \in R}\ c2(0, 1, r) = 4$. Consider $c4 = x \oplus k \oplus (r1 \oplus r2)$ in Fig. 2.1 as another example. The SMT solver will not be able to find any solution because it is perfectly masked. For instance, when x=0, k=0 and k'=1, we have $\sum_{r \in R} c4(0, 0, r) = 2$ and $\sum_{r \in R} c4(0, 1, r) = 2$.

6.4 Partitioned Synthesis Algorithm

SMT solver based inductive synthesis has the advantage of being exhaustive during the search of countermeasures within a bounded design space. With the help of the verification subprocedure, our method also guarantees that the resulting program is secure by construction. However, its main disadvantage is the limited scalability, since the SMT solver slows down quickly as the program size increases. Although we expect SMT solvers to continue improving in the coming years, it is unlikely that a monolithic SMT based synthesis procedure will scale up to large programs (this is consistent with what other researchers in the field have observed [4, 3]). In this section, we propose a *partitioned* synthesis procedure to combine a simple static code analysis with judicious use of an SMT solver, so that the combined method is able to handle cryptographic software of realistic size.

The partitioned synthesis procedure (Fig. 6.6) starts by traversing the AST nodes of the program in

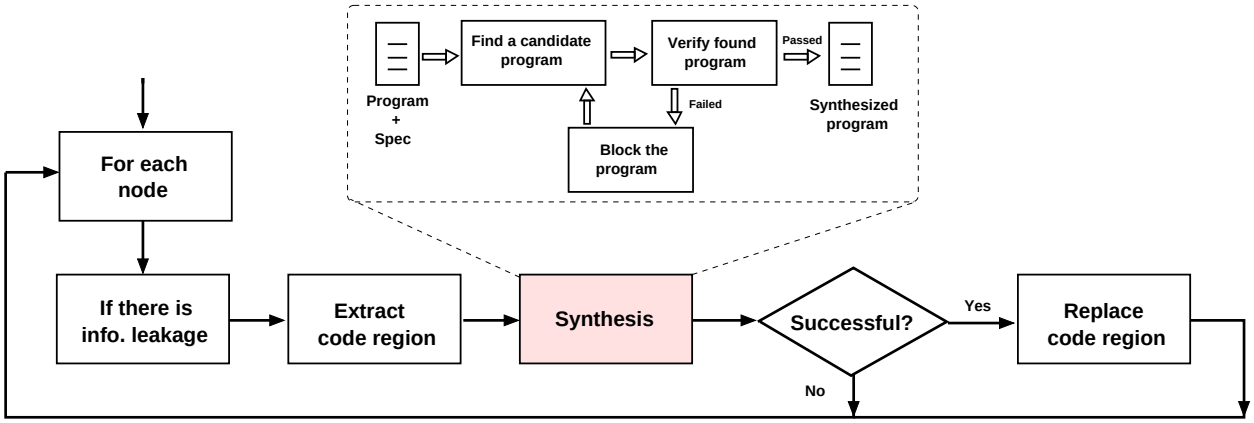


Figure 6.6: The partitioned synthesis procedure for applying masking countermeasures locally.

a topological order from the inputs to the output. Depending on whether the AST node n is linear or nonlinear as shown in Algorithm 8, it invokes either MASKLINEAR or SYNTHESIZEMASKING (presented in the previous section). When n is a linear function, we mask its input variables and demask the output with random variables, without modifying the linear function itself, as explained in Section 6.1. When n is a nonlinear function, instead of invoking SYNTHESIZEMASKING for the entire fan-in cone of n, we partition it into small code regions, and synthesize a masked version for each region. Then, we substitute the original code region reg in program P with the new code region new_reg. The entire synthesis procedure terminates when all small code regions of all nonlinear AST nodes in program P are perfectly masked.

Algorithm 8 Partitioned synthesis algorithm for masking the program P.

```
1: PARTITIONEDSYNTHESIS (P, inputs, output) {
2:   for each (AST node n ∈ P) {
3:     if ( n represents a linear function)
4:       new_n ← MASKLINEAR(P, inputs, n);
5:       replace n in program P with new_n;
6:     else {
7:       while ( ∃ unprotected code region reg ∈ FanIn(n) ) {
8:         Let (reg_ins, reg_out) be the inputs and output of reg;
9:         new_reg ← SYNTHESIZEMASKING(P, reg_ins, reg_out);
10:        replace reg in program P with new_reg;
11:      }
12:    }
13:  }
14:  return P;
15: }
```

6.4.1 Selecting a Code Region

While selecting a code region in $FanIn(n)$, we first start from an AST node $m \in FanIn(n)$ that is not yet perfectly masked, and then include a number of its connected unprotected nodes. The exact number of fan-in nodes to be included in the code region of node m is controlled by a user specified bound. Choosing the right bound, and hence the size of the code region, is a tradeoff between the compactness of the synthesized program and the computational overhead. If we set the bound to positive infinity, the partitioned synthesis procedure would degenerate to the monolithic approach. However, this approach is limited by the capacity of the SMT solvers. On the other hand, if the bound is too small, the synthesized solution may be suboptimal in that some of the masking operations are unnecessary.

For illustration purposes only, we consider an extreme case where the region size is set to 1, meaning that each nonlinear AST node is masked separately. Under this assumption, we illustrate the process of masking the $\chi()$ function in Fig. 6.1. The first code region involves the NOT operation at Line 3, whose masked version is shown as follows:

```
                                              b1  = i2  ⊕ r1;
Line 3:    n3 = ¬ i2;          →              t1  = ¬ b1;
                                              n3  = t1  ⊕ r1;
```

The second code region involves the AND operation at Line 4, whose masked version is shown as follows:

```
                                                    b3  = i3  ⊕ r3;
                                                    b2  = n3  ⊕ r2;
                                                    t10 = ¬ b2;
                                                    t9  = b3  ∧ r2;
Line 4:    n2 = n3 ∧ i3;          →                 t8  = ¬ r3;
                                                    t7  = t10 ∧ r3;
                                                    t6  = b2  ∧ b3;
                                                    t5  = ¬ t9;
                                                    t4  = t8  ∨ r2;
                                                    t3  = t6  ∨ t7;
                                                    t2  = t4  ⊕ t5;
                                                    n2  = t2  ⊕ t3;
```

The third code region involves the XOR of $n2$ and $i1$ at Line 5, whose masked version is shown as follows:

```
                                                    b4  = n2  ⊕ r4;
                                                    b5  = i1  ⊕ r1;
Line 5:    n1 = n2 ⊕ i1;          →                 t12 = b4  ⊕ b5;
                                                    t11 = r1  ⊕ r4;
                                                    n1  = t11 ⊕ t12;
```

It is worth pointing out that, in this extreme case, the resulting program will be suboptimal. However, the actual implementation of our partitioned synthesis procedure was able to obtain a perfectly masked countermeasure whose size is more compact.

6.4.2 Replacing the Code Region

Continue with the above *extreme case* exercise, we now explain how to use the newly synthesized code region (new_reg) to replace the original code region (reg) in program P. The replacement process is mostly straightforward, due to the fact that our partitioned synthesis procedure traverses regions in a bottom-up topological order. However, there is one caveat – before demasking the output of the new region new_reg, we need to mask it with another random variable; otherwise, the output of new_reg would become unmasked.

We solve this problem by asserting, while computing the candidate program in procedure SYNTHESIZEMASKING, that the output and all inputs must be an XOR operation with some random variables. Due to the associativity of XOR operations, and the fact that now two adjacent code regions are connected through two XOR operations, we can switch the order of the two XOR operations during region replacement, without modifying the functionality of the final output.

Below is an example for chaining the three new code regions of the χ function obtained in our *extreme case* exercise, by swapping their adjacent XOR operations.

```
                                        b1  = i2  ⊕ r1;
Line 3:    n3 = ¬ i2;            →      t1  = ¬ b1;
                                        n3  = t1  ⊕ r2;  //swapped with r1
                                        b3  = i3  ⊕ r3;
                                        b2  = n3  ⊕ r1;  //swapped with r2
                                        t10 = ¬ b2;
                                        t9  = b3  ∧ r2;
Line 4:    n2 = n3 ∧ i3;         →      t8  = ¬ r3;
                                        t7  = t10 ∧ r3;
                                        t6  = b2  ∧ b3;
                                        t5  = ¬ t9;
                                        t4  = t8  ∨ r2;
                                        t3  = t6  ∨ t7;
                                        t2  = t4  ⊕ t5;
                                        n2  = t2  ⊕ r4;  //swapped with t3
                                        b4  = n2  ⊕ t3;  //swapped with r4
                                        b5  = i1  ⊕ r1;
Line 5:    n1 = n2 ⊕ i1;         →      t12 = b4  ⊕ b5;
                                        t11 = r1  ⊕ r4;
                                        n1  = t11 ⊕ t12;
```

6.4.3 Reusing Random Variables

To further reduce the size of the synthesized program, we reuse random variables as much as possible while masking the non-adjacent code regions. Specifically, while building the candidate program *skeleton* for a code region reg, we first need to create a list of random variables to be used in the V nodes. The number of random variables is at most as large as the number of input

variables in reg. However, we do not have to create fresh random variables every time they are needed. Instead, we can reuse existing random variables in the program, as long as they are not used in the code regions adjacent to reg. This optimization can significantly reduce the number of random bits required in the masked new program, while at the same time soundly maintaining the statistical independence of the masked nodes.

6.5 Experimental Results

We have implemented our synthesis method in a software tool built upon the LLVM compiler frontend and the Yices SMT solver. Our tool runs in two modes: the monolithic mode and the partitioned mode, to facilitate experimental comparison of the two approaches. We have evaluated our method on a set of cryptographic software benchmarks. Our experimental evaluation was designed to answer the following questions:

- How effective is the new synthesis method in eliminating side channel leaks? Is the synthesized program as compact as the countermeasures handcrafted by experts?
- How scalable is the tool in handling code of realistic size? Our partitioned synthesis procedure is designed to address the scalability problem. Is it effective in practice?

Our benchmarks fall into three categories. The first set, from P1 to P8, are medium sized cryptographic functions that are partially masked. Specifically, P1 and P2 are taken from Bayrak *et al.* [10], which are incorrectly masked computations due to code motion in compiler optimization. P3 and P4 are from Herbst *et al.* [42], which are gate-level implementations of partially masked AES. P5 and P6 are masked versions of the χ function from Bertoni *et al.* [14], after integer to Boolean compilation with optimizations. P7 and P8 are two modified versions of the MAC-Keccak nonlinear χ functions. The second set, from P9 to P12, are small to medium sized cryptographic functions that are completely unmasked. Specifically, P9 is the original MAC-Keccak χ function taken from the reference implementation [14] (Equation 5.2 on Page 46). P10 and P11 are two

Table 6.1: Comparing performance of the monolithic and partitioned synthesis algorithms.

Program						**Monolithic**			**Partitioned**		
Name	LoC	Keys	Plains	Rands	Nodes	Rands	Nodes	Time	Rands	Nodes	Time
P1	79	16	16	16	47	16	85	2.9s	16	85	2.9s
P2	67	8	8	16	31	16	55	1.5s	16	55	1.5s
P3	32	2	2	2	9	4	15	8.3s	4	15	8.1s
P4	32	2	2	2	6	6	9	0.2s	6	9	0.2s
P5	59	3	3	4	18	8	24	19m17s	8	27	8.3s
P6	60	3	3	4	18	8	24	0.5s	8	24	0.5s
P7	66	3	3	4	22	8	25	0.3s	8	25	0.3s
P8	66	3	3	4	22	8	25	0.3s	8	25	0.3s
P9	9	3	0	0	3	-	-	TO	4	14	3.1s
P10	57	8	0	0	37	-	-	TO	8	264	4m36s
P11	82	8	0	0	48	-	-	TO	4	485	13m10s
P12	365	8	0	0	182	-	-	TO	8	1072	22m10s
P13	56k	58	161	58	19k	-	-	TO	58	20k	24m7s
P14	56k	58	161	58	19k	-	-	TO	58	21k	41m37s
P15	56k	58	161	58	19k	-	-	TO	58	21k	36m21s
P16	56k	58	161	58	19k	-	-	TO	58	21k	35m42s
P17	56k	58	161	58	19k	-	-	TO	58	21k	48m15s
P18	56k	58	161	58	19k	-	-	TO	58	20k	23m41s

nonlinear functions, *mul4* and *invg4*, from an implementation of AES in [17]. P12 is a single-round complete implementation of AES found in [17]. The third set, from P13 to P18, are partially masked large programs with a significant number of instructions not yet masked. These programs are generated by us from the MAC-Keccak reference code [64] after converting it from an integer program to a Boolean program. In each case, the whole program has been transformed into a single function to test the scalability of our new methods.

Table 6.1 shows the experimental results obtained on a machine with a 3.4 GHz Intel i7-2600 CPU, 4 GB RAM, and a 32-bit Linux OS. Columns 1-6 show the statistics of each benchmark, including the name, the lines of code, the number of key bits, the number of plaintext bits, the number of random bits, and the number of operations (Nodes). Columns 7-9 show the results of the monolithic synthesis algorithm, including the number of random bits and the number of operations (Nodes) in the synthesized program, as well as the run time. Columns 10-12 show the results of the partitioned synthesis algorithm, including the number of random bits and the number of operations (Nodes) in the synthesized program, as well as the run time. Here, TO means that the tool ran out of the time limit of 4 hours.

The experimental results show that our new synthesis method, especially when it runs in the partitioned mode, is scalable in handling cryptographic software of realistic size. On the first set of test cases, where the programs are small, both monolithic and partitioned procedures can complete quickly, and the differences in run time and compactness of the new program are small. However, on large programs such as AES and MAC Keccak, the monolithic method can not finish within four hours, whereas the partitioned method can finish in a reasonably small amount of time. Furthermore, we can see that most of the existing random bits in the original programs were reused.

As far as the compactness of the new program is concerned, we know of only one benchmark (P9) that has a previously published masking countermeasure. The countermeasure [14], handcrafted by cryptographic engineering experts, has 14 operations. The countermeasure synthesized by our own tool (using the partitioned approach) also has 14 operations. Therefore, at least for this example, it is as compact than the handcrafted countermeasure. However, recall that our method has the additional advantages of being fully automated and at the same time guaranteeing that the synthesized new program is provably secure. Furthermore, when given more CPU time – for example, by setting the time limit to 10 hours and using a slightly larger region size – our synthesis procedure, based on exhaustive search, was able to produce a countermeasure with only 12 operations, which is even more compact than the countermeasure handcrafted by experts.

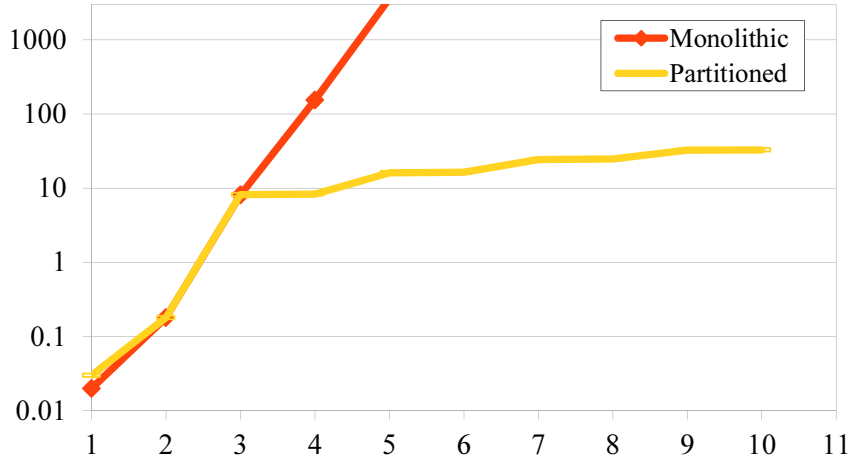


Figure 6.7: Results: Comparing the execution time of the two synthesis procedures.

As another measurement of the scalability of our new methods, we conducted experiments on a parameterized version of test program P1 by expanding it from 1 encryption round up to 10 rounds. In each program, the input for one round is the output from the previous round. We ran the synthesis tool twice, once with the monolithic approach and once with the partitioned approach. The results are plotted in Fig. 6.7, where the x-axis shows the program size and the y-axis shows the run time (in seconds). Note that the y-axis is in logarithmic scale. Whereas the monolithic approach quickly ran out of time for programs with ≥ 5 rounds, the execution time increase of the partitioned approach remains modest.

6.6 Summary

We have presented a new synthesis method for automatically generating perfect masking countermeasures for cryptographic software to defend against power analysis attacks. It guarantees that the resulting software code is secure by construction. We have implemented our method in a tool and evaluated it on a set of cryptographic software benchmarks. Our experiments show that the new method is effective in eliminating side channel leaks and at the same time is scalable for handling programs of practical size. For future work, we plan to continue optimizing our SMT based encoding and at the same time, extending it to handle additive masking, multiplicative masking, as well as application specific masking such as RSA blinding.

Chapter 7

Synthesis of Countermeasures for Fault Attacks

The rising security risks in embedded computing systems have led to the increasingly widespread use of cryptographic modules, implemented either in hardware or in software, to guarantee secure authentication, privacy, and integrity. Although modern cryptographic algorithms are designed to be secure against hundred years of brute-force cryptanalysis, their implementations in hardware or software are often not as secure. For example, there have been reported cases of successful attacks on cryptographic modules in embedded systems, the majority of which were carried out through side-channel attacks [66, 59, 7]. In this context, the adversaries leverage their knowledge of the cryptographic implementations – which are typically available to the public – as well as supplementary information leaked through various side-channels.

Fault Sensitivity Analysis (FSA) is a side-channel attack against cryptographic hardware [55, 73], which exploits the correlation between secret data values and the time needed to propagate them through the cryptographic module and become externally observable. An FSA attack is typically carried out by first injecting a fault to the circuit, e.g., by aggressively reducing the clock period, and then gradually increasing its fault intensity until logical errors occur in the output. One can measure the fault intensity *critical level*, which is defined as the faulty intensity where a faulty

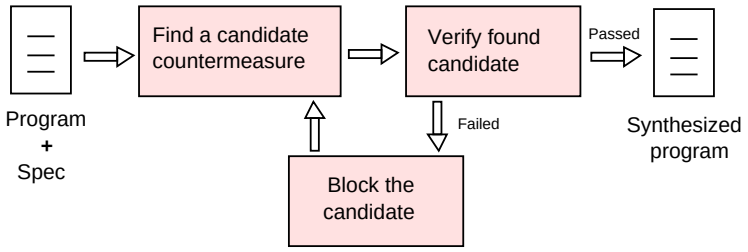


Figure 7.1: Our iterative procedure for FSA countermeasure synthesis.

output first occurs. This critical level is then compared, via statistical analysis, with a simulated critical level computed *a priori*. The comparison result can be used to determine the most likely values of the secret variables.

In this chapter, we propose a new synthesis method for constructing countermeasures of cryptographic hardware to defend against FSA attacks. Given an unprotected circuit as input, together with its sensitive signals clearly marked, our method returns as output a functionally equivalent circuit where the time delay for all output signals are independent of the values of the sensitive signals. In other words, the new circuit is guaranteed to be resistant to FSA attacks.

Figure 7.1 shows the overall flow of our method. Given the circuit C and a set S of sensitive signals, our method first generate a candidate circuit C', which produces the same result as C at least for some input values and is also more likely to have balanced delay along sensitive paths. Then, our method invokes a verification subroutine to formally verify, for all input values, C' is functionally equivalent to C. Furthermore, C' is FSA resistant in that the output delay along all paths are independent of the secret data values. If C' passes the verification step, we are done. Otherwise, we add some logical constraints to block the bad candidate so that it will never be reexamined in the future. In this work, our verification subroutine is implemented as a formal equivalence checking process augmented with lightweight static analysis for computing the delay along sensitive paths.

In practice, the main hurdle of applying inductive synthesis, such as the one illustrated in Fig. 7.1, is scalability. Since the search space can be enormous, inductive synthesis procedures typically work well on small programs or circuits but do not scale up to larger designs. Fortunately, for

this particular application, we can make inductive synthesis efficient by exploiting the idea of compositionality. Since the delay of a path in a circuit is always the summation of the delays of its individual path segments, we can design a partitioned synthesis procedure to scale up the baseline inductive synthesis algorithm. Our new method relies on first partitioning the entire circuit into smaller regions, then synthesizing a solution for each individual region, and finally composing these partial solutions to form a countermeasure for the whole circuit.

We have implemented our new method in a software tool and evaluated it on a set of cryptographic circuits, including nonlinear components of the AES and MAC-Keccac algorithms. Our experimental results show that the new method is both effective in eliminating FSA attack vulnerabilities and efficient enough for practical use.

To summarize, this chapter makes the following contributions:

- We propose a new inductive synthesis method for generating provably secure cryptographic circuits to defend against FSA attacks;
- We propose a partitioned synthesis procedure to scale up our new method in order to handle large circuits;
- We implement the new methods in a software tool and demonstrate their effectiveness on a set of cryptographic circuit benchmarks.

The remainder of this chapter is organized as follows. We start by illustrating our main idea using examples in Section 7.1. We present our baseline countermeasure synthesis algorithm in Section 7.2, followed by the partitioned synthesis procedure in Section 7.3. We present in more detail the synthesis subroutine in Section 7.4. Our experimental results are presented in Section 7.5. We review the related work in Section 7.6. Finally, we give our summary in Section 7.7.

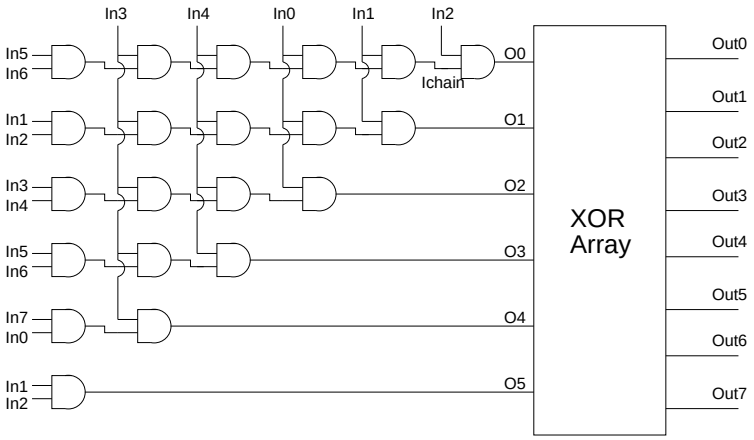


Figure 7.2: Partial PPRM1 AES S-box: vulnerable to FSA attacks.

7.1 Illustrative Example

In this section, we use examples to illustrate the main ideas behind our new method for synthesizing countermeasures to defend against FSA attacks. Our example circuit is an implementation of part of the Advanced Encryption Standard (AES) algorithm. AES has four main functions that are repeated for a number of rounds depending on the required length of the secret key. In this section, we focus on one representative function, the S-box, since it is the only nonlinear function in AES. In cryptographic algorithms, nonlinear functions are often the hardest to protect with countermeasures.

We use the PPRM1 AES S-box implementation proposed in [61], a schematic representation of which is shown in Fig. 7.2. This implementation scheme has been widely used as a benchmark in the cryptographic engineering field [35]. The circuit is constructed from two networks. The first one is a network of XOR gates and the second one is a network of AND gates. For simplicity, we will only explain the synthesis of a countermeasure for the network of AND ages. Therefore, in the remainder of this section, we assume that the AND gate network is the complete circuit. Later in this chapter, we explain how our method can be applied to larger circuits, by first partitioning a circuit into smaller regions, synthesizing countermeasures for these regions individually, and then composing the solutions.

The reason why the circuit in Fig. 7.2 is vulnerable to FSA attacks is because the time taken for the output of the circuit to be computed is dependent on the values of the sensitive inputs. Consider the output signal `o0` of the AND network and the two input signals `In2` and I_{chain}. Let T_{Ichain} be the signal arrival time of I_{chain} and T_{o0} be the time required for the last AND gate to propagate input signals to the output.

If we assume that all input signals `In0-In7` have the same arrival time, and all gates have the same delay, then we have $T_{Ichain} > T_{In2}$. Furthermore, the value of T_{Ichain} depends on the value of the input signals $In0, In1, In3, In4, In5, In6$ as well as the number of gates along the path. If we assume `In2` to be a sensitive variable, the aforementioned mismatch in the arrival time of the signals at the input of the last AND gate will lead to signal `o0` being sensitive. In other words, the secret value of `In2` can be determined using a statistical analysis of the output signal $o0$ based on its dependency of *In2*. In the next paragraph, we briefly explain the intuition behind this attack.

In the context of FSA attacks, we say that the output `o0` is statistically dependent on the sensitive variable `In2` for the following reasons. When the value of `In2` is logical 1, the delay T_{o0} is determined by T_{Ichain}. In contrast, when the value of `In2` is logical 0, the delay T_{o0} is determined by T_{In2}. Since $T_{Ichain} > T_{In2}$, the dependency relation between the required transition time and the secret value of `In2` causes a leak of the sensitive information, which is recoverable by correlation analysis techniques such as FSA [55, 73].

All previously proposed countermeasures, which were typically hand-crafted by cryptographic engineering experts [35, 34], rely on adding delay components to certain input-output paths to eliminate information leaks arising from the nonuniform signal arrival time. For example, Fig. 7.3 shows a recently published countermeasure for the circuit in Fig. 7.2, implemented by manually analyzing the input-output signal paths for each output gate and then adding buffers accordingly to make the delay along all its sensitive signal paths equal.

However, such countermeasure often results in an unnecessarily large number of additional gates in the circuit, thereby leading to higher area cost and energy consumption. Our new method, in contrast, can synthesize a countermeasure with a significantly lower gate count. Fig. 7.4 shows the

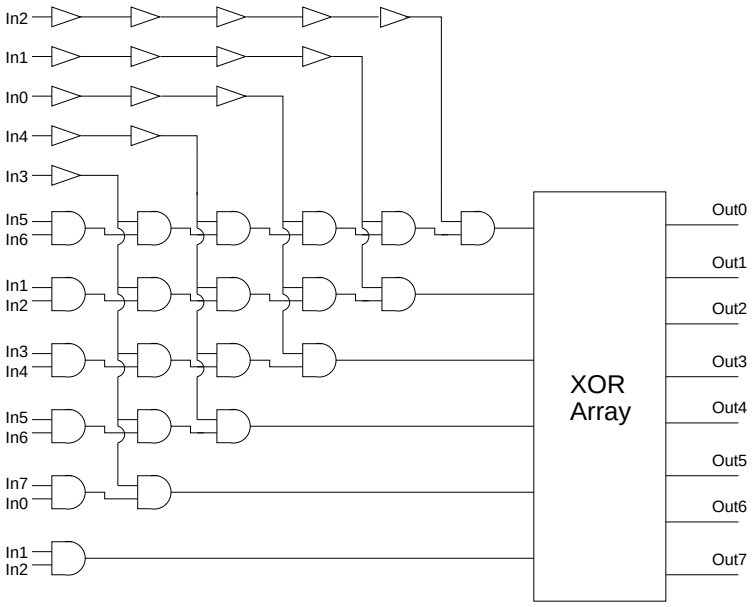


Figure 7.3: AES S-box with inefficient implemented countermeasures.

circuit synthesized by our method, which is functionally equivalent to the circuit Fig. 7.2 and at the same time guarantees to be resistant against FSA attacks. That is, each output gate in the new circuit has the same arrival time for all sensitive input variables.

Comparing our synthesized countermeasure in Fig. 7.4 with the prior solution in Fig. 7.3, we can see that our solution is more efficient, both in terms of the area and the latency of the circuit, and in terms of the power consumption. In fact, our solution uses only 13 gates as opposed to the 41 gates used by the circuit in Fig. 7.3. Furthermore, our new circuit is even smaller in size than the original circuit in Fig. 7.2, which has 21 AND gates.

At this moment, it is worth pointing out that traditional logic synthesis techniques, such as two-level and multi-level minimization algorithms implemented in state-of-the-art EDA tools, do not have the capability of synthesizing secure-by-construction FSA countermeasures similar to ours. Although these existing tools can optimize the area and power consumption of a circuit as well as reducing the delay of critical paths, they do not guarantee that all sensitive signal paths exhibit the same amount of delay. Furthermore, due to the lack of a solid theoretical foundation, it is difficult to customize such tools to solve the FSA countermeasure synthesis problem.

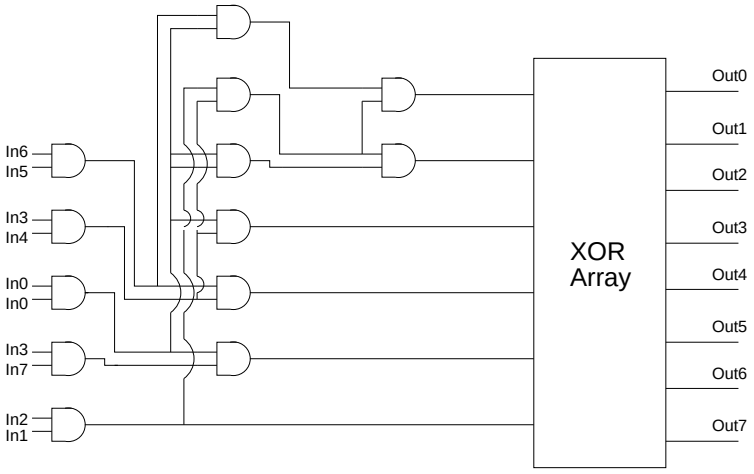


Figure 7.4: AES S-box with efficient implemented countermeasures.

In this work, we leverage the idea of *inductive synthesis* to generate secure-by-construction FSA countermeasures. Although inductive synthesis has been successfully applied to many domains, to the best of our knowledge, it has never been used to synthesize FSA countermeasures. However, it is not easy to synthesize FSA attack countermeasures because we have to search for an alternative, equivalent, and FSA resistant implementation of the given circuit within an extremely large design space. Furthermore, cryptographic circuits are known to be difficult to analyze by symbolic techniques. Therefore, scaling up the synthesis algorithm for cryptographic circuits of practical size is a challenging task. After presenting our baseline algorithm in Section 7.2, we will leverage the *divide-and-conquer* principle to scale our countermeasure synthesis method to circuits of practical size and complexity.

7.2 Synthesis of FSA Countermeasures

In this section, we present our new method for synthesizing FSA countermeasures, which takes an unprotected circuit as input and returns an FSA-resistant circuit as output.

Recall that in the context of inductive synthesis, there needs to be a synthesis subroutine and a verification subroutine. The synthesis subroutine guesses a candidate solution and the verifica-

tion subroutine formally verifies that it is a valid solution. In the context of synthesizing FSA countermeasures, our verification subroutine needs to check the following two properties:

- (1) the new circuit is functionally equivalent to the original circuit; and
- (2) the new circuit is FSA-resistant, meaning that it does not have unequal delay paths from sensitive data to the circuit output.

To reduce the computational overhead, we formulate the synthesis subproblem in such a way that, every candidate solution is already guaranteed to be FSA-resistant (Property 2). In such case, the verification subroutine has to check only the functional equivalence between the candidate solution and the original circuit (Property 1).

The main idea is to construct a so-called *template circuit*, whose instantiations are guaranteed to be FSA-resistant and at the same time cover all possible countermeasure solutions. Without loss of generality, we assume that all logical gates of the same type have equal propagation delay to ease our presentation. Under this assumption, we can ensure FSA-resistance by requiring all paths from sensitive (input) signals to the output signals to have an equal number of logic gates.

Consider the motivating example in Fig. 7.2 again. Its FSA-resistant template circuit can be illustrated by the diagram in Fig. 7.5, where gates and signals are distributed to five levels. Here, Level 0 generally consists of the output signals, whereas Level 4 consists of the input signals. In between these two levels, there can be logic gates of various types such as AND, OR, and NOT. This is a template circuit because the internal logic gates have not yet been chosen and signals have not yet been connected to each other.

To make sure that all instantiations of this template circuit are FSA-resistant, we restrict the connection of signals as follows:

- All sensitive input nodes are placed on the same level — note that they do not have to be at the bottom level;

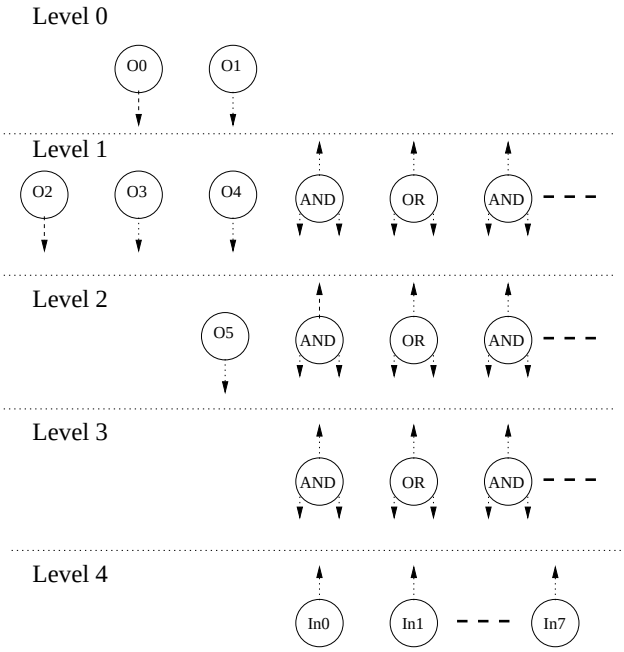


Figure 7.5: The FSA-resistant template circuit structure.

- Each node is constrained to connect to a node either one level higher or one level lower to ensure the levels assigned to each node remain valid.

These constrains guarantee an equal number of gates between any gate's output and all of its connected sensitive inputs. In turn, this ensures the arrival time of all sensitive inputs are equal. However, it is worth pointing out that we do not require the signal arrival time of any two unconnected gates to be equal.

To reduce the computational overhead of the synthesis procedure, we statically estimate the level at which the output signals should be placed, based on the number of inputs it is connected to and the level required for each input. The initial level assigned to an output node is the minimal depth needed to separate an output node from the sensitive input nodes to fit all nodes in a tree structure. If the synthesis procedure fails to find a solution using the estimated level, another call to the synthesis procedure will be performed, after shifting the output nodes one level up in order to search for a larger candidate circuit.

We have formulated our countermeasure synthesis algorithm in the SyGuS specification language,

and then leverage solvers implemented in the SyGuS tool [5] to generate the countermeasure. The SyGuS tool currently has three backend engines, from which one can be chosen to solve the problem. One backend engine is based on the use of SMT solvers, another is based on heuristic guided enumeration, and the third is based on a stochastic search of the design space. Our experience shows that, for this work, the enumerative solver in SyGuS often has the best performance.

However, since SyGuS was not designed to synthesis programs with multiple output signals, the existing SyGuS tool can be suboptimal when used to synthesize FSA countermeasures. To scale up our new method to realistic cryptographic circuits, we encode the synthesis subproblem to allow different outputs to share the same set of intermediate gates, as shown in the synthesized circuit in Fig. 7.4. This is crucial for our method to generate circuits that are small in size; indeed, the synthesized circuit with FSA countermeasure is sometimes smaller than the original circuit.

Therefore, we not only propose the first method that leverages SyGuS to solve the FSA countermeasure synthesis problem, but also extended the existing implementation of the enumerative solver in SyGuS, to speed up the computation and reduce the solution size. In other words, we allow SyGuS to synthesize functions with multiple inputs and multiple outputs, where the internal nodes are shared among the output functions as much as possible, as shown in Fig. 7.4.

Unfortunately, applying the existing solvers in SyGuS directly to FSA countermeasure synthesis is not practical for cryptographic circuits of large size. This is due to the inherent complexity of the inductive synthesis method, whose runtime increases rapidly as the size and complexity of the circuit increase. As a result, only small circuits can be handled by SyGuS directly. To overcome this scalability problem, we propose a new partitioned synthesis approach, which applies SyGuS only to small circuit regions, one at a time, as opposed to the entire circuit at once. Due to the compositionality of the FSA countermeasures, our partitioned approach has the capability of handling cryptographic circuits of any size.

7.3 The Partitioned Synthesis Approach

In this section, we present our partitioned approach to synthesizing FSA countermeasures. Our method starts by parsing the original circuit and creating an intermediate representation (IR) in the form of a directed acyclic graph (DAG). The reason why the original circuit is a DAG is because it represents on the combinational part of a sequential circuit. More specifically, the input signals are either primary inputs or pseudo primary inputs (output of latches from the previous clock cycle).

The DAG is first transversed in a topological order and then partitioned into a set of smaller circuit regions. Each circuit region is statically analyzed to see if it is vulnerable to FSA attacks. For example, if there are discrepancies between the delay along different paths from sensitive inputs to the outputs, we would assume it is vulnerable. For each vulnerable circuit region, we invoke the SyGuS-based synthesis subroutine to generate a new circuit. By replacing the vulnerable circuit region with the synthesized circuit region, we can eliminate the vulnerability. This process (extracting and replacing vulnerable regions) is repeated until no vulnerable circuit region can be found.

Algorithm 9 shows the overall flow of the new partitioned synthesis procedure, where P is the original circuit, $InputSort$ is a map from each primary input to a type (sensitive or non-sensitive), $GatesPD$ is a map from each logic gate in the original circuit to its propagation delay, $GatesSyn$ is a set of logic gates to be used during synthesis of the new circuit, and lev is the maximum depth of the circuit to be synthesized, which in turn determines the maximum size of the synthesized circuit.

The partitioned synthesis method first finds a sensitive gate in the circuit, denoted $sGate$, based on which it partitions the circuit P into smaller circuit regions. More specifically, it starts by statically analyzing each logical gate g in the circuit P and creating three tables with values associated for each gate. The first, $MaxPD$ is the maximum path delay from the gate to the output of the circuit. The second, $MinAr$ is the minimum arrival time of any of the sensitive inputs to the gate. The third, $MaxAr$ is the maximum arrival time of any of the sensitive inputs to the gate.

Algorithm 9 Partitioned Synthesis of the FSA-resistant circuit.

```
1: ANALYZE (P, InputSort, GatesPD, GatesSyn, lev) {
2:   while true{
3:     for each gate g ∈ P{
4:       MaxPD[g] ← GETMAXPD(g, GatesPD, P);
5:       MinAr[g] ← GETMINAR(g, GatesPD, P);
6:       MaxAr[g] ← GETMAXAR(g, GatesPD, P);
7:     }
8:     sGate ← GETSENSITIVE(MaxPD, MinAr, MaxAr, P);
9:     if (sGate == { })
10:      return P;
11:    n = 2^lev − 1;
12:    newReg ← {}
13:    while (newReg == {}){
14:      reg ← GETREG(sGate, MinAr, MaxAr, P, n);
15:      newReg ← SYNTHESIZE(reg, MinAr, GatesSyn, lev);
16:      n − −;
17:    }
18:    P ← UPDATEREGION(P, reg, newReg)
19:  }
20: }
```

The subroutine GETSENSITIVE returns the next sensitive gate $sGate$ that is vulnerable to FSA attacks. It is a gate where the maximum arrival time $MaxAr[g]$ differs from the minimum arrival time $MinAr[g]$. In the presence of multiple sensitive gates, this subroutine returns the sensitive gate with the minimum propagation delay from the sensitive inputs. If there is a tie, the gate with the maximum propagation delay to the output of the circuit is selected as $sGate$. This ranking heuristic is crucial to ensure that our method finds a small countermeasure circuit.

Next, we iteratively extract a circuit region reg of size n, consisting of gates close to the sensitive gate $sGate$, and synthesize a new region $newReg$. The subroutine GETREG returns a region reg consisting of both $sGate$ and gates close to it. Then, the subroutine SYNTHESIZE is invoked to compute the new region $newReg$: it needs to be functionally equivalent to reg and at the same time more FSA-resistant than reg. We say that $newReg$ is more resistant, rather than completely resistant, if the mismatch between the maximum and minimum arrival times of the inputs of reg exceeds the maximum depth of $newReg$ defined by the user in lev. In such case, $newReg$ reduces the mismatch in arrival time between the inputs, and then the arrival time mismatch will be eliminated in later synthesis iterations. We illustrate the subroutine in more details in Section 7.4.

If the synthesis subroutine fails to find $newReg$ within the given size n, it will be invoked again to search for a solution for another circuit region reg with a smaller number of gates $(n-1)$. Since lev has to be assigned to a value greater than one, the procedure will avoid running in an infinite loop at Line 13. It will always find a new region before n reaches zero.

After a successful synthesis of a countermeasure for a new region circuit, the region in the original program will be replaced with the synthesized region. The partitioned synthesis method will continue until no more sensitive gates remain in the circuit. At this point, the synthesis is considered complete and the new circuit P is returned to the user.

7.3.1 Region Selection

The subroutine GETREG in Algorithm 9 is responsible for extracting a vulnerable circuit region with at most n gates. Inside this subroutine, the sensitive gate $sGate$ is first added in the region reg. Then, reg is expanded by adding the chain of sensitive fan-out gates. If no further sensitive fan-out gates are available, the chain of sensitive fan-in gates of $sGate$ are added. If there are more than one sensitive fan-in gate, the gate with the minimum arrival time is added to reg first.

It is worth pointing out that the above ordering heuristic for the gate selection can guarantee termination of the partition method. The reason is that it ensures our synthesis of countermeasures for the circuit regions follows a topological order, starting from the inputs. This avoids the need to re-synthesize countermeasures for the same gate. At the same time, it reduces the maximum mismatch in the arrival time by decreasing the circuit maximum depth rather than inserting gates for a delay effect, which in turn avoids a blow-up in the synthesized circuit size.

The selected region reg would have a maximum size of $n = 2^{lev} - 1$ gates, which occurs if all inputs variables have equal arrival time from the inputs. If the region inputs have different arrival times, this is accounted and compensated for by assigning the inputs at different depths in new region. In this case, the number of gates would decrease because some of the internal nodes are converted to input nodes and their children are removed. Fig. 7.6 illustrates why for a $newReg$

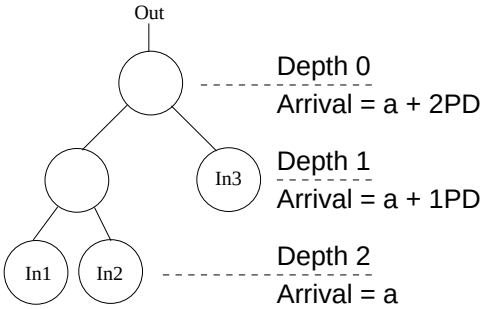


Figure 7.6: Example for a selected region reg.

with lev of 2 and three inputs with arrival time of a, a and $a+1$ cannot have any more inputs added to it, although normally a tree of depth 2 can have up to 4 inputs.

7.3.2 Relevant Parameters

In Algorithm 9, the parameters $GatesSyn$ and lev are decided by the user to find a sweet spot among the several optimization factors. For example, by including more types of gates in $GatesSyn$ (to be used during the synthesis process), the number of possible solutions will increase, which may lead to a more compact countermeasure. On the other hand, it will also increase the search space and make our method less scalable.

Increasing lev will increase the size of the extracted region, which in turn improves the quality of the synthesized circuit regions. This is because optimizations such as gate sharing is more likely in larger circuits than in smaller circuits. On the other hand, increasing lev will lead to harder synthesis subproblems, which would take the SyGuS solver more time to return a solution.

7.4 The Synthesis Subroutine

The subroutine SYNTHESIZE attempts to generate a new circuit $newReg$ that is logically equivalent to reg and at the same time FSA-resistant. The pseudocode is shown in Algorithm 10, where the input consists of the region reg, the map $MinAr$ from inputs to each gate minimum arrival

time, the set $GatesSyn$ of logic gates to be used in the new circuit, and lev the the maximum allowed depth of the new circuit. The procedure returns a candidate circuit if such a solution exists within the given solution space.

Algorithm 10 Inductively synthesizing the new circuit region.

```
 1: SYNTHESIZE (reg, MinAr, GatesSyn, lev) {
 2:   testEx ← { };
 3:   Depth ← GETINPUTDEPTH (reg, lev, MinAr);
 4:   while (true) {
 5:     newReg ← GENNEWREGION(reg,testEx, Depth, GatesSyn, lev);
 6:     if (newReg exists) {
 7:       test ← CHECKEQUIVALENCE(reg, newReg);
 8:       if (test == { })
 9:         return newReg;
10:       testEx ← testEx ∪{test};
11:     }
12:     else
13:       return { };
14:   }
15: }
```

The synthesis subroutine starts by initializing the set $testEx$ of test examples to an empty set. This set consists of test examples used to check the partial equivalence between the candidate circuit $newReg$ and the original circuit reg. That is, at least for all the test examples in $testEx$, the two circuits should behave the same. Subroutine GETINPUTDEPTH computes the appropriate depth for each of the input signals in order to reduce the discrepancies among their arrival time at the outputs.

Then, the synthesis subroutine enters a while-loop containing two main steps. The first step, consisting of a call to GENNEWREGION, searches for a candidate solution $newReg$ that behaves the same as the original circuit, at least for all test examples in $testEx$. The second step, consisting of a call to CHECKEQUIVALENCE, tries to prove the functional equivalence of reg and $newReg$ for all input values. If the two circuits are not equivalent, it computes a test example that can differentiate them. This test example is added to the set $testEx$ before the while-loop enters the next iteration.

The while-loop terminates either when a candidate solution is proved to be the real solution, or no

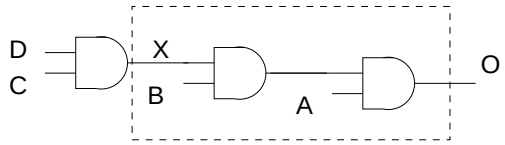


Figure 7.7: Example for a vulnerable circuit.

new candidate solution can be computed.

7.4.1 Computing the Input Depth

The subroutine GETINPUTDEPTH computes, for each input signal in reg, its allowed depth in $newReg$ (or the so-called *level*). Since each of the input signals in reg may have a different arrival time, it must be placed at different depths in order to reduce, or eliminate, the mismatch in the time taken for them to arrive at the outputs.

Consider, for example, the vulnerable circuit illustrated by Fig. 7.7, which has the following input-output relations:

```
X = C AND D;
Y = X AND B;
O = Y AND A;
```

Due to the mismatch in the number of gates between the circuit inputs and the output, this circuit is vulnerable to FSA attacks.

Assume that, during synthesis, the boxed region in Fig. 7.7 is the extracted region reg. Each of the input nodes (A, B and X) could have a different depth assigned in the new region to be synthesized. To eliminate the mismatch in the delay, in the synthesized circuit, nodes A and B should be placed one gate closer to the output than node X. The pseudocode to compute such depths for all input signals is shown in Algorithm 11.

Algorithm 11 Computing the depths of inputs in $newReg$.

```
1: GETINPUTDEPTH (reg, lev, MinAr) {
2:   minMinAr ← minimum of MinAr[in] for all input in;
3:   for each (input signal in ∈ reg) {
4:     deltaArMismatch ← MinAr[in] − minMinAr
5:     newRegDepth[in] ← MAX(2, (lev − deltaArMismatch))
6:   }
7:   return newRegDepth;
8: }
```

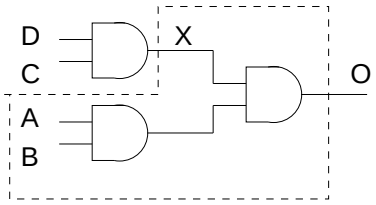


Figure 7.8: Example for a countermeasure circuit.

7.4.2 Generating a Candidate Circuit

Subroutine GENNEWREGION computes a candidate circuit $newReg$ that is functionally equivalent to reg on the set $testEx$ of examples. It first generates a logical formula and then invokes the SyGuS solver to search for a candidate solution. Recall that, in constructing the logical formula, we use a certain template circuit to ensure that the new circuit $newReg$ is less vulnerable to FSA attacks than the original circuit reg. A solution returned for the example in Fig. 7.7 is shown in the boxed area in Fig. 7.8, whose corresponding input-output relations are as follows:

```
X = C AND D;
W = A AND B;
O = X AND W;
```

If no solution can be found, the subroutine returns immediately. Then, inside Algorithm 9, the synthesis subroutine is invoked again for a smaller extracted region reg.

The logical formula generated by our method is

$$\Phi = \Phi_{reg} \wedge \Phi_{skel} \wedge \Phi_{EqI} \wedge \Phi_{EqO} \wedge \Phi_{testEx},$$

where the subformulas are defined as follows:

- Extracted region (Φ_{reg}): Encodes the logical function of the extracted circuit region.

- Skeleton (Φ_{skel}): Encodes the structure and possible logical functions of the new circuit to be synthesized.
- Equal input (Φ_{EqI}): Asserts that the inputs of the extracted and synthesized circuits have the same values.
- Equal output (Φ_{EqO}): Asserts that the outputs of the extracted and synthesized circuits have the same values.
- Test cases (Φ_{testEx}): Restricts the input signals of the circuits to the values in these test cases.

7.4.3 Verifying the Equivalence

After a candidate circuit is generated by solving the formula Φ, which is a satisfiability problem, we need to prove that the two circuits not only behave the same for the set $testEx$ of test examples, but also prove that they are functionally equivalent for all possible input values. Toward this end, we construct a new logical formula Ψ, whose satisfying assignment represents a test example that can differentiate the behaviors of reg and $newReg$. By showing that Ψ is unsatisfiable, we can prove that the two circuits are functionally equivalent.

The new logical formula generated by our method is

$$\Psi = \Psi_{reg} \wedge \Psi_{newReg} \wedge \Psi_{EqI} \wedge \Psi_{UneqO},$$

where the subformulas are defined as follows:

- Extracted region (Ψ_{reg}): Encodes the logical function of the extracted region.
- Candidate region (Ψ_{newReg}): Encodes the logical function of the synthesized circuit region.
- Equal input (Ψ_{EqI}): Asserts that the inputs of the extracted and synthesized circuits have the same values.

- Unequal output (Ψ_{UneqO}): Asserts that the outputs of the extracted and synthesized circuits have different values.

If the formula Ψ is unsatisfiable, we have proved that the two regions are equivalent; in such case, the synthesized circuit is returned. On the other hand, if the formula Ψ is satisfiable, the two regions are not equivalent. In such case, the solution to Ψ represents a new test example, which differentiates the behaviors of the two regions. In order to avoid the bad solution $newReg$ from being resynthesized in the future, the new test example is added to $testEx$ before GENNEWREGION is invoked again.

7.5 Experimental Results

In this section, we present the experimental results of applying our new method to a set of cryptographic circuits. Our method has been implemented using the SyGuS solver.

The benchmarks used in our experiments are circuits that implement parts of the AES and MAC-Keccak. Table 7.1 shows the statistics of these benchmarks. Columns 1 and 2 show the name and brief description of the benchmark circuit, respectively. Column 3 shows the size of the benchmark circuit. Column 4 shows the number of node count. Columns 5 and 6 show the number of input signals and output signals, respectively.

Among the circuits in Table 7.1, C1 and C2 are two different versions of the MAC-Keccak nonlinear Chi function [14], designed with masking countermeasures for power side-channel attacks. C3 and C4 are different implementations of the Chi function with masking countermeasures. C5 is the original unmasked Chi function from [14]. C6 and C7 are implementations of part of the AES, i.e., the Boyar-Peralta S-box nonlinear invg4 and mul4 functions, respectively, from [17]. C8 is the combination of all the S-box nonlinear functions in [17]. C9 is the complete AES PPRM1 implementation of the S-box in [62]. C10 is a different version of the AES Boyar-Peralta S-box implementation from [17].

Table 7.1: The statistics of the benchmark circuits used in our experimental evaluation.

Name	Description	Circuit Size	Node count	Input bits	Output bits
C1	MAC-Keccak nonlinear masked Chi function 1	59	35	10	1
C2	MAC-Keccak nonlinear masked Chi function 2	60	35	10	1
C3	Generated MAC-Keccak masked Chi function 1	67	44	10	1
C4	Generated MAC-Keccak masked Chi function 2	66	44	10	1
C5	Unmasked MAC-Keccak nonlinear Chi function	10	6	3	1
C6	AES S-Box design of nonlinear invg4 function	15	83	4	4
C7	AES S-Box design of nonlinear mul4 function	14	63	8	4
C8	AES S-Box single round nonlinear functions	41	209	8	8
C9	Complete AES PPRM1 S-box design	1045	8054	8	8
C10	Complete AES Boyar-Peralta S-box design	276	156	8	8

We have conducted the experimental evaluation in order to answer the following research questions:

- Can our method synthesize more efficient countermeasures against FSA attacks compared to existing techniques such as buffer insertion?
- Can our method robustly handle cryptographic circuits of practical size and complexity?

7.5.1 Experimental Results

We evaluated our method on all benchmarks. During the evaluation, we used the AND, XOR, OR and NOT gates as logic gates allowed in $GateSyn$. We assumed that all logical gates are designed with the same propagation delay. We set the depth of the synthesized circuit (lev) to 3, which allows for the synthesis of a circuit region of up to 7 gates for each invocation of the synthesis subroutine. We ran all experiments on a computer with a 3.4 GHz Intel i7-2600 processor, 4 GB RAM, and a 64-bit Ubuntu operating system.

Table 7.2 shows the results of our experiments. Here, we first compare the performance of our new method and the buffer insertion method. Recall that the prior technique relies on inserting buffers in the circuit to eliminate mismatch in the signal arrival time for each gate, whereas our method achieves the same goal by generating an entirely different circuit implementation. In the results table, Column 1 shows the benchmark name. Column 2 shows the number of nodes in the original

circuit. Columns 3 and 4 show the number of nodes in the new circuit obtained by buffer insertion and the increase in number of nodes, respectively. Columns 5 and 6 show the number of nodes in the new circuit obtained by our method and the increase in the number of nodes, respectively.

Table 7.2: Synthesis results.

Name	**Original**	**Buffer insertion alg.**		**New Synthesis alg.**	
	Nodes count	**Synthesized nodes**	**Nodes increase**	**Synthesized nodes**	**Nodes increase**
C1	35	51	45.71%	42	20.00%
C2	35	48	37.14%	40	14.29%
C3	44	54	22.73%	48	9.10%
C4	44	59	34.09%	45	2.27%
C5	6	9	50%	9	50%
C6	83	134	61.45%	98	18.07%
C7	63	79	25.40%	73	15.87%
C8	209	292	39.71%	244	16.75%
C9	8054	77717	864.90%	8943	11.04%
C10	156	9585	6044%	370	137.2%

The results demonstrate the effectiveness of our new method in synthesizing more compact countermeasures against FSA attacks. Compared to the buffer insertion method, the circuits produced by our method do not need as many additional gates. For example, our new circuit for C9 has only 11.04% more nodes, whereas the circuit produced by the buffer insertion method has 864.9% more nodes than the original circuit.

7.5.2 Detailed Statistics

Table 7.3 shows the detailed statistics of our countermeasure synthesis process. Column 1 is the name of the benchmark. Column 2 is the number of calls to the SyGuS solver, which attempts to computes the new region for a given region. Column 3 is the number of successful SyGuS solver calls, which found a new circuit. Column 4 is the number of unsuccessful SyGuS solver calls, which failed to find a new circuit — in such cases, the size of the extracted circuit had to be reduced before invoking the SyGuS solver again. Column 5 is the reduction in the number of nodes in our synthesized circuit compared the that of the buffer insertion method. Column 6 is the time (in seconds) spent by our method.

Table 7.3: Statistical data.

Name	Synthesis iterations	Successful iterations	Failed iterations	Nodes reduction	Run time [s]
C1	7	7	0	17.65%	1.22
C2	5	5	0	16.67%	0.10
C3	4	4	0	11.11%	0.09
C4	2	2	0	23.73%	0.06
C5	4	3	1	0%	0.13
C6	23	23	0	26.87%	0.48
C7	12	12	0	7.60%	0.26
C8	47	47	0	16.44%	1.11
C9	2627	2627	0	88.49%	412.32
C10	219	217	2	96.14%	13.74

Our results show that, for most of the benchmarks, the number of failed SyGuS calls is 0 for $lev = 3$, which leads to a reduction in total countermeasure synthesis time. This is advantageous not only because failed synthesis attempts are a waste of time but also because, in general, failed SyGuS calls take significant more time than successful SyGuS calls. The reason is that solving UNSAT instances are usually more difficult than solving SAT instances. Therefore, in practice, the key to achieve a significant runtime reduction is to accurately estimate an important parameter: the number of nodes to be included in the extracted region before invoking the SyGuS solver.

We also notice from Table 7.3 the scalability of our tool. The partitioned method has made the SyGuS-based countermeasure synthesis process more scalable; the overall runtime increases only moderately as the circuit size increases. Furthermore, our new method is effective in reducing the size of the new circuit when compared to the prior techniques. As the circuit size increases, the saving by our new method in terms of the number of added nodes also increases significantly compared to the buffer insertion method.

7.6 Related Work

As we have mentioned earlier, our method is the first inductive synthesis-based method for synthesizing FSA countermeasures for cryptographic circuits. Since it is based on inductive synthesis, our method has the potential to search within a significantly larger design space than prior techniques.

Ghalaty et al. [35] proposed a method for implementing FSA countermeasures based on the addition of delay elements at the input of certain gates in the circuit to equalize the path delays from all the sensitive inputs to the output. It can lead to countermeasures with more additional gates than ours, as we have demonstrated in experiments. Furthermore, their method does not eliminate mismatch in the arrival time of the input signals for all logical gates. In particular, it ignores XOR gates. After analyzing their method, we have found that their countermeasure could still be vulnerable to FSA attacks if the attacker uses the data dependency of hamming distance of successive inputs.

Endo et al. [34] proposed another countermeasure to defend against FSA attacks based on adding a configurable buffer circuitry to delay the propagation of the output signals from the cryptographic module. However, the method is a post-silicon solution and therefore is expensive to implement in practice, since the delay period needs to be configured after the chip is manufactured. To configure the delay, they first measure the delay needed for securing the manufactured cryptography module and then store the delays in an on-chip memory. They implemented the proposed countermeasure only for the benchmark C9, and reported a gate overhead of 10% to 16%. However, note that their countermeasure was designed manually, whereas in our method, the countermeasure is generated fully automatically.

Furthermore, both of the existing techniques [35, 34] incur a significant amount of area overhead and performance degradation in terms of the operating frequency. In contrast, our new method, due to its capability of discovering entirely new circuit implementations, has the potential to significantly reduce the area overhead and performance degradation. In some cases, our method can actually reduce the area overhead and improve the performance, as shown in the example in Fig. 7.2 – Fig. 7.4.

7.7 Summary

We have presented a new method for synthesizing countermeasures to defined against fault sensitivity analysis-based side-channel attacks. Our method relies on inductive synthesis to search for a new circuit that is functionally equivalent to the original circuit and at the same time is FSA resistant. It has the potential to discover more compact and efficient countermeasure implementations than the prior techniques such as buffer insertions. We have implemented the new method and evaluated it on a set of cryptographic circuits. Our experiments show that the use of partitioned synthesis approach can scale up our method to circuits of large size. For future work, we plan to evaluate our synthesized countermeasures on real devices to assess its effectiveness in defending against FSA attacks.

Chapter 8

Conclusions

In this dissertation, I have presented a set of automated techniques for improving the reliability and security of hardware and software code in critical embedded applications. Experimental results show that all the proposed methods are indeed effective when applied to practical benchmarks.

8.1 Summary

In Chapter 3, we proposed a new inductive program synthesis-based method for optimizing the fixed-point arithmetic computation software executed on embedded processors with relatively small register bitwidths. Our method guarantees that the new software is functionally equivalent to the original one, but without overflow and underflow errors.

In Chapter 4, we proposed a new formal verification method to check for sensitive information leakage in cryptographic software via power side-channels. In contrast to the existing method, that is capable only of checking whether the intermediate computation results are logically dependent on some random bits, our new method checks for statistical independence, which is more accurate.

In Chapter 5, we proposed a new method for quantifying the strength of masking countermeasures against power side-channel attacks. Our evaluation, based on measuring the power traces on real

devices, showed that our method is accurate enough for practical use.

In Chapter 6, we proposed a new method for automatically generating masking countermeasures for cryptographic software code vulnerable to power side-channel attacks. We have demonstrated the method efficiency and its capability of synthesizing countermeasures more compact than those proposed by cryptographic experts.

In Chapter 7, we proposed a new method for synthesizing provably secure cryptography circuit implementations to defend against fault sensitivity analysis (FSA) side-channel attacks. The synthesized countermeasures are shown to be extremely efficient when compared to those generated by the previous methods.

For all of the presented techniques, we have proposed partitioning approaches, which combine statistical analysis procedures within the methods in order to improve scalability. We have implemented our methods in software tools and demonstrated their effectiveness using realistic practical benchmarks.

In summary all the presented methods clearly show improvement over the previously used methods. The presented program synthesis methods would certainly ease development of embedded software and hardware.

8.2 Future Work

Although I explored many new research topics in this dissertation, and in some cases were the first to introduce automated verification and synthesis methods to these applications, this is just the beginning. There are still much more opportunities for improvement.

The work on optimizing fixed-point arithmetic computation code has clearly attracted the attention of the community. Extending the method to optimize floating-point code and expanding the set of supported instructions and theories could be beneficial. Optimizing other performance aspects, such as the execution time and power consumption, could be a good research direction as well.

Implementing cryptographic systems has always been a labor-intensive task. It is time-consuming and error-prone even for cryptographic experts. Developing more types of automation tools to assist in the software/hardware implementation process is desirable and promising. In addition to verification and synthesis, developing new methods for computer-aided error diagnosis [83, 13, 90, 89] and program repair [48] could be a good research direction.

Another open question is whether it is possible to automatically synthesize cryptographic software code that are better than those designed manually by experts. In this dissertation, I have made the first step by showing this is possible for countermeasures against power side-channel attacks and fault attacks. Still, there are many other types of side-channel attacks and related countermeasures, which require further investigation, e.g., synthesizing countermeasures against timing attacks.

All the techniques presented in this dissertation have overcome the scalability problems of the underlying verification and synthesis techniques by exploiting compositionality and using fast static analysis methods. This hybrid method makes it possible to apply the proposed techniques on large practical benchmarks. One potential disadvantage that arises from using these hybrid methods, however, is that the synthesized code may not be guaranteed optimum. If the underlying verification and synthesis techniques can be advanced to handle larger hardware and software code, it would certainly improve the quality of the synthesized code.

Bibliography

[1] Johan Agat. Transforming out timing leaks. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 40–53, 2000.

[2] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. A code morphing methodology to automate power analysis countermeasures. In *ACM/IEEE Design Automation Conference*, pages 77–82, 2012.

[3] Takuya Akiba, Kentaro Imajo, Hiroaki Iwami, Yoichi Iwata, Toshiki Kataoka, Naohiro Takahashi, Michal Moskal, and Nikhil Swamy. Calibrating research in program synthesis using 72,000 hours of programmer time. Technical report, MSR, 2013.

[4] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *International Conference on Formal Methods in Computer-Aided Design*, pages 1–17, 2013.

[5] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[6] Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *IEEE Symposium on Security and Privacy*, pages 141–153, 2009.

[7] Josep Balasch, Benedikt Gierlichs, Roel Verdult, Lejla Batina, and Ingrid Verbauwhede. Power analysis of Atmel CryptoMemory - recovering keys from secure EEPROMs. In *CT-RSA*, pages 19–34, 2012.

[8] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 394–403, 2006.

[9] Gilles Barthe, Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Leakage resilience against concurrent cache attacks. In *International Conference on Principles of Security and Trust*, pages 140–158, 2014.

[10] Ali Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. Sleuth: Automated verification of software power analysis countermeasures. In *Cryptographic Hardware and Embedded Systems*, 2013.

[11] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. A first step towards automatic application of power analysis countermeasures. In *ACM/IEEE Design Automation Conference*, pages 230–235, 2011.

[12] Ali Galip Bayrak, Nikola Velickovic, Paolo Ienne, and Wayne Burleson. An architecture-independent instruction shuffler to protect against side-channel attacks. *TACO*, 8(4):20, 2012.

[13] Mitra Tabaei Befrouei, Chao Wang, and Georg Weissenbacher. Abstraction and mining of traces to explain concurrency bugs. In *International Conference on Runtime Verification*, pages 162–177, 2014.

[14] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. Keccak implementation overview. URL: http://keccak.neokeon.org/Keccak-implementation-3.2.pdf.

[15] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, pages 513–525, 1997.

[16] Johannes Blömer, Jorge Guajardo, and Volker Krummel. Provably secure masking of AES. In *Selected Areas in Cryptography*, pages 69–83, 2004.

[17] Joan Boyar and René Peralta. A small depth-16 circuit for the aes s-box. In *SEC*, pages 287–298, 2012.

[18] Wihelm Burger and Mark Burge. *Digital Image Processing*. Springer, 2008.

[19] David Canright and Lejla Batina. A very compact "perfectly masked" S-Box for AES. In *ACNS*, pages 446–459, 2008.

[20] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, pages 398–412, 1999.

[21] Chris Lattner and Vikram Adve. The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.

[22] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[23] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. Synthesis of fixed-point programs. In *ACM international conference on Embedded software*, pages 1–10, 2013.

[24] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A tool for the static analysis of cache side channels. In *USENIX Security*, pages 431–446, 2013.

[25] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *International Conference on Computer Aided Verification*, pages 81–94. Springer, 2006. LNCS 4144.

[26] Hassan Eldib and Chao Wang. An SMT based method for optimizing arithmetic computations in embedded software code. In *International Conference on Formal Methods in Computer-Aided Design*, 2013.

[27] Hassan Eldib and Chao Wang. An SMT based method for optimizing arithmetic computations in embedded software code. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 33(11):1611–1622, 2014.

[28] Hassan Eldib and Chao Wang. Synthesis of masking countermeasures against side channel attacks. In *Computer Aided Verification, CAV*, 2014.

[29] Hassan Eldib and Chao Wang. Synthesis of countermeasures for fault attacks. In *(manuscript in preparation)*, 2015.

[30] Hassan Eldib, Chao Wang, and Patrick Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Trans. Softw. Eng. Methodol.*, 24(2):11:1–11:24, 2014.

[31] Hassan Eldib, Chao Wang, and Patrick Schaumont. SMT based verification of software countermeasures against side-channel attacks. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2014.

[32] Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. QMS: Evaluating the side-channel resistance of masked software from source code. In *ACM/IEEE Design Automation Conference*, 2014.

[33] Hassan Eldib, Chao Wang, Mostafa Taha, and Patrick Schaumont. Quantitative masking strength: Quantifying the power side-channel resistance of software code. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2015.

[34] Sho Endo, Yang Li, Naofumi Homma, Kazuo Sakiyama, Ohta Ohta, Daisuke Fujimoto, Makoto Nagata, Toshihiro Katashita, Jean-Luc Danger, and Takafumi Aoki. A silicon-level countermeasure against fault sensitivity analysis and its evaluation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, PP(99):1–10, 2014.

[35] Nahid Farhady Ghalaty, Aydin Aysu, and Patrick Schaumont. Analyzing and eliminating the causes of fault sensitivity analysis. In *DATE*, pages 1–6. IEEE, 2014.

[36] Louis Goubin. A sound method for switching between boolean and arithmetic masking. In *Cryptographic Hardware and Embedded Systems*, pages 3–15, 2001.

[37] Philipp Grabher, Johann Großschädl, and Dan Page. Cryptographic side-channels from low-power cache memory. In *International Conference on Cryptography and Coding*, pages 170–184, 2007.

[38] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 317–330, 2011.

[39] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 62–73, 2011.

[40] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 281–292, 2008.

[41] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 317–328, 2011.

[42] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. An AES smart card implementation resistant to power analysis attacks. In *ACNS*, pages 239–252, 2006.

[43] Franjo Ivančić, I. Shlyakhter, Aarti Gupta, M.K. Ganai, V. Kahlon, Chao Wang, and Z. Yang. Model checking C program using F-Soft. In *International Conference on Computer Design*, pages 297–308, October 2005.

[44] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE (1)*, pages 215–224, 2010.

[45] Susmit Kumar Jha. *Towards Automated System Synthesis Using SCIDUCTION*. PhD thesis, UC Berkeley, Nov 2011.

[46] Rajeev Joshi, Greg Nelson, and Keith H. Randall. Denali: A goal-directed superoptimizer. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 304–314, 2002.

[47] Marc Joye, Pascal Paillier, and Berry Schoenmakers. On second-order differential power analysis. In *Cryptographic Hardware and Embedded Systems*, pages 293–308, 2005.

[48] Sepideh Khoshnood, Markus Kusano, and Chao Wang. Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs. In *International Symposium on Software Testing and Analysis*, 2015.

[49] Seehyun Kim, Ki-Il Kum, and Wonyong Sung. Fixed-point optimization utility for c and c++ based digital signal processing programs. In *IEEE Trans. Circuits and Systems II*, volume 45, pages 1455–1464, 1998.

[50] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *International Cryptology Conference – CRYPTO'96*, pages 104–113, 1996.

[51] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *CRYPTO*, pages 388–397, 1999.

[52] Boris Köpf and Markus Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *IEEE Symposium on Computer Security Foundations*, pages 324–335, 2009.

[53] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. Automatic quantification of cache side-channels. In *International Conference on Computer Aided Verification*, pages 564–580, 2012.

[54] Bing Li, Chao Wang, and Fabio Somenzi. Abstraction refinement in symbolic model checking using satisfiability as the only decision procedure. *International Journal on Software Tools for Technology Transfer*, 7(2):143–155, 2005.

[55] Yang Li, Kazuo Sakiyama, Shigeto Gomisawa, Toshinori Fukunaga, Junko Takahashi, and Kazuo Ohta. Fault sensitivity analysis. In *CHES*, pages 320–334. Springer, 2010.

[56] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Sards*. Springer, 2007.

[57] Adolfo Anta Martinez, Rupak Majumdar, Indranil Saha, and Paulo Tabuada. Automatic verification of control system implementations. In *ACM international conference on Embedded software*, pages 9–18, 2010.

[58] Thomas S. Messerges. Securing the AES finalists against power analysis attacks. In *Fast Software Encryption*, pages 150–164, 2000.

[59] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. On the vulnerability of FPGA bitstream encryption against power analysis attacks - extracting keys from Xilinx Virtex-II FPGAs. *IACR Cryptology*, 2011.

[60] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the limits: A very compact and a threshold implementation of aes. In *EUROCRYPT*, pages 69–88, 2011.

[61] Sumio Morioka and Akashi Satoh. An optimized s-box circuit architecture for low power aes design. In *CHES*, pages 172–186. Springer, 2002.

[62] Sumio Morioka and Akashi Satoh. An optimized s-box circuit architecture for low power aes design. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 172–186. Springer, 2003.

[63] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In *Cryptographic Hardware and Embedded Systems*, pages 58–75, 2012.

[64] NIST. Keccak reference code submission to NIST's SHA-3 competition (Round 3). URL: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/documents/Keccak_FinalRnd.zip.

[65] Elisabeth Oswald, Stefan Mangard, Norbert Pramstaller, and Vincent Rijmen. A side-channel analysis resistant description of the AES S-Box. In *International Workshop on Fast Software Encryption*, pages 413–423, 2005.

[66] Christof Paar, Thomas Eisenbarth, Markus Kasper, Timo Kasper, and Amir Moradi. Keeloq and side-channel analysis-evolution of an attack. In *FDTC*, pages 65–69, 2009.

[67] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 275–286, 2012.

[68] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In *Advances in Cryptology ¡V EUROCRYPT 2013*, pages 142–159. Springer, 2013.

[69] Shehrzad Qureshi. *Embedded Image Processing on the TMS320C6000 DSP*. Springer, 2005.

[70] Radu Rugina and Martin C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *PLDI*, pages 182–195, 2000.

[71] Majumdar Rupak, Indranil Saha, and Majid Zamani. Synthesis of minimal-error control software. In *ACM international conference on Embedded software*, pages 123–132, 2012.

[72] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[73] Hikaru Sakamoto, Yang Li, Kazuo Ohta, and Kazuo Sakiyama. Fault sensitivity analysis against elliptic curve cryptosystems. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2011, Tokyo, Japan, September 29, 2011*, pages 11–20, 2011.

[74] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 305–316, 2013.

[75] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *International Conference on Computer Aided Verification*, pages 634–651, 2012.

[76] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 398–414, 2015.

[77] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodík, Vijay A. Saraswat, and Sanjit A. Seshia. Sketching stencils. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 167–178. ACM, 2007.

[78] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodík. Sketching concurrent data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 136–148, 2008.

[79] Armando Solar-Lezama, Rodric M. Rabbah, Rastislav Bodík, and Kemal Ebcioglu. Programming by sketching for bit-streaming programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 281–294, 2005.

[80] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[81] M. Taha and P. Schaumont. Differential power analysis of MAC-Keccak at any key-length. In *IWSEC*, 2013.

[82] Chao Wang, Gary D. Hachhtel, and Fabio Somenzi. *Abstraction Refinement for Large Scale Model Checking*. Springer, 2006.

[83] Chao Wang, Zijiang Yang, Franjo Ivancic, and Aarti Gupta. Whodunit? causal analysis for counterexamples. In *Automated Technology for Verification and Analysis*, pages 82–95, 2006.

[84] Chao Wang, Zijiang Yang, Franjo Ivancic, and Aarti Gupta. Disjunctive image computation for software verification. *ACM Trans. Design Autom. Electr. Syst.*, 12(2), 2007.

[85] Xilinx. Microblaze soft processor core. URL: http://www.xilinx.com/tools/microblaze.htm.

[86] Jianxin Xiong, Jeremy R. Johnson, Robert W. Johnson, and David A. Padua. Spl: A language and compiler for dsp algorithms. In *PLDI*, pages 298–308, 2001.

[87] Zijiang Yang, Chao Wang, Aarti Gupta, and Franjo Ivancic. Model checking sequential software programs via mixed symbolic analysis. *ACM Trans. Design Autom. Electr. Syst.*, 14(1), 2009.

[88] Randy Yates. *Fixed-point arithmetic: An introduction*. Digital Signal Labs, Technical Reference, 2013.

[89] Qiuping Yi, Zijiang Yang, Jian Liu, Chen Zhao, and Chao Wang. Explaining software failures by cascade fault localization. *ACM Transactions on Design Automation of Electronic Systems*, 2015.

[90] Qiuping Yi, Zijiang Yang, Jian Liu, Chen Zhao, and Chao Wang. A synergistic analysis method for explaining failed regression tests. In *International Conference on Software Engineering*, 2015.