2018 9th ACM/IEEE International Conference on Cyber-Physical Systems

SAT-based Synthesis of Spoofing Attacks in Cyber-Physical Control Systems

Omar Inverso
Gran Sasso Science Institute L'Aquila
omar.inverso@gssi.it

Alberto Bemporad Mirco Tribastone
IMT School for Advanced Studies Lucca
{alberto.bemporad,mirco.tribastone}@imtlucca.it

***Abstract*—We propose an approach to either certify that a given control system is safe under possible cyber-attacks on the measured data used for feedback and/or the commanded control signals, or alternatively synthesise a particular spoofing attack that corrupts the signals to make the closed-loop system unsafe. We assume that a (possibly nonlinear) dynamical model of the physical plant is available along with the control law, but that no on-line diagnosis is in place to detect attacks. After converting the model to a piecewise polynomial discrete-time form, we interpret the synthesis of the spoofing attack as a software verification query by means of an encoding into a Boolean satisfiability problem. Using a prototype implementation of our verification engine, we demonstrate its effectiveness on a case study of cyber-attack to a chemical reactor.**

I. Introduction

There is a clear technological trend in increasing the interconnection between physical devices and digital units, programmed to automatically change the physical behavior of the device for new or improved functionalities. Examples range from autonomous driving to smart grids, water networks, passenger aircrafts, nuclear and chemical plants, and many others. The new technologies come at the price of making such *cyber-physical systems* (CPS) more fragile to malicious attacks from adversaries that want to extract information, cause malfunctioning, or otherwise interfere with the expected operating conditions of a system to intentionally cause damages, and ultimately threaten human lives in many cases. For this reason, *security* of CPS has attracted increasing attention in recent years. Assessing that a given CPS is safe can be rather challenging, because of the inherently intertwined nature of the digital and physical components. For example, security weaknesses may be found in the communication infrastructure (e.g., in the underlying network that connects sensors, the physical systems, and control units) or in the digital components, e.g., from software vulnerabilities or by exploiting lack of protection mechanisms (as in communication protocols in cars, e.g., [1]) because of the computational overheads they typically introduce.

In this paper we focus on CPS security from the point of view of sensor/actuator spoofing in an automatic control system. That is, we deal with compromising sensor measurements that are fed back to the controller, and/or the input signals commanded by the controller, in order to steer the controlled physical system toward an unwanted or unexpected condition that is arbitrarily defined by the attacker (e.g., when tampering with an anti-lock braking system [2]). We take a model-based white-box approach to this problem, assuming that a model of the plant is available together with the control law for security analysis. We devise an automatic framework that takes the closed-loop model, the set of corruptible signals of the controller, and a target condition as inputs. The output is an *attack sequence*, i.e., the actual sequence of corrupted signals that brings the system to the target, or an answer that no sequence exists that can reach the target, for a given precision chosen to describe the model of the system and the resolution of the attack signal.

In practice, this can be a relevant scenario for a CPS stakeholder to understand whether a certain condition (such as a given dangerous critical temperature) can be reached by only appropriately tampering with the measurements acquired by the controller, and/or by altering the control signals transmitted on the bus to the physical actuators. Of course, our algorithm can be dually used in principle by an attacker, although this would require to know (or estimate) a model of the system and the control law.

Our approach leverages ideas and techniques from software verification. The model of the controlled system is represented as a program consisting of a bounded cycle where each iteration computes the next state of the system. Each variable that represents a corruptible input to the closed-loop system, at each time step, is declared as nondeterministic; the target condition, instead, is represented as a logical assertion. With this set-up, the problem of finding an attack sequence is translated into the satisfiability problem which looks for an assignment of the nondeterministic variables such that the given assertion holds true.

The specific encoding of the satisfiability problem depends on the semantics used for the program, particularly for the representation of numbers. Here we consider a fixed-point representation of real numbers, also motivated by the fact that nowadays embedded systems still run in fixed-point rather than floating-point arithmetics, as it is computationally less expensive and therefore requires cheaper hardware. By analyzing the program with fixed-point semantics we are able to capture the typically limited dynamic range of many real (micro-)controllers in a natural way. Further, in our framework we allow arithmetic operations between numbers with different precision: this allows us to use larger precisions for the variables that represent the dynamics of the system for

0-7695-6378-3/18/$31.00 ©2018 IEEE
DOI 10.1109/ICCPS.2018.00009

maintaining modeling accuracy.

The synthesis problem can be solved by encoding the program as a propositional logic formula, using established approaches in software verification, which is then fed to a Boolean satisfiability (SAT) solver. If the problem is satisfiable we obtain a candidate attack sequence; otherwise, unsatisfiability means that no such sequence exists, giving a model-based certificate of robustness to spoofing attacks.

II. Related Work

The topic dealt with in this paper is somewhat dual to that on attack detection, whose goal it to infer which states and outputs are being compromised. White-box approaches assume existing knowledge of the system and model the attack as a disturbance entering the model equations, mostly linear dynamical models [3], [4]. In [5], a black-box approach is presented which learns the model from observations of the system in nominal conditions.

Research on secure state estimation considers the problem of detecting an attack on some sensors and reconstructing the correct state by using the uncorrupted sensor measurements [6], [7], [8], [9], [10], [11]. Among such techniques, [9], [10], [11] are similar in that they use program-analysis approaches based on satisfiability modulo theories (SMTs) [12].

More closely related to our approach is [13], which is also concerned with the synthesis of sensor spoofing attacks. However, it focuses on a specific scenario, namely embedded systems running on the MSP430 family of micro-controllers, with attack sequences that are generated via the framework of symbolic execution [14].

Another rigorous way to study sensor spoofing is to interpret the attack sequence as a disturbance and model the closed-loop system as an uncertain system, computing the reach set (or an under-/over-approximation thereof) which can be interpreted as a “safety envelope” (cf. [5]). The reach set can be used to decide whether an attack sequence exists, however it does not synthesise one. Finally, as in the approach in [15], [16] based on piecewise linear hybrid dynamical models, we mention that for models described by nonlinear dynamics the synthesis of an attack sequence could be posed as a nonlinear optimization problem subject to equality (dynamic) constraints and inequality (reachability) constraints. While any feasible (even suboptimal) solution would be a valid attack sequence, proving safety could be a rather difficult task, as one should prove that no feasible solution exists to a (possibly highly nonconvex) mathematical programming problem.

SMT-based reasoning has been used to verify fixed-point implementations and properties (such as stability) of digital filters and controllers [17], [18]. These techniques, however, only work on specific classes of fixed-point programs.

Theorem proving has been applied to automatically verify the correctness of translations from floating-point to fixed-point programs [19] and in particular digital filters [20].

The idea of reducing the analysis of closed-loop control systems to software verification by translating them into programs is not new [21]. Our approach is different in that it can also handle polynomial systems. Both our approach and the cited one evaluate the robustness of the system up to a given time horizon. However, we concentrate on specifically checking whether it is possible to synthesise an attack vector to steer the system to an undesired condition, rather than by analysing the propagation of uncertainty with time. We use bounded model-checking rather than concolic execution for the analysis of the translated program.

```
nondeterministic a[k], k = 1,...,K

for k = 1 : K
  y[k] = g(x[k], a[k])
  u[k] = h(y[k])
  x[k+1] = f(x[k], u[k])
end

assert target_condition
```

Listing 1: Synthesis problem as program analysis.

III. Problem Statement

We consider a closed-loop control system described by the following discrete-time state-space model

$$\begin{aligned} x(k+1) &= f\big(x(k), u(k)\big) \\ y(k) &= g\big(x(k), a(k)\big) \\ u(k) &= h\big(x(k)\big) \end{aligned} \tag{1}$$

where $x(k) \in \mathbb{R}^n$ includes the states of the process and the controller, $y(k) \in \mathbb{R}^m$ is the measured output that is affected by the attack vector $a(k) \in \mathbb{R}^d$, and $u(k) \in \mathbb{R}^p$ is the command input.

We assume that functions f, g, and h are piecewise multivariate polynomials over the variables $x(k)$, $u(k)$, and $a(k)$. Many physical systems can be well captured by (1) under such an assumption, which is therefore not very restrictive, as we will exemplify in Section VI.

We assume that the attack target is a condition expressed in quantifier-free first-order logic over the variables $x(k)$, $u(k)$ and $a(k)$, over a fixed time horizon K. Then, the synthesis problem consists in finding an assignment of $a(0)$, $a(1)$, ..., $a(K-1)$, such that the attack-target condition is satisfied.

We cast this problem into a software verification one. The corresponding program to be verified, in pseudocode, is given in Listing 1. The attack sequence is declared as nondeterministic, i.e., the values are not specified in the program. Hence, the program is not executable, but it can be analysed by a satisfiability solver, which looks for an assignment of the nondeterministic variables such that the target condition declared in the assertion holds true.

In the next sections we detail how such software verification can be realised. In particular, we consider the case where the program variables are represented as fixed-point numbers, leading to an encoding into a Boolean satisfiability problem. In particular, we first introduce fixed-point arithmetics (Section IV). Then we describe a modular approach to encode the semantics of fixed-point arithmetics as an intermediate program representation based on standard integer arithmetics over arbitrary-sized vectors of bits—*bitvectors* for short (Section V).

IV. Fixed-point Arithmetics

Fixed-point arithmetics approximates computations over the rational numbers via standard integer arithmetics. In fixed-point notation, numbers are expressed as constant-length sequences of binary digits where a radix point at a specific position separates the integer part of the representation from the fractional one. We say that the *precision* or *format* of a given fixed-point variable is $p.q$ when its integer and fractional parts are represented using p and q binary digits, respectively. We denote such variable with $\mathtt{x}_{(p.q)} = \langle a_{p-1}, \ldots, a_0.a_{-1}, \ldots, a_{-q}\rangle$, omitting the subscript when irrelevant or clear from the context. Since the format, or the location of the radix point, is not part of the representation, the storage size for a fixed-point variable is $p + q$, plus one extra bit to hold the sign in case of signed arithmetics. In this case, usually two's complement format with sign extension for signed values is used. The *resolution* of a fixed-point format is the magnitude of the smallest representable non-zero value, while the *range* is the representation interval. For example a format $p.q$ has resolution is 2^{-q}, and ranges $[0, 2^p - 2^{-q}]$ and $[-(2^p), 2^p - 2^{-q}]$, respectively for unsigned and signed representations, assuming that the sign bit is not counted.

Unlike floating-point arithmetics, fixed-point arithmetics does not currently follow any specific standard and thus its support is limited to vendor-specific solutions. A specific issue regarding fixed-point arithmetics is the separate bookkeeping required to keep track of the radix point, as this is not explicitly represented. The ISO/IEC TR 18037 [22] proposes language extensions for the C programming language to support the fixed-point data type, which vendors are expected to follow, and that have already been implemented in the GNU compiler.

In practice, fixed-point numbers are manipulated much like regular integers. The exact format is chosen depending on the application, and it is generally worked out by considering the required range and resolution. A sum operation under fixed-point arithmetics is performed as in regular integer arithmetics and takes one extra bit to hold the result, e.g., $\mathtt{z}_{(p+1,q)} = \mathtt{x}_{(p.q)} + \mathtt{y}_{(p.q)}$ as long as the two operands are in the same format. If the formats are different from each other, then shift operations, truncation, or format extension of one or both operands need to be carried out upfront to obtain the same format. Further details are discussed in the next paragraph. The multiplication of two fixed-point variables is also performed as in integer arithmetics. In this case the two operands are not required to be in the same format. The format to store the result uses the sum of the integer and fractional precision of the two operands for the integer and fractional part of the representation respectively. For instance $\mathtt{z}_{(2p.2q)} = \mathtt{x}_{(p.q)} \cdot \mathtt{y}_{(p.q)}$, assuming the same precision for the two operands. Conversely, a division operation between operands of the same precision requires left shifting the first operand to extend its representation before the actual integer division can take place, $\mathtt{z}_{(p.q)} = \mathtt{x}_{(2p.2q)}/\mathtt{y}_{(p.q)}$.

```
fixedpoint x(3.2), y(3.2), z(3.2), z(4.2);

x(3.2) = 7.5_10;                 // +111.10, 011110
y(3.2) = 0.5_10;                 // +000.10, 000010
z(4.2) = x(3.2) + y(3.2); // +8.0, +1000.00, 0100000
z(3.2) = x(3.2) + y(3.2);
```

Listing 2: Overflow in fixed-point arithmetics.

```
fixedpoint x(3.2), y(3.2), z(3.2), z(3.3);

x(3.2) = 0.25_10;                // +000.01, 000001
y(3.2) = 0.5_10;                 // +000.10, 000010
z(3.3) = x(3.2) · y(3.2); // +0.25, +000.001, 0000001
z(3.2) = x(3.2) · y(3.2);
```

Listing 3: Numerical error in fixed-point arithmetics. We use decimal constants for initialising the variables; the equivalent fixed-point binary, and two's complement bitvector representations are shown in the comments.

The operations above introduce format restrictions on the variable that holds the result or on the involved operands. Therefore, it may be necessary to convert variables to smaller or greater formats. We refer to any of these operations as a *precision conversion*. Converting a variable to a smaller integer format is done by either saturating or simply trimming down the representation starting from the most significant digit. In this case *overflow* may occur. An example of such a situation is the code snippet from Listing 2, where variable $\mathtt{z}_{(3,2)}$ uses an insufficient number of bits to store the integer part of the result of the sum between $\mathtt{x}_{(3,2)}$ and $\mathtt{y}_{(3,2)}$. Reducing the fractional precision is done by either rounding or truncating, and may cause *numerical error*. An example is shown in Listing 3, where variable $\mathtt{z}_{(3,2)}$ uses an insufficient number of bits to store the fractional part of the result of the multiplication between $\mathtt{x}_{(3,2)}$ and $\mathtt{y}_{(3,2)}$, and thus the least significant digit of the result (correctly stored in $\mathtt{z}_{(3,3)}$) is dropped. A similar problem may occur in the presence of right shift operations. Extending the integer part of the representation requires sign extension in case of signed arithmetics. Extending the fractional part simply requires left shifting to zero-pad the required amount of least significant positions. Other operations such as square root can be rewritten by combining the operations described above. In the rest of the paper we focus on non-saturated signed fixed-point arithmetics over mixed-precision variables.

V. Verification Approach

We consider programs with arithmetic expressions over fixed-point variables, or *fixed-point programs* for short. These are C-like programs with scalars, arrays and loops, plus an extra data type for fixed-point variables, and with verification-oriented primitives for symbolic (i.e., nondeterministic) initialisation and assertion checking to express the requirements

```
1   fixedpoint x(3.2), y(1.2), z(3.2), z(3.3);
2
3   main() {
4       x(3.2) = 0.5₁₀;
5       y(1.2) = nondet();
6
7       z(3.3) = x(3.2) · y(1.2);          // first
8       z(3.2) = x(3.2) · y(1.2);          // second
9
10      assert(z(3.3) == z(3.2));
11  }
```

Listing 4: A fixed-point program.

of interest (i.e., the attack condition). Fixed-point variables can have arbitrary formats, encoded as part of the corresponding variable identifier. Arithmetic operations of fixed-point operands of mixed size are allowed.

A fixed-point program is shown in Listing 4. The program multiplies constant $\mathtt{x}_{(3.2)}$ by variable $\mathtt{y}_{(1.2)}$ and stores the result to variables of different formats, $\mathtt{z}_{(3.3)}$ and $\mathtt{z}_{(3.2)}$. The nondeterministic initialisation of variable $\mathtt{y}_{(1.2)}$ models all possible executions of the program for any possible value of the variable allowed by its format. The assertion at the end checks whether the result is consistent across the two considered precisions.

Analysing the program shown in Listing 4 is equivalent to checking whether there exists an assignment for $\mathtt{y}_{(1.2)}$ that leads to a violation of the assertion, that is $\mathtt{z}_{(3.3)} \neq \mathtt{z}_{(3.2)}$. An error trace is $\mathtt{y}_{(1.2)} = 1.25_{10}$, which causes the result of the multiplication to be represented accurately in $\mathtt{z}_{(3.3)} = 0.625_{10}$, but not in $\mathtt{z}_{(3.2)} = 0.5_{10}$, where due to insufficient fractional precision the least significant digit is lost. The program is said to contain a *reachable assertion failure*.

Our program analysis approach consists of two main phases. In the first phase, the original fixed-point program (Listing 4) is transformed into a *bitvector program*, that is essentially a loop-free C-like program that uses standard integer arithmetics over arbitrary-sized vectors of bits (Listing 6). Our transformation guarantees that the bitvector program contains a reachable assertion failure if and only if the fixed-point program contains either a reachable assertion failure or an overflow. For example, Listing 6 contains a reachable assertion failure at line 18 if and only if the multiplication at line 7 of Listing 4 can overflow. In the second phase, we convert the bitvector program into the appropriate format accepted by the verification tool of preference. We then feed the as-converted bitvector program to the backend, and map back any error trace from the backend to refer to the bitvector program, and from there to the initial fixed-point program.

We now describe more in detail how to translate a fixed-point program (Figure 4) into a bitvector program (Figure 6). We assume the following format:

(A1) the fixed-point program is unfolded;
(A2) each line of code contains at most one statement;
(A3) the number of involved operands for these is always exactly two;
(A4) the only arithmetic operations in the program are additions and multiplications.

Assumption (A1) ensures termination and finiteness of the SAT encoding; (A2) and (A3) are without loss of generality since a fixedpoint program can always be transformed in a format that satisfy these assumptions by means of source-to-source transformations. Assumption (A4) is the main reason for restricting to (piece-wise) polynomial dynamics in the closed-loop control system. Other nonlinearities (e.g., square root) can be covered by exploiting the fact that they can be expressed as sums and multiplications in fixed-point arithmetic, or by appropriate variable transformation at the level of the dynamical model of the control system, as done in Section VI.

For each operation on fixed-point variables we introduce, if needed, a fresh variable in a sufficient precision to store the full actual result of the operation without overflow or numerical error (see Section IV). We then introduce a new assignment statement from this variable to the one initially intended for storing the result of the original operation, thereby forcing an implicit format cast (see Section IV). This mechanism is formalised by the following rewrite rules:

$$[\![\mathtt{z}_{(p.q)} = \mathtt{x}_{(p.q)} + \mathtt{y}_{(p.q)}]\!] \;\to\; \begin{array}{l} \mathtt{z'}_{(p+1,q)} = \mathtt{x}_{(p.q)} + \mathtt{y}_{(p.q)} \\ \mathtt{z}_{(p.q)} = \mathtt{z'}_{(p+1.q)} \end{array}$$

$$[\![\mathtt{z}_{(p.q)} = \mathtt{x}_{(p.q)} \cdot \mathtt{y}_{(p.q)}]\!] \;\to\; \begin{array}{l} \mathtt{z'}_{(2p.2q)} = \mathtt{x}_{(p.q)} \cdot \mathtt{y}_{(p.q)} \\ \mathtt{z}_{(p.q)} = \mathtt{z'}_{(2p.2q)} \end{array}$$

where the notation $[\![\cdot]\!]$ denotes the mapping from the argument (an original line of code) into the transformed code. We note that the second formula uses the same precision for the two operands in a multiplication. This is not required, but only assumed for simplicity.

The manipulation described above guarantees that in the output program no overflow or loss of accuracy can happen in any arithmetic operation, and any occurrence of these problems is limited to format casting operations. An example is discussed later in this section, at the end of the description of our encoding.

The next step in the preparation of our bitvector program consists in splitting all format casting operations in two statements for the integer part and fractional part, when needed. More precisely we introduce the following transformation rule:

$$\underset{[p \neq p',\, q \neq q']}{[\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'.q')}]\!]} \;\to\; \begin{array}{l} \mathtt{x''}_{(p.q')} = \mathtt{x'}_{(p'.q')} \\ \mathtt{x}_{(p.q)} = \mathtt{x''}_{(p.q')} \end{array}$$

so that only separate cases of format casting (Section IV) may occur. These cases are handled by the following rewrite rules:

$$\underset{[p < p']}{[\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'.q)}]\!]} \;\to\; \begin{array}{l} \mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'.q)} \\ \mathtt{assert}(\mathtt{x'}_{(p'.q)} == \mathtt{x}_{(p.q)}) \end{array}$$

$$\underset{[p > p']}{[\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'.q)}]\!]} \;\to\; \mathtt{x}_{(p.q)} = \mathtt{x'}_{(p'.q)}$$

$$\underset{[q > q']}{[\![\mathtt{x}_{(p.q)} = \mathtt{x'}_{(p.q')}]\!]} \;\to\; \mathtt{x}_{(p.q)} = \mathtt{x'}_{(p.q')} << (q - q')$$

$$[\![\mathtt{x}_{(p.q)} = \mathtt{x}'_{(p.q')}]\!]_{[q < q']} \quad \rightarrow \quad \mathtt{x}_{(p.q)} = \mathtt{x}'_{(p'.q)} >> (q' - q)$$

Note that the format casting operations considered above can overflow in the first case only, because of the loss of integer precision. We therefore insert an explicit assertion to make sure that no overflow occur. On the other hand, no manipulation is required when converting to a greater integer precision. Converting to greater or smaller fractional precision will instead need shifting left or right an appropriate number of positions, as shown in the last two rewrite rules.

The effect of applying the rewrite rules presented above may be observed by comparing Listing 4 to the transformed program shown in Listing 5. For example, the multiplication at line 7 of Listing 4 corresponds to lines 7-12 in Listing 5. Note that in the transformed program no overflow can happen during the actual multiplication at line 8 because the result is now guaranteed to be stored using sufficient precision. The overflow can instead occur during the format casting at line 11, where the most significant digit of $\mathtt{w2}_{(4.3)}$ is lost during the assignment to $\mathtt{w2}_{(3.3)}$. The assertion at line 12 fails when this happens.

We finalise our encoding by simply transforming all fixed-point variables into bitvectors of appropriate length, as in Listing 6. The bitvector program so obtained is further processed in order for the actual analysis to take place. In particular, we convert it into the format accepted by the verification tool or reasoning engine chosen as a backend. This step may be more or less straightforward, depending on the backend. A possible option would consist in simply replacing bitvectors with standard integer variables of appropriate size. For example, a fixed-point variable $\mathtt{x}_{(7,8)}$ would require a 16-bit bitvector, which can be stored as a `short`. Such an instrumentation adapts any verification tool capable of symbolic analysis of C programs to the case of fixed-point programs.

Implementation: We implemented our verification approach in a prototype. Our tool performs a sequence of source transformations that implement the rewrite rules described in Section V. As verification backend we chose CBMC [23], due to its native support for arbitrary-sized bitvectors and its bit-precise encoding based on propositional satisfiability. CBMC converts the input program (i.e., our encoding from Listing 6) into a propositional formula that is satisfiable if and only if the program contains a reachable assertion failure. It uses the MiniSat [24] SAT solver to check the satisfiability of the formula, automatically generating an error trace from any satisfiable assignment of the propositional variables. Our prototype then converts any error trace from CBMC by automatically mapping the line numbers back to the initial fixed-point program. Indeed, due to loop unfolding and to the program transformations performed in the first phase, a statement in the original program will correspond to one or more statements in the bitvector program. Therefore in the error trace, if any, the line numbers need to be mapped from the bitvector program back to the initial fixed-point program.

```
1   fixedpoint x(3.2), y(1.2), z(3.2), z(3.3);
2
3   main() {
4       x(3.2) = 0.5₁₀;
5       y(1.2) = nondet();
6
7       fixedpoint w1(4.4);                    // first
8       w1(4.4) = x(3.2) · y(1.2);
9       fixedpoint w2(4.3);
10      w2(4.3) = w1(4.4) >> 1;
11      z(3.3) = w2(4.3);
12      assert(z(3.3) == w2(4.3));
13      fixedpoint w3(4.4);                    // second
14      w3(4.4) = x(3.2) · y(1.2);
15      fixedpoint w4(4.2);
16      w4(4.2) = w3(4.4) >> 2;
17      z(3.2) = w4(4.2);
18      assert(z(3.2) == w4(4.2));
19
20      fixedpoint w5(3.3);
21      w5(3.3) = z(3.2) << 1;
22      assert(z(3.3) == w5(3.3));
23  }
```

Listing 5: Transformed fixed-point program.

```
bitvector x(6), y(4), z(6), z(7);

main() {
    x(6) = 0.5₁₀;
    y(4) = nondet();

    bitvector w1(9);                       // first
    w1(9) = x(6) · y(4);
    bitvector w2(8);
    w2(8) = w1(9) >> 1;
    z(7) = w2(8);
    assert(z(7) == w2(8));
    bitvector w3(9);                       // second
    w3(9) = x(6) · y(4);
    bitvector w4(7);
    w4(7) = w3(9) >> 2;
    z(6) = w4(7);
    assert(z(6) == w4(7));

    bitvector w5(7);
    w5(7) = z(6) << 1;
    assert(z(7) == w5(7));
}
```

Listing 6: Bitvector encoding.

VI. Case Study

A. Model

We consider a sensor spoofing attack in process control. The process is represented by a continuous stirred-tank reactor (CSTR) model [25] that considers a first-order irreversible reaction $A \rightarrow B$ that takes place in a vessel. The states of the process are the concentration C_A of reactant A and the reactor temperature T, whose dynamics are described by the nonlinear ordinary differential equations (ODEs)

$$\dot{C}_A = \frac{F}{V}(C_{Af} - C_A) - C_A k_0 e^{\frac{-\Delta E}{RT}} \tag{2a}$$

$$\dot{T} = \frac{F}{V}(T_f - T) + \frac{UA}{\rho C_p V}(T_j - T) - \frac{\Delta H}{\rho C_p} C_A k_0 e^{\frac{-\Delta E}{RT}} \tag{2b}$$

where $\dot{x}$ denotes the derivative of a quantity x with respect to time. The following are given fixed parameters: F is the flow rate, V is the reactor volume, k_0 is a nonthermal factor, ΔE is the activation energy, R is Boltzmann's gas constant, ΔH is the heat of the reaction, U is the heat transfer coefficient, A is the tank area, C_p is the heat capacity, and ρ is the density. The inputs C_{Af} and T_f give the concentration and the temperature of reactant A in the feed stream, respectively, while the control acts on input T_j, which gives the temperature of coolant flow that surrounds the reactor jacket.

Although (2) is a simple two-state model, it is nontrivial to deal with, as it is formulated in continuous time and contains exponential and fractional expressions. This requires converting the model in polynomial form, discretising time, and choosing a proper precision to represent the variables of the model.

We design a state feedback linear quadratic regulator (LQR) with saturation and integral action for ensuring zero tracking errors in steady state. The LQR control law is designed based on (2) augmented by the integral of the tracking error Q

$$\dot{Q} = C_A - C_A^{\text{ref}} \tag{3}$$

between C_A and a given reference concentration C_A^{ref}:

$$\begin{aligned} T_j = \max(&T_{j0} - l_1(C_A - C_A^0) \\ &- l_2(T + T_S - T^0) - l_3 Q, 273.15) \end{aligned} \tag{4}$$

where T_{j0}, C_A^0, T^0 are reference values for the jacket temperature, the concentration of A, and the reactor temperature, respectively, and l_i are given constants. Saturation at 0° C is imposed for the physical realisability of the actuation command.

The problem is to understand whether the reactor can be steered to the overheat condition

$$T \geq 420\ K \tag{5}$$

by altering the temperature measurements that are consumed by the controller (4).

In order to synthesize the attack sequence, we first need to replace the exponential and rational expressions in (2) by polynomial expressions. This is done by introducing the following two auxiliary state variables X and Y

$$X = e^{\frac{-\Delta E}{RT}}, \quad Y = 1/T$$

where X replaces the exponential expression in (2) to make the ODEs polynomial. By applying the chain rule, the time-derivatives of X, Y are also polynomial

$$\dot{X} = \frac{\Delta E}{R} X Y^2 \left[\frac{F}{V}(T_f - T) + \frac{UA}{\rho C_p V}(T_j - T) - \frac{\Delta H}{\rho C_p} C_A k_0 X\right]$$

$$\dot{Y} = Y^2 \left[\frac{F}{V}(T_f - T) + \frac{UA}{\rho C_p V}(T_j - T) - \frac{\Delta H}{\rho C_p} C_A k_0 X\right]$$

TABLE I: Attack sequences for the CSTR model

p	*s*	*Time*	*Attack Sequence*
8.0	0	27.8	⟨ -145, -187, 13, -232, -208, -221, -212, -240, -217 ⟩
8.0	1	18.8	⟨ -131, -232, -237, -240, -105, -151, -212, -157, 0 ⟩
8.0	2	99.5	⟨ -254, -192, -256, 69, -91, -72, -256, 0, 0 ⟩
8.0	3	45.4	⟨ -27, -147, -212, -132, -230, -120, 0, 0, 0 ⟩
8.0	4	3.6	⟨ -159, -243, -238, 159, 63, 0, 0, 0, 0 ⟩
8.0	5	23.1	⟨ -7, -223, -152, -256, 0, 0, 0, 0, 0 ⟩
8.0	6	8.6	⟨ -181, -232, -240, 0, 0, 0, 0, 0, 0 ⟩
8.0	7	129.5	*unsat*
8.0	8	1.7	*unsat*
8.0	9	0.9	*unsat*
7.1	0	32.1	⟨ -25.0, -111.5, -101.5, -109.0, -88.5, -117.5, -84.5, -128.0, -127.0 ⟩
7.0	0	22.9	⟨ -109, -99, -72, -81, -93, -89, -127, -116, -125 ⟩
7.0	1	19.0	⟨ -102, -88, -34, -111, -125, -126, 0, -123, -128 ⟩
7.0	2	38.2	⟨ -124, -122, -106, -128, -128, -117, -103, 0, 0 ⟩
7.0	3	75.0	⟨ 0, -124, -125, -128, -128, -128, -126, 0, 0 ⟩
7.0	4	44.3	*unsat*
7.0	5	55.4	*unsat*
7.0	6	78.4	*unsat*
7.0	7	131.6	*unsat*
6.0	0	72.6	*unsat*

After the entire closed-loop system has been equivalently described by a polynomial ODE model, it is converted to discrete-time by using the first-order forward Euler method with sampling time 0.05 (in hours). Having converted the closed-loop system to a system of polynomial finite differences with saturation, it can be analysed using the encoding presented in Section V.

B. Results

Listing 7 shows the C-pseudocode of the program used for the analysis and the system parameters. We used 64-bit signed fixed-point representation for all the variables excluding the attack vector. We initially reserved the first 2 bytes for the integer part of the representation plus the sign. We then adjusted the integer precision by using our tool in a counterexample-guided loop performing overflow check within a small timeout, increasing the precision in steps of 4 bits when needed. This resulted in a required integer precision of 23 bits. We used the remaining 40 bits for the fractional part. For the nondeterministic variables representing the attack vector, we experimented with different precisions. In particular, we initially fixed the precision to one byte including the sign, and then considered one bit less and one extra bit for the integer part, as well as one extra bit for the fractional part. We also varied the *sparsity*, i.e. the number of zero elements, of the attack sequence. Increasing sparsity reduces the number of nondeterministic bits, but gives less freedom to the attacker. We limited the unfolding to $K = 9$ iterations of the main program's loop. With this set-up, we intended to consider both satisfiable and unsatisfiable problems for the synthesis of the attack sequence.

Table I summarises our experimental results. Here, p and s represent the precision for the attack sequence and its sparsity, respectively; v is the attack sequence found by the solver, or

```
const int K = 9,             // number of steps
          N = 5;                 // system size

/* Model parameters */
fixedpoint ts = 0.05,              // time step
           CT = 420,      // target temperature
           x0[N] =         // initial condition
          {6,3.3551E+2,1.908E-8,2.980E-3,0},
           k0 = 3.493E+7,
           FV = 1.0,
           Ul = 0.3,
           DeltaHl = -11.92,
           DeltaEl = -5.964+03,
           l1 = -1.6589,
           l2 = 8.1989,
           l3 = -39.2251,
           CA0 = 6.0,
           Tj0 = 301.135,
           T0 = 335.5168,
           CAf = 10,
           Tf = 298.15,
           CAref = 9;

/* Model variables, temperatures, a. vector */
fixedpoint x[N], t[K],
           attack[K] = nondet();

/* Auxiliary variables */
fixedpoint spoofed, control, tj, dxT;

/* Main controller loop */
for (int k=0; k<K; k++) {
  // spoofing attack sensor
  spoofed  = x[1] + attack[k];

  // compute control law
  control = Tj0;
  control -= l1*(x[0] - CA0);
  control -= l2*(spoofed - T0);
  control -= l3*x[4];
  tj = max(control,273.15);

  // variable C_A
  x[0] += ts*(FV*(CAf - x[0]) - x[0]*k0*x[2]);
  dxT = FV*(Tf - x[1]) + Ul*(tj - x[1]);
  dxT -= DeltaHl*x[0]*k0*x[2];
  // variable T
  x[1] += ts*dxT;
  // variable X
  x[2] += ts*(-x[2]*DeltaEl*x[3]*x[3]*dxT);
  // variable Y
  x[3] += ts*(-x[3]*x[3]*dxT);
  // variable Q
  x[4] += ts*(x[0] - CAref);

  t[k] = x[1];
}

assert !(t[0] > CT || ... || t[K-1] > CT);
```

Listing 7: Program implementation of the CSTR control system. The parameter expressions appearing the original equations are lumped into a single variable (e.g., the variable FV represents the product $F \cdot V$). Temperature is expressed in Kelvin degrees and time unit is 1 h.

unsat if no such sequence exists within the given precision and sparsity. The analysis runtime is expressed in hours, measured on a dual Xeon E5-2687W workstation clocked at 3.10GHz, for a total of 16 physical cores, with 128GB memory and running 64-bit Linux version 4.9.6.

The results show that using 8-bit integer precision it takes only three non-zero elements to build up a successful attack, or twice as much non-zero elements when lowering the integer precision just one bit. Lowering the precision one more bit yields no feasible attack of the considered kind within the given number of iterations. Regarding the analysis runtimes, we observe large performance fluctuations, typical of SAT-based procedures. Depending on the configuration, it took a minimum of less than four hours to a maximum of slightly more than four days to find satisfiable attacks. Similarly, it took less than one hour to over five days of computations and about 70GB of system memory to terminate the analysis of unsatisfiable instances.

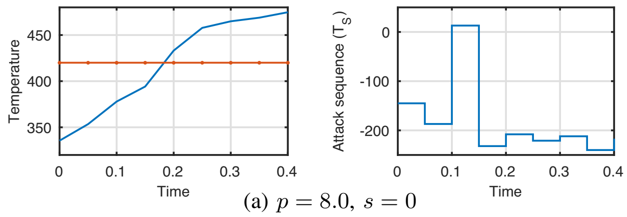
Note that, due to unfolding the main loop, splitting complex operations into multiple simple operations, and introducing intermediate computations to model the semantics of fixed-point arithmetics, the bitvector program ends up having about 20 times the number of lines of code in Listing 7. This in turn leads to propositional formulae of about 2M variables and 10.5M clauses, with negligible variations over the considered precision and sparsity ranges. The roughly constant size of the instances and the large performance gaps discussed above confirm that the expected performance of the solver on a given instance is hardly predictable by relying on simple metrics such as the number of variables and clauses. Nevertheless, the considerable size of the SAT encodings confirms once again the excellent scalability of modern SAT-based reasoning engines.

C. Validation

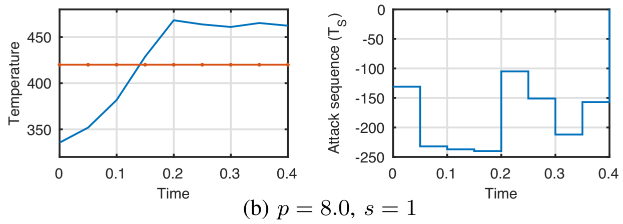
We validated the attack sequences on the original ODE model. We interpolated a continuous piece-wise constant attack signal from each candidate sequence found using the SAT procedure, considering the cases with precision 8.0. The ODE model with the as-constructed signals was then solved using MATLAB's *ode15s* solver. The results, presented in Fig. 1, confirm that the overheat condition is reached in all cases.

VII. Conclusions

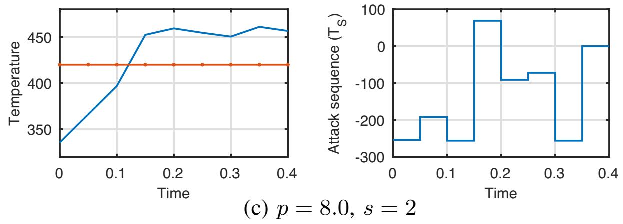
We have studied the problem of synthesising an attack to corrupt the signals in a controlled cyber-physical system that steer it toward a malicious target condition, or proving that such attacks do not exist within the considered precision and time horizon. In this paper we have considered an approach based on Boolean satisfiability by means of an interpretation as a software-verification query. In particular we have investigated a semantics based on fixed-point arithmetics with possibly different precisions, giving freedom to the modeller in tuning the dynamic range of the variables. The experimental results on a nontrivial (nonlinear) model have shown the feasibility of our approach, both in the case of satisfiability



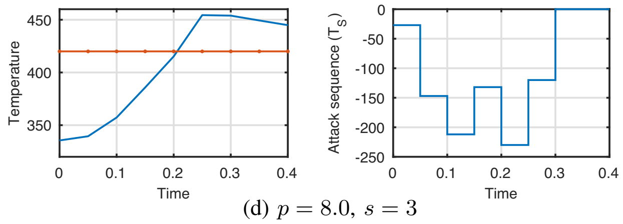
(a) $p = 8.0,\ s = 0$



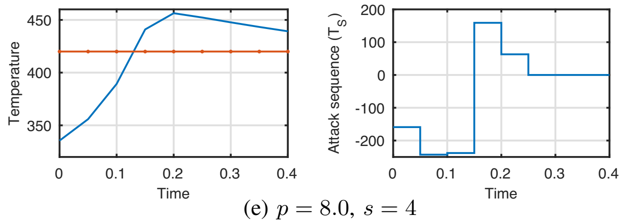
(b) $p = 8.0,\ s = 1$



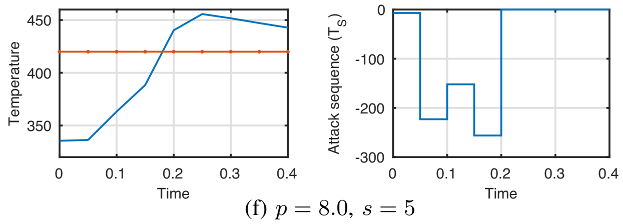
(c) $p = 8.0,\ s = 2$



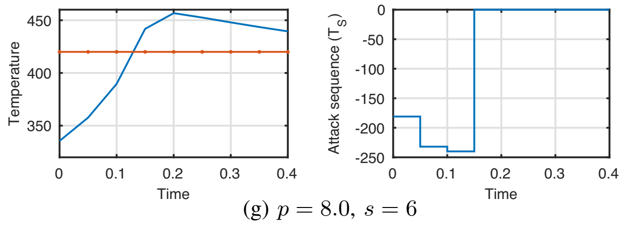
(d) $p = 8.0,\ s = 3$



(e) $p = 8.0,\ s = 4$



(f) $p = 8.0,\ s = 5$



(g) $p = 8.0,\ s = 6$

Fig. 1: Validation of attacks from Table I on the ODE model of the CSTR system. The overheat condition is the red straight line in the left plots. Time is given in hours.

(returning a candidate attack sequence) and unsatisfiability (guaranteeing safety under the given modelling assumption).

It is worth mentioning that with our technique the complexity of the state space is not directly related to that of the underlying system of equations, and thus to its dimensionality, but rather to the quantity of nondeterminism in the program, which for our case study is the size of the attack vector, which grows with the considered time horizon.

Our approach is quite general in that it can analyse piecewise polynomial dynamics. However, it can require significant computational effort, especially on unsatisfiable instances. Future work will aim at improving our approach in different directions.

First, it would be interesting to experiment with different solvers, such as optimised ones [26] or other families of solvers that have been reported to scale particularly well on large instances [27]. It would also be useful to simplify the encoding, for example by adapting techniques used in hardware circuits that allow overflow checking using less bits [28]. On large programs, adopting a block-based intermediate representation that avoids full program flattening and produces encodings whose size do not depend on the loop bound [29], or techniques for incremental analysis [30] might be beneficial. Evaluating the effectiveness of possible techniques for unbounded program analysis, such as those based on inductive reasoning [31], would be a very interesting step towards lifting the limitations on the time horizon.

Second, it might be possible to prune the search space with different strategies: including further logic constraints in the SAT problem from reach-set computations based on the physical model; using abstraction-based decision procedures that combine under- and over-approximations, as proposed for bitvector arithmetics [32] and floating-point programs [33]; performing abstract interpretation to automatically minimise the integer precision of fixed-point variables [34]; adapting lightweight static analyses based on intervals with probabilistic bounds [35] and affine arithmetics [36], developed for numerical error estimations in DSP designs.

Third, one could evaluate if further techniques for different semantics, for example using floating-point arithmetics or infinite precision (i.e., the theory of the reals) with Satisfiability Modulo Theories (SMT) can improve performance. Word-level reasoning is naturally supported in SMT-based encodings [37]. It would be interesting to evaluate any benefits of tailoring these intuitions to the specific context of programs for control systems. Structural information could also be used to speed up the analysis by altering the heuristic choices of a SAT solver [38].

Fourth, it will be worthwhile investigating the combination of numerical optimisation with SAT solvers (as in the case of piecewise linear hybrid models, see e.g. [39]), that would also allow synthesising "optimal" attacks, such as maximising the sparsity or minimising the intensity of the spoofing signal.

References

[1] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham *et al.*, "Experimental security analysis of a modern automobile," in *IEEE Symposium on Security and Privacy*, 2010, pp. 447–462.

[2] Y. Shoukry, P. Martin, P. Tabuada, and M. Srivastava, "Non-invasive spoofing attacks for anti-lock braking systems," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2013, pp. 55–72.

[3] A. Teixeira, I. Shames, H. Sandberg, and K. H. Johansson, "Revealing stealthy attacks in control systems," in *50th Annual Allerton Conference on Communication, Control, and Computing*, 2012, pp. 1806–1813.

[4] F. Pasqualetti, F. Dörfler, and F. Bullo, "Attack detection and identification in cyber-physical systems," *IEEE Transactions on Automatic Control*, vol. 58, no. 11, pp. 2715–2729, 2013.

[5] A. Tiwari, B. Dutertre, D. Jovanović, T. de Candia, P. D. Lincoln, J. Rushby, D. Sadigh, and S. Seshia, "Safety envelope for security," in *Proceedings of the 3rd International Conference on High Confidence Networked Systems (HiCoNS)*, 2014, pp. 85–94.

[6] H. Fawzi, P. Tabuada, and S. Diggavi, "Secure estimation and control for cyber-physical systems under adversarial attacks," *IEEE Transactions on Automatic Control*, vol. 59, no. 6, pp. 1454–1467, 2014.

[7] M. Pajic, J. Weimer, N. Bezzo, P. Tabuada, O. Sokolsky, I. Lee, and G. J. Pappas, "Robustness of attack-resilient state estimators," in *ACM/IEEE 5th International Conference on Cyber-Physical Systems (ICCPS)*, 2014, pp. 163–174.

[8] Y. Mo and B. Sinopoli, "Secure estimation in the presence of integrity attacks," *IEEE Transactions on Automatic Control*, vol. 60, no. 4, pp. 1145–1151, 2015.

[9] Y. Shoukry, A. Puggelli, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, and P. Tabuada, "Sound and complete state estimation for linear dynamical systems under sensor attacks using satisfiability modulo theory solving," in *American Control Conference (ACC), 2015*. IEEE, 2015, pp. 3818–3823.

[10] Y. Shoukry, M. Chong, M. Wakaiki, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, J. P. Hespanha, and P. Tabuada, "SMT-based observer design for cyber-physical systems under sensor attacks," in *ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*, 2016, pp. 1–10.

[11] Y. Shoukry, P. Nuzzo, A. Puggelli, A. L. Sangiovanni-Vincentelli, S. A. Seshia, and P. Tabuada, "Secure state estimation for cyber physical systems under sensor attacks: a satisfiability modulo theory approach," *IEEE Transactions on Automatic Control*, 2017.

[12] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009, ch. Satisfiability Modulo Theories.

[13] I. Pustogarov, T. Ristenpart, and V. Shmatikov, "Using program analysis to synthesize sensor spoofing attacks," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017, pp. 757–770.

[14] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[15] A. Bemporad and M. Morari, "Verification of hybrid systems via mathematical programming," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, F. Vaandrager and J. van Schuppen, Eds., 1999, vol. 1569, pp. 31–45.

[16] F. Torrisi and A. Bemporad, "Discrete-time hybrid modeling and verification," in *Proc. 40th IEEE Conf. on Decision and Control*, Orlando, Florida, 2001, pp. 2899–2904.

[17] A. Cox, S. Sankaranarayanan, and B. E. Chang, "A bit too precise? verification of quantized digital filters," *STTT*, vol. 16, no. 2, pp. 175–190, 2014.

[18] I. V. de Bessa, H. Ismail, L. C. Cordeiro, and J. E. C. Filho, "Verification of fixed-point digital controllers using direct and delta forms realizations," *Design Autom. for Emb. Sys.*, vol. 20, no. 2, pp. 95–126, 2016.

[19] B. Akbarpour, S. Tahar, and A. Dekdouk, "Formalization of fixed-point arithmetic in HOL," *Formal Methods in System Design*, vol. 27, no. 1-2, pp. 173–200, 2005.

[20] B. Akbarpour and S. Tahar, "Error analysis of digital filters using HOL theorem proving," *J. Applied Logic*, vol. 5, no. 4, pp. 651–666, 2007.

[21] R. Majumdar, I. Saha, K. C. Shashidhar, and Z. Wang, "CLSE: closed-loop symbolic execution," in *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, 2012, pp. 356–370.

[22] *Programming languages — C — Extensions to support embedded processors*. New York: Institute of Electrical and Electronics Engineers, 1987, iSO/IEC Technical Report 18037:2008(E).

[23] E. M. Clarke, D. Kroening, and K. Yorav, "Behavioral consistency of C and verilog programs using bounded model checking," in *Proceedings of the 40th Design Automation Conference, DAC*, 2003, pp. 368–371.

[24] N. Eén and N. Sörensson, "An extensible SAT-solver," in *6th International Conference on Theory and Applications of Satisfiability Testing, SAT*, 2003, pp. 502–518.

[25] B. W. Bequette, *Process control: modeling, design, and simulation*. Prentice Hall Professional, 2003.

[26] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 2009, pp. 399–404.

[27] M. Sheeran and G. Stålmarck, "A tutorial on stålmarck's proof procedure for propositional logic," *Formal Methods in System Design*, vol. 16, no. 1, pp. 23–58, 2000.

[28] M. J. Schulte, P. I. Balzola, A. Akkas, and R. W. Brocato, "Integer multiplication with overflow detection or saturation," *IEEE Trans. Computers*, vol. 49, no. 7, pp. 681–691, 2000.

[29] P. Ashar, M. K. Ganai, A. Gupta, F. Ivancic, and Z. Yang, "Efficient sat-based bounded model checking for software verification," in *International Symposium on Leveraging Applications of Formal Methods (ISoLA)*, 2004, pp. 157–164.

[30] A. Nadel, V. Ryvchin, and O. Strichman, "Ultimately incremental SAT," in *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, 2014, pp. 206–218.

[31] N. Eén and N. Sörensson, "Temporal induction by incremental SAT solving," *Electr. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 543–560, 2003.

[32] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. A. Brady, "An abstraction-based decision procedure for bit-vector arithmetic," *STTT*, vol. 11, no. 2, pp. 95–104, 2009.

[33] A. Brillout, D. Kroening, and T. Wahl, "Mixed abstractions for floating-point arithmetic," in *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD*, 2009, pp. 69–76.

[34] A. Ioualalen and M. Martel, "Synthesis of arithmetic expressions for the fixed-point arithmetic: The sardana approach," in *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing, DASIP 2012, Karlsruhe, Germany, October 23-25, 2012*, 2012, pp. 1–8.

[35] C. F. Fang, R. A. Rutenbar, and T. Chen, "Fast, accurate static analysis for fixed-point finite-precision effects in DSP designs," in *2003 International Conference on Computer-Aided Design, ICCAD 2003, San Jose, CA, USA, November 9-13, 2003*, 2003, pp. 275–282.

[36] C. F. Fang, R. A. Rutenbar, M. Püschel, and T. Chen, "Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling," in *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, 2003, pp. 496–501.

[37] L. Hadarean, K. Bansal, D. Jovanovic, C. Barrett, and C. Tinelli, "A tale of two solvers: Eager and lazy approaches to bit-vectors," in *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, 2014, pp. 680–695.

[38] M. K. Ganai and A. Gupta, *SAT-Based Scalable Formal Verification Solutions*, ser. Series on Integrated Circuits and Systems. Springer, 2007.

[39] A. Bemporad and N. Giorgetti, "Logic-based methods for optimal control of hybrid systems," *IEEE Transactions on Automatic Control*, vol. 51, no. 6, pp. 963–976, 2006.