Formal Verification for Security in IoT Devices

Keerthi K.[1], Indrani Roy[1], Aritra Hazra[2], and Chester Rebeiro[1]

[1] Indian Institute of Technology Madras, India
{keerthi, indrroy, chester}@cse.iitm.ac.in,
[2] Indian Institute of Technology Kharagpur, India
aritrah@cse.iitkgp.ernet.in

Abstract. Online detection of cyber-attacks on IoT devices is extremely difficult due to the limited battery and computational power available in these devices. An alternate approach is to shrink the attack surface in order to reduce the threat of attack. This would require that the device undergo more stringent security tests before deployment.
Formal verification is a promising tool that can be used to not only detect potential vulnerabilities but also provide guarantees of security. This chapter reviews several security issues that plague IoT devices such as functional correctness of implementations, programming bugs, side-channel analysis, and hardware Trojans. In each of these cases, we discuss state-of-the-art mechanisms that use formal verification tools to detect the vulnerability much before the device is deployed.

1 Introduction

The number of connected IoT devices has crossed 20 billion and expected to increase at a rate of 15% per year. IoT devices are typically battery operated, have low computing power, and less memory. Many of the devices have either no or tiny operating systems that have limited functionalities. Operating system functionalities are typically restricted to resource management for efficient energy utilization. Security features such as secure boot, trusted execution, and even memory protection are generally absent. These limiting features in IoT devices makes them vulnerable to a variety of cyber-attacks.

The attack surface for IoT devices is considerably large. An attacker may use one or more of the following attack vectors to compromise a device.

– First, there can be a weakness in the functionality of a device. For example, an operation that was intended to be present but is either absent in the implementation or not fully complete. For example, there may be an absence of strong authentication methods or an absence of meta-level encryption procedures.
– Even if the intended security operations are implemented in the device, there may be flaws in the implementation. This is especially a problem with cryptographic algorithms. The security guarantees of cryptographic algorithms are well studied and investigated. The weakest link is generally not due to

the mathematical underpinnings of the algorithms but rather their implementations. The huge state space present in cryptographic implementations, makes detecting these flaws difficult.

– Programming bugs such as buffer overflows, arithmetic overflow and underflows, and format-string vulnerabilities present in the implementations, can be used to craft malicious payloads that can subvert execution leading to security breaches.
– Trojans present in the hardware or software can provide unauthorized access to the system or lead to information leakage. These Trojans are introduced due to multiple third parties involved in the design and manufacture of IoT devices. They are very difficult to detect but provide an easy vector for an attacker.
– IoT devices are also vulnerable to multiple physical attacks such as fault injection attacks, differential power analysis, and timing attacks. These attacks would require the attacker to have physical access to the device, disturb the device operations by injection of a fault, or passively monitoring the device's side-channels such as its power and energy consumption, electro-magnetic radiation, and execution time to glean secret information.

This huge attack surface is a serious concern especially because many of the IoT devices are used in cyber-physical systems such as process-control, smart-grid, and medicals systems. A compromised device in any of these critical infrastructures could lead to considerable losses.

Due to the constrained resources in an IoT device, detecting cyber-attacks is only possible from outside the device. Two potential directions to detect malware outside the device is by monitoring the network for malware signatures or by using side-channels such as the device's power consumption, execution time, and electro-magnetic radiation to identify patterns that would indicate an attack. Both techniques face considerable challenges. For example, current side-channel analysis techniques requires a measurement setup and therefore only possible in a laboratory environment. Network monitoring techniques on the other hand will not be effective for attacks that do not transmit much over the network or those that camouflage their network activities. Furthermore both techniques will not be able to detect zero-day exploits.

An alternate approach to achieving security in IoT devices is to prevent rather than detect. This would require that the various attack vectors described above are eliminated at design time there by reducing the attack surface. Traditionally this is done by good design practices, use of secure coding techniques, static analysis, and extensive testing. However, these methodologies cannot provide guarantees of security; stronger mechanisms would therefore be required. One promising direction is the use of formal verification to provide security guarantees. Formal verification is a technique to mathematically ascertain the correctness of designs using a diverse set of mathematical and logical methods. Model checking [17] and theorem proving [23] procedures are often used to ensure the accuracy of implementations. In model checking, a model of a system is exhaustively and automatically verified with respect to a given specification;

whereas in theorem proving, the system characteristics are derived mathematically and solved using automated reasoning techniques to infer the correctness of the system.

The main drawback of using a theorem prover is that the user has to explicitly provide the design and specification characteristics as algebraic constraints or theorems. Model checkers, on the other hand, can act automatically over the implementation with the given specifications and formulate SAT clauses from the design behavior. It is, therefore, more effective and easier to use a model checker as compared to a theorem prover for validating large implementations, such as crypto-designs.

In this chapter, we shall discuss various applications of formal verification in order to improve an IoT device's security. We would use formal verification to `(a)` prove the correctness of implementations with respect to its formal specification. As a case study, we would consider the correctness verification of a multi-precision library used for public-key cryptographic algorithms such as ECC and RSA. `(b)` We would then prove the absence of programming vulnerabilities, such as buffer overflows, in the multi-precision library. Multi-precision library implementations are especially interesting for formal verification due to their critical usage with security sensitive aspects in the device, and their extremely complex design space where comprehensive testing becomes practically impossible.

The chapter would also discuss other state-of-the-art research in the use of formal verification for device security. In particular, `(c)` the chapter discusses the use of formal verification to validate physical attack countermeasures. While applying countermeasures for physical attacks such as side-channel analysis is easy, proving their effectiveness is considerably more challenging. Formal verification would help considerably to achieve these security proofs. `(d)` Another application of formal verification is to detect the presence of hardware Trojans in designs. We discuss a recent work which demonstrates the use of model checking tools to identify a Trojan that leaks sensitive information from the device. `(e)` Finally we discuss the use of formal verification to ensure completeness of security goals such as meta-level authentication and encryption.

The organization of this chapter is as follows: Section 2 provides the background about symbolic model checking covering SAT and BDD based model checking. The section also describes a model checking tool called CBMC for C based bounded model checking [18]. Section 3 describes the use of model checking to verify that an implementation is correct with respect to its formal specification. A case study of a multi-precision library used for ECC and RSA is verified. Section 4 describes program vulnerabilities and presents the use of CBMC as a tool to detect such vulnerabilities. Section 5 presents the use of formal verification for side-channel countermeasures, while Section 6 describes the use of formal verification for hardware Trojan detection. Section 7 shows how formal verification can be used to identify meta-level authentication issues, while the final section has the conclusion.

2 Background: Symbolic Model Checking

The term *symbolic model checking* is popularly interpreted as BDD-based model checking, however any model checking technique that works on a symbolic representation of the implementation can be called *symbolic* model checking. There are mainly two kinds of symbolic methodology found in the literature; namely *BDD-based model checking* and *SAT-based model checking*.

2.1 BDD-based Model Checking

Binary Decision Diagrams (BDDs) [8] are compact canonical representations of Boolean functions. BDDs utilize self-similarity in the decision trees based on Shanon's expansion to give a more compact representation. Ken McMillan first proposed model checking algorithms using BDD in his famous doctoral thesis [24]. The use of BDDs in model checking was instrumental in bringing the technology into practice. Experimental results showed that the BDD-based approaches were able to handle 10^{20} states and beyond [10] – which was unthinkable with algorithms that work on explicit representations of the state space.

To perform BDD-based model checking [14], a BDD representation, $\mathcal{Z}$, for the temporal property $\neg\varphi$ is created (when, φ be the temporal formula of interest). Then, the product of $\mathcal{Z}$ with the BDD for the transition relation of the design-under-test (DUT) is computed. Let the BDD for the product be $\mathcal{P}$. In the final step of the model checking, the strategy is to check whether $\mathcal{P}$ is empty, that is, whether the product has any fair path [9], [15].

2.2 SAT-based Model Checking

SAT is the traditional short form for the Boolean satisfiability problem. Given a Boolean formula f, the problem is to determine whether f is satisfiable, that is, whether there exists any valuation of the variables in f, under which f evaluates to $True$. Verification methods based on the SAT problem have recently emerged as a promising solution. Dramatic improvements in SAT solver technology over the past decade have led to the development of several powerful SAT solver tools [21], [28], [29], [31]. Verification methods based on these tools have been shown to push the boundaries of functional verification in terms of both capacity and efficiency, as reported in several academic and industrial case studies [2], [3], [7]. This has fueled further interest and intense research activity in the area of SAT-based FPV.

Bounded Model Checking (BMC) [13] based on SAT methods was introduced by Biere et al. in [5], [6], [13] and is rapidly gaining popularity today as a complementary technique to the existing BDD-based model checking. Given a temporal logic property, φ, to be verified on a finite state transition system $\mathcal{M}$, the essential idea is to search for counter-examples to φ in the space of all executions of $\mathcal{M}$ whose length is bounded by some finite integer k.

The problem is formulated by constructing the following propositional formula:

$$f^k \;=\; I \;\wedge\; \bigwedge_{i=0}^{k-1} R(S_i, S_{i+1}) \;\wedge\; (\neg\varphi^k)$$

where I is the characteristic function for the set of initial states of $\mathcal{M}$, $R(S_i, S_{i+1})$ is the characteristic function of the transition relation, relating the variables in S_i with those in S_{i+1}, of $\mathcal{M}$ for time step i. Thus, the formula $\left(I \wedge \bigwedge_{i=0}^{k-1} R(S_i, S_{i+1})\right)$ precisely represents the set of all executions of $\mathcal{M}$ of length k or less, starting with an initial state. $\neg\varphi^k$ is a formula representing the condition that φ is violated by a bounded execution of $\mathcal{M}$ of length k or less. Hence, f^k is satisfiable iff there exists an execution of $\mathcal{M}$ of length k or less that violates φ. f^k is typically converted to conjunctive normal form (CNF) and is solved by a conventional SAT solver.

Due to the success of SAT solvers in bounded model checking, there has been growing interest in their use for unbounded model checking. Few of the recent works in this direction can be found in [22], [25].

2.3 CBMC: The Formal Verification Tool

There are various academic as well as industrial tools available for formal verification of the system. In this work, we will evaluate the security attributes by symbolic model checking approach with the help of tool named **CBMC** which is a C based model checker [18]. CBMC takes two inputs: (i) the program to be verified (written in C or C++), and (ii) the formal specification, as shown in Figure 1. The loops in the program (`p`) are first unwound and then a Boolean model `Q(p)` is obtained, which is checked for satisfiability with the negation of the specification ($\neg$`q`) using an in built SAT solver. If the model is satisfied, which means that the negation of the specification is satisfied, then the verification fails. CBMC reports this failure with a counter-example. If the model is not satisfiable, then for all the possible combinations of the input, the specification conditions are correct, therefore the program is verified to be correct based on the given specifications.

3 Correctness of Crypto Implementations

An important security requirement for any program is to ensure its correctness with respect to a specification. This becomes even more critical when the program in consideration is an implementation of a cryptographic algorithm. An error in a crypto-implementation, could be exploited to leak secrets such as the cryptographic key. Extensive testing is the time-honored way of checking the correctness of a crypto implementation. This however cannot provide guarantees because of the huge state space of crypto-implementations. For example, the state space of a typical implementation of an elliptic curve cryptographic scheme is in the order of 2^{256}. This is too huge a space to exhaustively test. A

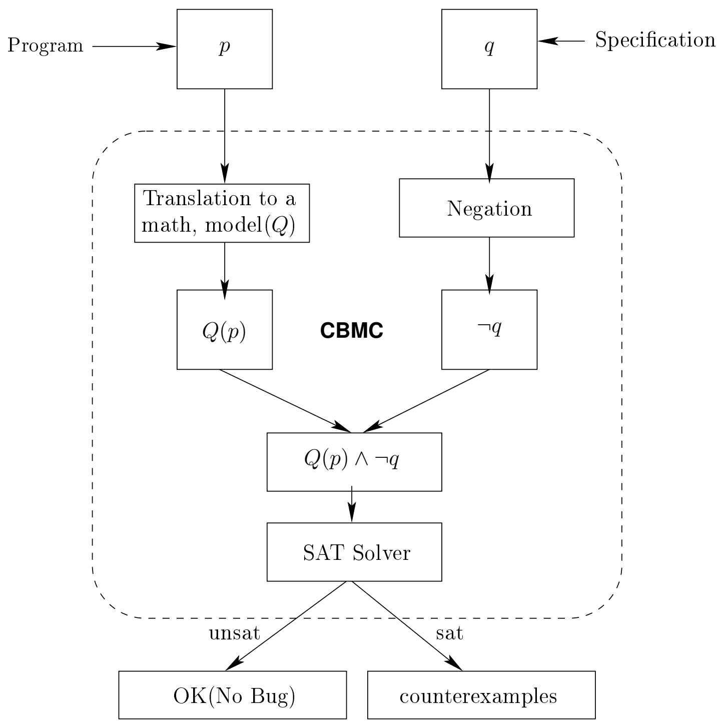

Fig. 1: Flow of a C program verification using CBMC, showing a SAT solver that checks the correctness of program p with respect to the specification q. The counter example generated gives the program inputs for which specification q fails [16].

more complete technique is to use formal verification, which generates stronger guarantees of correctness based on a *functional specification* of the algorithm.

There have been several efforts to prove the functional correctness of crypto-implementations using formal verification. For example, ciphers like AES, MARS, Twofish, RC6, Serpent, IDEA, and TEA were considered in [19]; several hash functions and block ciphers in [4] and [32]. For public-key ciphers, functional correctness has been discussed in [1] and [12].

In this section we discuss the formal verification of a multi-precision library. This library forms the base over which public-key algorithm like ECC and RSA are built.

3.1 Correctness of a multi-precision library

A multi-precision library for public-key cryptography includes implementations of finite-field operations such as addition, subtraction, multiplication etc. These implementations could have many flaws, most of which can be detected by formal verification.

The multi-precision library we consider is written in ‘C’. We use the model checking tool CBMC (ANSI-C Bounded Model Checking) [18] as a tool for verification. The approach we follow is to use a hierarchical verification technique to handle the scalability issues and the huge state space of the library.

Implementation Aspects and Notations. A central data-structure used to define multi-precision numbers is the `bignum_t` structure, which comprises of an array called `digits` used to store the multi-precision number and `sign` to store its sign.

```
typedef struct{
  word digits[MAXDIGITS];
  int  sign;
}bignum_t;
```

The multi-precision elements is represented using the macro `MAXDIGITS` - based on the number of digits in the multi-precision number. It depends on the `word` length of the processor executing the multi-precision library. For instance to represent a field element in $\mathbb{F}_{256}$ on a micro-controller platform with a 16 bit word size, `MAXDIGITS` would be defined as $\lceil 256/16 \rceil = 16$.

The finite field elements are represented using the above structure as follows: $A = (a_{15}, a_{11}, a_{10}, \cdots, a_2, a_1, a_0)$, where a_i $(15 \leq i \leq 0)$ are word sized numbers stored in `digits` in the `bignum_t` structure, a_{15} is the most significant digit, while a_0 is the least significant digit. Multi-precision operations are denoted in capitals with a ‘$*$’ on top. For example $A \overset{*}{+} B$ represents multi-precision addition, while the operations over the digits in the multi-precision number are in the standard representation, such as $a_i + b_i$ for digit addition. In the remainder of this section we demonstrate several cases of formal verification of multi-precision operations.

Case 1 (Multi-Precision Addition): Let A and B be two multi-precision numbers with n digits. To perform multi-precision addition ($S = A \overset{*}{+} B$), we add the digits of A and B as shown in Equation 1.

$$\begin{aligned} \mathrm{c}_{-1} &= 0 \\ (\mathrm{c}_i, s_i) &= a_i + b_i + \mathrm{c}_{i-1} \qquad (0 \leq i \leq n-1)\ , \end{aligned} \tag{1}$$

where s_i holds the sum of digits and c_i the carry of each digit addition. The output is stored in $(\mathrm{c}_{n-1} || \, S)$.

Verification of Multi-Precision Addition. In Equation 1, we first verify that the carry and sum of each digit (*i.e.* c_i, s_i) is correct. Each digit is of 8, 16, or 32 bits depending on the execution platform. To perform the verification, we provide conditions to CBMC (in the form of assertion statements) that will evaluate to *true* if the addition is correct and *false* otherwise. The condition to verify the addition of the digits of A and B considering the carry that occurs from one digit to the next, *i.e.* verifying $(\mathrm{c}_i, s_i) = (a_i + b_i + \mathrm{c}_{i-1})$ is

$$((\mathrm{c}_i, s_i) - (b_i + \mathrm{c}_{i-1}) = a_i) \quad 0 \leq i \leq n-1; \qquad \mathrm{c}_{-1} = 0 \ . \tag{2}$$

CBMC will perform the multi-precision addition $A \overset{*}{+} B$ and verify digit-by-digit checking exhaustively over the possible valuations of A and B, for the above condition to be satisfied. If a failure is obtained, it means that the specification failed for some values of A and B. CBMC will return the values which caused the failure. This counter-example is the proof that CBMC provides of a verification failure.

Listing 1.1: Implementation of multi-precision addition

```
void BN_uadd (bignum_t *S, bignum_t A, bignum_t B)
     {
        int i, j;
        word c = 0;
        S->sign = 1;     /* sign of result forced to positive */
        for( i = 0; i < MAXDIGITS; i++){
           S->digits[i] = A.digits[i] + c;
           c = ( S->digits[i] <  c );
           S->digits[i] = S->digits[i] + B.digits[i];
           c = c + ( S->digits[i] < B.digits[i] );
        }
        while( c!=0 ){
               S->digits[i] = S->digits[i] + c;
           c = (S->digits[i] < c);
           i = i + 1;
        }
     }
```

Listing 1.2: Specification for multi-precision addition

```
int BN_uadd_specification(bignum_t S, bignum_t A, bignum_t B)
    {
       int j;
       word z, c = 0;
       for(j = 0; j < MAXDIGITS; j++) {
          z = (S.digits[j] < c);
          S.digits[j] = S.digits[j] - c;
          c =  (S.digits[j] < B.digits[j]) + z;
          S.digits[j] = S.digits[j] - B.digits[j];
          if (S.digits[j] != A.digits[j]) return (false);
       }
       return (true);
    }
```

Listing 1.3: Verification of multi-precision addition

```
void BN_uadd_verify ()
     {
           bignum_t S,A,B;
           int nondet_int();
           A.sign = nondet_int();
           B.sign = nondet_int();
           __CPROVER_assume(A.sign==1 && B.sign==1);

        BN_uadd(&S,A,B); /* Compute S = A + B */
        assert(BN_uadd_specification(S,A,B));
     }
```

Step 1

```
P = 1;
j = 0;
if(j < MAXDIGITS) {
 //body
 z = (S.digits[j] < c);
 S.digits[j] = S.digits[j] - c;
 c =  (S.digits[j] < B.digits[j]) + z;
 S.digits[j] = S.digits[j] - B.digits[j];
 if (S.digits[j] != A.digits[j]) P = 0;
 j = j + 1;
 if(j < MAXDIGITS){
    //body
     j = j + 1;
     ...
     if(j < MAXDIGITS){
        //body
        j = j + 1;
        assert(!(j < MAXDIGITS));
      }
     ...
   }
 } assert(P==1);
```

Step 2

```
P1 = 1;
j1 = 0;
if(j1 < MAXDIGITS){
  //body
  z1 = (S1.digits[j1] < c1);
  S2.digits[j1] = S1.digits[j1] - c1;
  c2 = (S2.digits[j1] < B1.digits[j1]) + z1;
  S3.digits[j1] = S2.digits[j1] - B1.digits[j1];
  if (S3.digits[j1] != A1.digits[j1]) P2 = 0;
  j2 = j1 + 1;
  if(j2 < MAXDIGITS){
     //body
     j3 = j2 + 1;
     ...
     if(jn−1 < MAXDIGITS){
        //body
        jn = jn−1 + 1;
        assert(!(jn < MAXDIGITS));
      }
     ...
  }
} assert(P2==1 && P3==1 && ... && Pn==1);
```

Step 3

```
C := (P1 = 1)
    ∧ (j1 = 0)
    ∧ z1 = ((j1 < MAXDIGITS)∧(S1.digits[j1] < c1)) ? 1 : 0
    ∧ S2.digits[j1] = (j1 < MAXDIGITS) ? S1.digits[j1] - c1 : S1.digits[j1]
    ∧ c2 = ((j1 < MAXDIGITS)∧(S2.digits[j1] < B1.digits[j1])) ? (1+z1) : z1
    ∧ S3.digits[j1] = (j1 < MAXDIGITS)?(S2.digits[j1]-B1.digits[j1]) : S1.digits[j1]
    ∧ P2 = ((j1 < MAXDIGITS)∧(S3.digits[j1] ≠ A1.digits[j1])) ? 0 : P1
    ∧ j2 = (j1 < MAXDIGITS) ? (j1 + 1) : j1
    ∧ ...
P := (P2 == 1 ∧ P3 == 1 ∧  . . . Pn  ==  1)
```

Fig. 2: Given the function `BN_uadd_specification`, Step 1 shows how CBMC unwinds the loop and the Step 2 shows the renaming of the variables in function body. Step 2 is converted to Boolean formula, with set of constraints $\mathcal{C}$ and properties $\mathcal{P}$ is given in Step 3

Listing 1.1 gives the implementation of multi-precision addition shown in Equation 1. Functional correctness of the implementation is done by digit addition of the result with operand B as given in Equation 2. The implementation aspects of the addition specification is shown in Listing 1.2. Implementation `BN_uadd_verify` in Listing 1.3 performs the formal verification, which invokes `BN_uadd` and `BN_uadd_specification`. CBMC performs the following steps for verification:

1. CBMC unwinds all the loops in the program, based on the number of iterations specified using command '`--unwind N`'. Each copy of the loop is replaced by an `if` statement to check the terminating conditions and at the end of `N` copies, an *unwinding assertion* is added by CBMC to avoid further iterations. The unwinding of `BN_uadd_specification` in listing 1.2 is shown

in Step 1 of Figure 2. We have used an additional variable P to represent the return value of the function, which is initialized to 1 and made 0 if *false* is returned (corresponding to Line 10 of `BN_uadd_specification`). The assertion in the last line of Step 1 corresponds to the assertion in Line 9 of the function `BN_uadd_verify`.

2. The next step is to rename the program variables as shown in Figure 2 Step 2, which is transformed to SSA (static single assignment). For example `j = j + 1` is converted to $\mathtt{j_2} = \mathtt{j_1} + \mathtt{1}$. In Step 2, each `if` block renames the variable P to P_1, P_2, ..., P_n. Therefore, the assertion statement should check all the n values of P instead of one variable.
3. In the third step, CBMC uses set of rules to convert SSA statements are converted to Boolean formula [18]. The rules define a set of constraints ($\mathcal{C}$) and set of properties ($\mathcal{P}$). These are shown in Step 3 of Figure 2.
4. The SAT solver in CBMC, try to solve $\mathcal{C} \wedge \neg\mathcal{P}$ and returns counter-example if solution found, which means that verification is failed; otherwise it returns verification successful

Case 2 (Multi-Precision Subtraction): Let A and B be two multi-precision numbers of n digits each. To perform multi-precision subtraction ($D = A \overset{*}{-} B$), we subtract the digits of A and B as shown in Equation 3.

$$\begin{aligned} \mathrm{b}_{r_{-1}} &= 0 \\ d_i &= (\mathrm{b}_{r_i}, (a_i - \mathrm{b}_{r_{i-1}})) - b_i \qquad (0 \leq i \leq n-1) \ , \end{aligned} \tag{3}$$

where d_i holds the difference between digits and b_{r_i} holds the borrow of each individual digits. The output of the result is stored in $\mathbf{d}$.

Verifying Multi-Precision Subtraction. To verify subtraction, we assume that multi-precision addition has already been verified and proven to be correct. Thus, verifying $D = A \overset{*}{-} B$ is simply done using $A = D \overset{*}{+} B$. In the specification function for multi-precision subtraction, we invoke multi-precision addition (Equation 1) with operands D and B. The result is verified to be equal to the original value of A using an `assert` statement.

Case 3 (Multi-Precision Left-Rotation): The multi-precision left-rotate takes a multi-precision number A having b bits and an integer m. It shifts A left by m bits and the bits that fall off at the most significant end are inserted in the least significant end. This is represented as $A_r = A \overset{*}{\lll} m$.

Verifying Multi-Precision Left-Rotation. We verify multi-precision left-rotate, by checking the bit position before and after rotation. *i.e.*, the specification given to CBMC checks that m most significant bits of A is shifted to corresponding least significant positions in A_r. The remaining $(b - m)$ bits of A are left shifted by m bits in A_r.

For example: let $A = (a_{15}, a_{14}, a_{13}, \cdots, a_2, a_1, a_0)$, be a multi-precision number on a 16-bit platform and a_i $(11 \leq i \leq 0)$ be 16 bit words in A. $(A \lll m)$

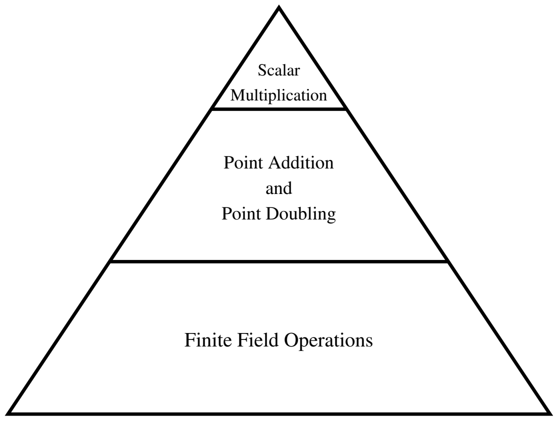

Fig. 3: Pyramid-like structure of an ECC implementation, where the bottom layer finite field operations include multi-precision addition, multiplication etc

where $m = 16$, will result in $A_r = (a_{14}, a_{13}, a_{12}, \cdots, a_1, a_0, a_{15})$. Verification is done by checking whether all the bits are shifted exactly m bits to lower position and also the most significant m bits are shifted to least significant m bits.

3.2 Verifying an ECC Implementation

Elliptic curve cryptography implementations have a pyramid-like structure as shown in Figure 3. The operations in the base of the pyramid are multi-precision algorithms like addition, subtraction, multiplication, and inversion; all done in the underlying finite field of the Elliptic curve. The verification of the point addition and doubling functions is thus carried out to ensure that the correct invocation of all the lower level finite field functions.

The verification of ECC scalar multiplication is done in a similar manner, since the algorithm invokes the point addition and point doubling functions. Given that point addition and point doubling have already been verified to be correct, the scalar multiplication involves the number of times point addition and point doubling is computed based on the scalar.

Results. This section contains the results of the verification of our Elliptic curve crypto-library. Implementation of four out of the five NIST specified Elliptic curves over $\mathbb{F}_{192}$, $\mathbb{F}_{224}$, $\mathbb{F}_{256}$, and $\mathbb{F}_{384}$ [34] were verified. CBMC Version 5.7 [18] on an Ubuntu 14.04 Linux machine on a quad core Intel i5-3340 CPU @ 3.10GHz was used as the platform for the study. In all cases, we assume a 16 bit word size that forms a digit used to partially represent a multi-precision number. Table 1 shows the verification time for the four different NIST specified curves. Figure 4 shows the verification time and execution time for 4 NIST specified

Table 1: Time taken (in hours) by CBMC to verify scalar multiplication operation of 4 different curves, assuming a 16 bit word size.

Function		Description	Time(in hours)
Scalar Multiplication	192-bit	$\mathbb{F}_{192}$	23
	224-bit	$\mathbb{F}_{224}$	34
	256-bit	$\mathbb{F}_{256}$	64
	354-bit	$\mathbb{F}_{384}$	183

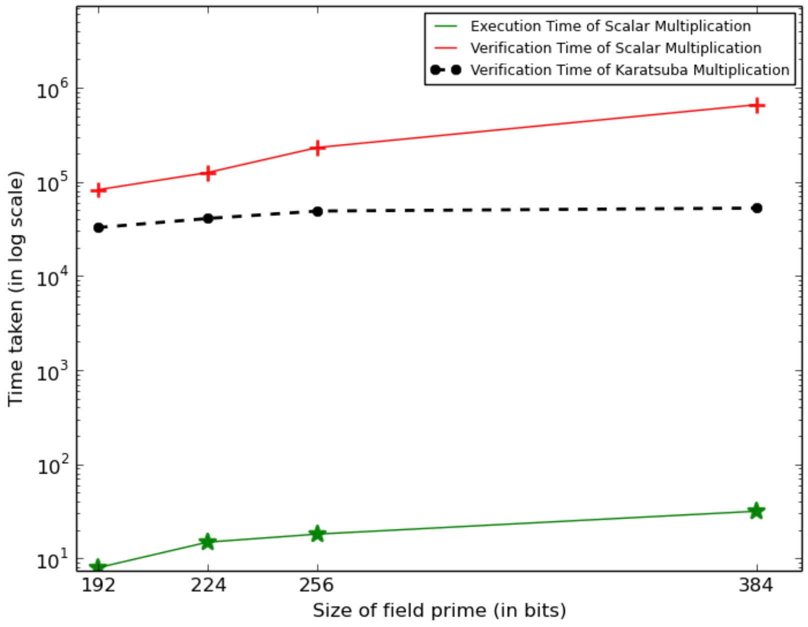

Fig. 4: Time taken in log scale for execution and verification of scalar multiplication for the 4 NIST Elliptic curves $\mathbb{F}_{192}$, $\mathbb{F}_{224}$, $\mathbb{F}_{256}$, and $\mathbb{F}_{384}$.

Elliptic curves $\mathbb{F}_{192}$, $\mathbb{F}_{224}$, $\mathbb{F}_{256}$ and $\mathbb{F}_{384}$ [34] and also the increase in verification time of Karatsuba multiplication as field size increases.

3.3 Verifying an RSA Implementation

RSA also has a similar pyramidal structure like ECC but with just two levels. The base comprises of multi-precision operations just as in ECC, albeit with much larger numbers. The upper level comprises of modular exponentiation, which is used for encryption and decryption. Hence it is clear that the verification is done using the assume-guaranteed hierarchical verification technique, where all the finite field operations are verified and found to be correct.

4 Program Vulnerability Detection

Even though programs may be verified to be correct, minor bugs in the code can be exploited to subvert the implementation and execute malicious payloads. To detect such bugs, which we call programming vulnerabilities, we need to analyze the implementation. Formal verification tools such as CBMC [18] can be used for this purpose.

The first step is to classify bugs according to vulnerability numbers present in the CVE database [26]. This database lists publicly known vulnerabilities and exposures. The CWE database classifies lists according to various known weaknesses [27]. The sub-category CWE-310, View-658 lists the various weaknesses of software written in the C programming language. Another sub-category View-702, lists potential weaknesses introduced during implementation. To detect program vulnerabilities, we assume that these databases would help identify a majority of the vulnerabilities in the implementation. We begin this section by highlighting the various programming vulnerabilities, before discussing the detection details. The CVE database [26] reports 8690 buffer overflow (including arithmetic overflow) vulnerabilities. This can be mapped to 82 CWE weaknesses [27], of which we found that 42 are applicable to a multi-precision library. The model-checking tool CBMC [18], can verify the array bounds, pointer safety, and integer arithmetic etc. This covers 83% of the classified bugs in the databases.

Buffer Overflow. Buffer overflow is a serious security concern, which has been studied for several years. A buffer is a continuous chunk of memory, such as an array, and is associated with a pointer. Buffer overflows occur when the pointer is accessed beyond the bounds. An attacker may use buffer overflows to write into an illegal memory that could then be used to subvert execution. Programming languages such as *C* and *C++* are prone to buffer overflows, where there are no built-in bound check conditions. These overflows, such as in the example given below, are typically not detected at compile time.

```
int buffer[10];
int i = 10;
buffer [i+1] = 20;
```

Buffer overflows have different variants such as – (1) Heap overflows: where the overflow occur in dynamically allocated memory; (2) Stack overflows: exploits overflow the stack based buffer, which can change the local variables in the program, return address, and even function pointers.

Consider the buffer overflow example above, CBMC will detect the buffer overflow in the program by inserting proper bound check conditions for each array access. The output from CBMC tool is as follows:

Result from CBMC

```
State <S> file <name.c> line <L> function main
----------------------------------------------
  i=10 (00000000000000000000000000001010)

Violated property:
  file <name.c> line <L> function main
  array ‘buffer’ upper bound
  (signed long int)(1 + i) < 10l

VERIFICATION FAILED
```

CBMC performs buffer overflow checks by inserting conditions in the source code that would validate a pointer access is legal. It would then check whether the conditions are reachable or not. As an example, conditions such as

$$(\texttt{i} >= 0) \;\; \texttt{and} \;\; \neg(\texttt{i} >= \texttt{MAXDIGITS} - 1)$$

These conditions automatically determine if an array of size `MAXDIGITS`, accessed with index `i`, violates the lower or upper bound properties (i.e., whenever `a[i]` occurs in the program).

Integer Overflow. An integer overflow could occur in programs that deal with arithmetic operations, where a computation result exceeds the range of the representation. Integer overflows can also occur due to improper type conversion, which can affect the security of the program and lead to unintended behavior of the program.

```
int i = -2147483648;
i = i - 1;
```

The example given above is an integer overflow, where the value of `i` has the lowest value in its representation and the subtraction changes its value to `2147483647`.

To detect potential integer overflows, CBMC automatically inserts specific properties for each variable as given below.

$$(\texttt{i} < \texttt{INT\_MIN}) \,||\, (\texttt{i} > \texttt{INT\_MAX})$$

This determines whether the defined variable `i` with data-type (in our example `int`) violates the underflow or overflow property. CBMC determine whether the property is reachable for each access of these variables. For the integer overflow program given above, the verification result returned by CBMC is as follows :

Result from CBMC

```
State <S> file <name.c> line <L> function main
----------------------------------------------
i=-2147483648 (10000000000000000000000000000000)
```

```
Violated property:
  file <name.c> line <L> function main
  arithmetic overflow on signed -
  !overflow("-", signed int, i, 1)

VERIFICATION FAILED
```

4.1 Program Vulnerability Detection in a Multi-precision Library

Direct verification for buffer overflow and integer overflow need not be effective for a crypto implementation due to its large mathematical model. For detecting all possible vulnerabilities, we have used a hierarchical verification. The method verify all the lower level implementation for vulnerabilities. These verified implementation can be used for verification of higher level implementations. Hierarchical verification will help to determine the flow of these vulnerabilities in the program and also help to speed up the verification.

Table 2 shows the verification time for vulnerability detection in a multi-precision library.

5 Formal Verification of Side Channel Countermeasures

IoT devices that are physically accessible to an attacker are vulnerable to side-channel attacks. These attacks detect sensitive information flowing through unintended covert channels such as the device's power consumption. For example, if the device performs an operation, E, such as $y \leftarrow E(x, k)$, with a secret key k and plaintext bits x, then information about k is leaked through the device's power consumption.

Masking is a popular countermeasure used to prevent this leakage [11]. With this countermeasure, sensitive variables like k are masked so that there is no leakage through the power traces. The mask is then removed at the end of the

Table 2: Program vulnerability detection in a multi-precision library

Function	**Description**	**Time(in seconds)**	**Number of Vulnerabilities Detected**
Copy	$x \leftarrow y$	0.002	1
Addition	$r = x + y$	0.333	3
Subtraction	$r = x - y$	0.398	3
Compare	$x == y$	0.628	0
Karatsuba multiplication	$r = x \times y$	53.46	16
Left-shift	$r = x \gg 1$	0.004	0
Right-shift	$r = x \ll n$	0.006	2

operation. For example, if E is a linear function, then for a randomly chosen value of mask, r, the operation $E(x, k \oplus r)$ is done instead. No leakage about k is present in the power consumption if r is secret. Moreover, the correct result y is obtained by computing $E(x, k \oplus r) \oplus E(x, r)$. This works because E is a linear function and $E(x, k \oplus r) = E(x, k) \oplus E(x, r)$.

Masking non-linear operations is not so trivial. It is time consuming and error-prone as information can leak through intermediate operations. For example, consider the non-linear operation ($o = x \wedge k \wedge r$). A value of $o = 1$, which can be distinguished in the power consumption, will leak the value of k, which would also be 1 in this case. Formal verification has been used to verify perfect masking. The property used in formal verification is that every intermediate operation I that is used in the computation of the non-linear function E, should be perfectly masked [20]. We assume that every intermediate operation I is Boolean and has parameters x bits of the plaintext, k secret key, and random mask r. The property fed to the model checker to verify perfect masking is the following:

$$\exists x \, \exists k \, \exists k' \, (\sum_r I(x, k, r) \neq \sum_r E(x, k', r)) \ .$$

The property verifies that for any input x and a pair of keys k and k' ($k \neq k'$), the probability distribution of $I(x, k, r)$ differs from that of $I(x, k', r)$. If this holds then some information about the secret key k is leaked through the side-channel. In this case, the intermediate operation I is not perfectly masked. If the model checker finds that the above property is not satisfied, it means that no information is leaked. Hence the intermediate operation I is perfectly masked.

Besides power consumption based channels, execution time can also lead to information leakage. Developing constant time implementations is a difficult task in modern processor environments because the execution time not only depends on the implementation but also depends on micro-architectural components such as cache memories, branch prediction, multi-threading, etc. We could have secret independent branching to avoid the timing attack based on the secret data. The secret dependency, the implementation should avoid (a) conditional branching should not depend on secret data (b) indirect load using the secret data [33].

6 Detecting Hardware Trojans using Formal Verification

To minimize development costs and time-to-market, most system developers operate in a fabless mode, integrating multiple third party Intellectual Property cores into System on Chips (SoC) that are used in IoT devices. This design flow creates multiple opportunities for malicious code or circuits to be introduced into the device that could act as Trojans. For example, Trojans in connected IoT devices could act as backdoors and permit unauthorized users to enter into the device and access privileged information. This can lead to denial of service attacks, manipulation of data, or interception of sensitive data.

A typical Trojan is designed to be passive most of the time and active only when triggered. In a normal setting, the trigger is an extremely rare event, for

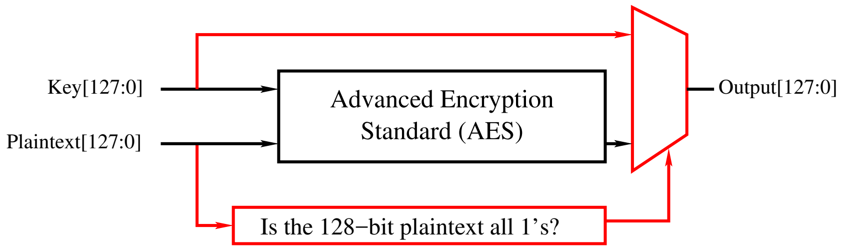
Key[127:0]
Plaintext[127:0]
Advanced Encryption Standard (AES)
Output[127:0]
Is the 128−bit plaintext all 1's?

Fig. 5: The Trojan is triggered when the plaintext (128-bit) is all ones, which triggers the multiplexer and returns the secret key as Ciphertext[30].

example, a specific time event or particular inputs. Due to this stealthy nature, detecting the presence of Trojans is a considerable challenge especially when they are introduced in the hardware. Even 100% test coverage, will not guarantee the absence of Trojans in a hardware design. An attacker, however, with knowledge of the trigger conditions, could easily activate the Trojan to compromise the device.

In the literature several algorithms have been suggested to detect hardware Trojans. One of the most popular is FANCI, which identifies stealthy signals in a design [35]. Another tool called VeriTrust, uses the fact that gates triggered by Trojans, will not be driven by functional inputs. VeriTrust marks these gates as suspicious [36]. While these approaches can detect a Trojan with certain level of accuracy, they will not be able to guarantee the absence of a Trojan in the design. Moreover, they can only evaluate combinational parts of the design and would require considerable manual analysis.

In [30], Rajendran et al. proposed to use formal verification to detect Trojans in hardware designs. If a Trojan is present, formal verification can guarantee finding its trigger condition. Rajendran et al. use model checking for the purpose of detecting hardware Trojans that leak sensitive information from the device. The input to the model checker is the target property to be checked along with a formal description of the design in temporal logic. The property to be checked is: "does the design leak sensitive information?". If a Trojan is present in the design, the output of the model checker will be a set of states denoting the trigger condition. As a case study, [30] considered a Trojan that could leak an encryption key as shown in Figure 5. In normal working conditions, the design would encrypt data with the stored key. However, when a trigger is fed via the plaintext input, the output of the circuit would be the AES encryption key rather than the ciphertext, thereby leaking the key. In the simplest case, a specific value of plaintext, for example all 1s, could act as trigger resulting in leakage of the entire secret key or a subset of the secret key. This can be modeled formally using the property:

$$\exists i \in I \ni D \models (s == o) \ .$$

It means that there exists some trigger i from the set of possible input patterns I such that in the design D, the secret s is mapped to the output o. The model checker would search for an input assignment over the entire input space I that would satisfy the key leakage property, which is $(s == o)$. If such an assignment can be found then the corresponding input becomes the trigger for the Trojan.

The limitation of this approach is that the model checking property should comprehensively capture all trigger and leakage conditions if the Trojan is to be detected. Rajendran *et al.* [30] discusses multiple other options for creating the trigger and leaking the secret key. For example, the trigger could arrive over multiple clock cycles. The secret key leakage could be only a few bits of the entire key. Alternatively, the leakage could be a function of the key rather than the actual key bits.

Another limitation is in the scalability of formal verification to detect well concealed hardware Trojans in larger circuits. The results of Rajendran *et al.* were limited to preforming bounded model checking on 12 clock cycles of the designs. It would take considerably longer to verify if the Trojan's trigger occurred much later clock cycles. The scalability of formal verification to detect better concealed Trojans in larger designs is still an open problem.

7 Leveraging Formal Verification to Identify Meta-level Authentication Loopholes

Designers are often more keen to adopt security counter-measures over the cryptographic implementations and apply formal certification procedures to eliminate possible security flaws in the system design. However, there can be functional gaps in the meta-level of the implementation which may lead to weaknesses. Such gaps are often manifested, as designers (while implementing) are unaware of the IoT environment where these devices will be deployed in future. Such meta-level gaps can be categorized as follows:

- `[Level-1]` *Absence of Authentication.* The access points of a secure implementation may require the use of authentication, may be in the form of passwords, which is completely oblivious to the designer. Hence, (s)he has not performed/devised any password or authentication checks in the high-level invocation of this implementation which may led to exploitation in the implementation and extract out secure data.
- `[Level-2]` *Inability to Provide Strong Authentication.* Though the designer has implemented an authentication mechanism to safeguard the access of a secure implementation, however there may be shortcomings in the formulations where the authentication mechanism may be simple enough (for example, in case of simple passwords or reduced set of variations in the key space) that it may get regenerated by exhaustive enumeration of the possible variations of the encryption within computational limits.
- `[Level-3]` *Missing Checks over Authentication Process.* Even if a strong authentication wrapper is present around the invocation of a secure implementation, there may be loopholes in the usage of the authentication process

which may lead to serious flaws in the design. A strong authentication may get destroyed when multiple attackers can log through the same authentication strategy and distribute the search space to derive the encryption strategy/key. There is a possibility that this approach may lead to breakthrough in the authentication barrier which may have been computationally infeasible by single or two simultaneous user. Therefore, placing a limit on the number of accesses (for example, there may be checks to prohibit more than two login simultaneously) may resolve these issues.

The above discussion points to the fact that it may not be sufficient to formally certify the secure design/implementation, but we need to formally model the environment where this implementation is being deployed/invoked from. Then, another round of formal certification is mandatory to ensure the flaws in the meta-level authentication over the design. To adopt the formal certification in this level, we perform the following strategy:

– First, we abstract the functionality of the secure implementation in the form of *assumptions* (assume properties).
– Then, we formally model the authentication wrapper which invokes the secure design implementation in the form of assume properties.
– Next, the mentioned three-level attributes are captured in terms of a set of formal specifications.
– Finally, formal verification is performed over the authentication model (which instantiates the design in the form of assume properties) with respect to the formal specifications formed.

8 Conclusions

Recent developments in formal verification have significantly extended the capabilities of these tools. Theorem proving and model checking can potentially be applied to solve several hard problems in security, especially in the IoT domain. This chapter provided an overview of formal verification applied to solve five critical security issues related to an IoT device. Formal verification scales very well for some of the problems considered, such as detecting programming bugs in software and proving side-channel security.

Innovative usage of the formal verification tools is required to solve certain problems involving huge state space. This chapter demonstrated the use of a hierarchical verification methodology for verifying the correctness of cryptographic implementations, which has considerably huge state space making a naïve invocation of the formal verification tool fail. Identifying certain security problems, such as detection of hardware Trojans, though feasible with formal verification, is very restricted. The state-of-the-art can for instance, only detect Trojans that are triggered in the first few clock cycles of the device operation. If the Trojan is well concealed, for example, gets triggered much later in the device operation, then identifying them would be considerably more difficult.

References

1. Affeldt, R.: On construction of a library of formally verified low-level arithmetic functions. In: Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012. pp. 1326–1331 (2012)
2. Amla, N., Du, X., Kuehlmann, A., Kurshan, R.P., McMillan, K.L.: An Analysis of SAT-Based Model Checking Techniques in an Industrial Environment. In: Proceedings of International Conference on Correct Hardware Design and Verification Methods (CHARME). pp. 254–268 (2005)
3. Amla, N., Kurshan, R.P., McMillan, K.L., Medel, R.: Experimental Analysis of Different Techniques for Bounded Model Checking. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 34–48 (2003)
4. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Trans. Program. Lang. Syst. 37(2), 7:1–7:31 (2015), http://doi.acm.org/10.1145/2701415
5. Biere, A., Cimatti, A., Clarke, E.M., Fujita, M., Zhu, Y.: Symbolic Model Checking Using SAT Procedures Instead of BDDs. In: Proceedings of 36th Annual Design Automation Conference. pp. 317–320 (1999)
6. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. Lecture Notes in Computer Science 1579, 193–207 (1999)
7. Biere, A., Clarke, E.M., Raimi, R., Zhu, Y.: Verifying Safety Properties of a PowerPC Microprocessor Using Symbolic Model Checking without BDDs. In: Proceedings of International Conference on Computer-Aided Verification (CAV). pp. 61–71 (1999)
8. Bryant, R.: Graph-based Algorithms for Boolean-function Manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
9. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L.: Sequential Circuit Verification Using Symbolic Model Checking. In: Proceedings of 28th Annual Design Automation Conference. pp. 46–51 (1991)
10. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic Model Checking: 10^{20} States and Beyond. Information and Computation 98(2), 142–170 (1986)
11. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Wiener, M.J. (ed.) Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1666, pp. 398–412. Springer (1999), https://doi.org/10.1007/3-540-48405-1
12. Chen, Y., Hsu, C., Lin, H., Schwabe, P., Tsai, M., Wang, B., Yang, B., Yang, S.: Verifying curve25519 software. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014. pp. 299–309 (2014)
13. Clake, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded Model Checking Using Satisfiability Solving. The Journal of Formal Methods in System Design 19(1), 7–34 (2001)
14. Clarke, E.M., Grumberg, O., Hamaguchi, K.: Another Look at LTL Model Checking. In: Proceedings of International Conference on Computer-Aided Verification (CAV). pp. 47–71 (1994)
15. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)
16. Clarke, E., Kroening, D.: The CPROVER User Manual (2006)

17. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001)
18. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. pp. 168–176 (2004)
19. Duan, J., Hurd, J., Li, G., Owens, S., Slind, K., Zhang, J.: Functional correctness proofs of encryption algorithms. In: Logic for Programming, Artificial Intelligence, and Reasoning, 12th International Conference, LPAR 2005, Montego Bay, Jamaica, December 2-6, 2005, Proceedings. pp. 519–533 (2005)
20. Eldib, H., Wang, C., Schaumont, P.: Formal verification of software countermeasures against side-channel attacks. ACM Trans. Softw. Eng. Methodol. 24(2), 11:1–11:24 (2014), http://doi.acm.org/10.1145/2685616
21. Goldberg, E., Novikov, Y.: BerkMin: A Fast and Robust SAT-Solver. In: Proceedings of Design Automation and Test Conference in Europe Conference (DATE). pp. 142–149 (2002)
22. Kang, H.J., Park, I.C.: SAT-based Unbounded Model Checking. In: Proceedings of 40th Annual Design Automation Conference. pp. 840–843 (2003)
23. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Texts in Theoretical Computer Science. An EATCS Series, Springer (2008), https://doi.org/10.1007/978-3-540-74105-3
24. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
25. McMillan, K.L.: Applying SAT Methods in Unbounded Symbolic Model Checking. In: Proceedings of International Conference on Computer-Aided Verification (CAV). pp. 250–264 (2002)
26. The MITRE Corporation: Common Vulnerabilities and Exposures, https://cwe.mitre.org/
27. The MITRE Corporation: Common Weakness and Enumerations, https://cwe.mitre.org/
28. Moskewicz, M., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Effcient SAT Solver. In: Proceedings of 38th Annual Design Automation Conference. pp. 530–535 (2001)
29. Nguyen, D.M., Stoffel, D., Welder, M., Kunz, W.: Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In: Proceedings of International Conference on Computer-Aided Design (ICCAD). pp. 442–449 (2002)
30. Rajendran, J., Dhandayuthapany, A.M., Vedula, V., Karri, R.: Formal security verification of third party intellectual property cores for information leakage. In: 29th International Conference on VLSI Design and 15th International Conference on Embedded Systems, VLSID 2016, Kolkata, India, January 4-8, 2016. pp. 547–552. IEEE Computer Society (2016), https://doi.org/10.1109/VLSID.2016.143
31. Silva, M., Sakallah, K.A.: GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Transactions on Computing 48(5), 506–521 (1999)
32. Smith, E.W., Dill, D.L.: Automatic formal verification of block cipher implementations. In: Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008. pp. 1–7 (2008)
33. Tsai, M., Wang, B., Yang, B.: Certified verification of algebraic properties on low-level mathematical constructs in cryptographic programs. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017. pp. 1973–1987 (2017)
34. U.S. Department of Commerce, National Institute of Standards and Technology: Digital signature standard (DSS) (2000)

35. Waksman, A., Suozzo, M., Sethumadhavan, S.: FANCI: identification of stealthy malicious logic using boolean functional analysis. In: Sadeghi, A., Gligor, V.D., Yung, M. (eds.) 2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013. pp. 697–708. ACM (2013), http://doi.acm.org/10.1145/2508859.2516654
36. Zhang, J., Yuan, F., Wei, L., Liu, Y., Xu, Q.: VeriTrust: Verification for Hardware Trust. IEEE Trans. on CAD of Integrated Circuits and Systems 34(7), 1148–1161 (2015), https://doi.org/10.1109/TCAD.2015.2422836