*South American Journal of Logic*
Vol. 3, n. 1, pp. 25–57, 2017
ISSN: 2446-6719

# Logical Analysis of Cyber Vulnerability and Protection

Esther David, Dov Gabbay, Guy Leshem and Students of CS Ashkelon

**Abstract**

The growth in IoT will far exceed that of other connected devices. In contrast to the rapid assimilation of the IoT devices and interfaces, the technology that ensures these systems are safe and secure is left behind. The IoT safety, security and privacy is composed of several key aspects. Attention has already been given to vulnerabilities associated with networked systems as authentication and encryption. However there exist more vulnerabilities issues in IoT systems for which we aim to develop a detection methodology and propose possible solutions. Specifically, here we propose techniques for security verification of an IoT system that mainly involves inconsistency and vulnerability. The proposed methods are logic based techniques that were developed especially for the new challenges that the IoT environment is facing.

## 1 Motivation

The Internet of Things (IoT), which excludes PCs, tablets and smartphones, is the network of physical objects that contain embedded technology to communicate and sense or interact with their internal states or the external environment [2, 6]. Peter Middleton, research director at Gartner said: "The growth in IoT will far exceed that of other connected devices. By 2020, The number of PCs, smartphones, and tablets in use will reach about 7.3 billion units, in contrast, the IoT will have expended at a much faster rate, resulting in a population of about 26 billion units at that time". In response to this phenomena, the Industry develops universal protocols and interfaces (e.g., AllJoyn [3], Thread [7]) to allow these devices to communicate and inter-operate.

In contrast to the rapid assimilation of the IoT devices and interfaces, the technology that ensures these systems are safe and secure is left behind [4, 8]. The current available security solutions including hardware and software for non IoT devices as PCs is not adequate for IoT devices mainly for limited computational resources and capabilities reasons [5]. Namely, the IoT systems

are in common usage, but without the safety guarantees. This becomes critical when we consider medical care base IoT devices as an insulin pump that may directly affect humanity [6]. Moreover, since IoT devices will typically be embedded deep inside networks, they are attractive attack targets and may become the "weakest-link" for breaking into a secure IT infrastructure and leaking sensitive information [8].

This sensitive situation according to [8] is due to the fact that most vendors only deal with parts of IoT ecosystem and, typically, their priorities have been providing novel functionality, getting their products to market soon, and making them easy to use, while the security issues are of lower priority.

The IoT safety, security and privacy is composed of several aspects. Attention has already been given to vulnerabilities associated with networked systems as authentication and encryption [6]. However there exist more vulnerabilities issues [8] in IoT systems for which we aim to develop a detection methodology and propose possible solutions.

Against this background in this paper we propose techniques for security verification of an IoT system that mainly involves inconsistency and vulnerability. For both aspects we assume the system is defined by a set of rules that describe the desirable progressing of the system. To illustrate an inconsistency, consider the case of two IoT devices that are controlling the door of a house and under certain condition simultaneously, one devices command to open the door while the second command to close the door. To illustrate a vulnerability situation assume the existence of the following rule: "if the temperature goes over 30 Celsius degrees, open the door". Under these circumstances, a hacker may compromise the controlling unit of an air conditioner to heat the house and indirectly cause the door to be open.

Interestingly we discover that the classic logic theory is not appropriate for this problem for three main reasons: (i) the assumption that rules are transitive in the classic logic theory does not hold in IoT since all the rules must be performed simultaneously rather than sequentially, (ii) the rules are not a simple "if then" structure but rather "if then do", namely we define action rather condition, (iii) the predicates are no longer Boolean values that may be just "true" or "false" but rather it also include the case of an "unknown" value.

The most relevant research belongs to Liang *et al.* [6]. They propose to translate the rules to a programming language while we propose algorithm and logic model that will directly be applied on the rules that are written in a way that is easy for human to understand. Moreover, according to our proposed algorithms we are able to detect loops while their algorithm runs $K$ fixed iterations. Specifically they state as follows:

> "The Safety Engine adopts a bounded model checking flavor of solution. It begins by unrolling the control flow graph of the IoT

app for a fixed number steps ($k$). Conceptually checking for unsafe situations extends within this number of $k$ steps. To perform this, $C\sharp$ code is generated where each rule becomes multiple statements including an IF statement along with a set of supporting additional ones".

Next we provide a use case study to build up the model we propose. Next we provide the techniques for detecting inconsistency and the vulnerability of the IoT system.

## 2 Orientation and a Case Study

This section gives a simple example to explain our language, our problem and our remedies.

1. The programming language is a time-action-logic language studied in Ashkelon Academic College in 2015. Call it ATAL – Ashkelon Time Action Logic.

2. The sample problem is very simplified but adequate for presenting our model in principle.

   We have a structure in a very cold area collecting important and sensitive data. The structure contains important computers and data. It has to be kept at a temperature range $0 \leq T \leq 20$.

   It has a heating system controlled by a computer program which adjusts the temperature by switching the heating system on and off. If the heating does not respond then in addition the program sends a signal to maintenance which then comes to check within a short time. If the malfunction in the heating system is that it is not switching off and the temperature rises above 20, then the computer opens a window to cool the place down.

3. The time action logic language describing the above scenario has a set of atoms $\{a, b, c, \ldots\}$ representing statements of positive facts. Some of them may be determined by sensors not under our control, such as "the temperature is 15 degrees" and some are under our control such as "the window is open". To be able to completely handle facts which are under our control, we also use a symbol "$\sim$" for forming more sentences out of atoms, the strong negation of facts, i.e. the definite negation of the fact, e.g. $\sim$window open = window closed, which can be put in databases. We expect databases of facts to be consistent, so a database of facts cannot

contain $x$ and $\sim x$ together but it may not contain any of them, i.e. neither $x$ nor $\sim x$. We can also query a factual database, say database $\Delta$. We can query $\Delta?x$, and answer yes iff $x \in \Delta$, or $\Delta? \sim x$ and answer yes iff $\sim x \in \Delta$. We also have a symbol "$\neg$" which can be used only in queries from databases. The answer to the query $\Delta?\neg x$ is yes iff $x$ is not in $\Delta$. Similarly for the query $?\neg \sim x$. Thus $\neg a$ reads that $a$ is not included in the database, so we may have that neither window open nor $\sim$window open are recorded in the database, and thus the answer to both the queries ?$\neg$window open and to ?$\neg$ $\sim$window open may be affirmative. Finally we also have the action symbol "$\Rightarrow$" which can be used to formulate rules. The arrow $\Rightarrow$ is not logical implication but an action symbol so for example we may write

$$(\neg\text{window open}) \text{ and } (\neg \sim\text{window open}) \Rightarrow \text{window open}$$

which reads

If the database does not record anything about the window then open the window.

Exact formal definitions will be given later. Meanwhile, let us write what we have:

(a) atomic sentences $a, b$,

(b) negation for forming sentences $\sim a$ from atoms $a$.

(c) databases $\Delta$ containing both types of sentence of the form $x$ and $\sim x$.

(d) using the symbol $\neg$, we can form queries of the form $?\neg a, ?\neg \sim a, \ldots$

(e) we also use the conjunction symbol to form conjunction of queries $Q \wedge Q'$

(f) we use the action arrow $\Rightarrow$ to form action rules.
The structure of the rule is conjunction of queries of facts $\Rightarrow$ fact data item. Since the facts appearing to the left of the "$\Rightarrow$" are queries, we need not write the question mark.

The meaning of, say,

$$a\wedge \sim b \wedge \neg \sim c \wedge \neg d \Rightarrow e$$

is the following, when applied to a given database $\Delta$:

If it is confirmed that $a$ and $\sim b$ are in $\Delta$ (i.e. the answer to each of the queries $\Delta?a$ and $\Delta? \sim b$ is yes and $\sim c$ and $d$ are not in $\Delta$ (i.e. the answer to each of the queries $\Delta?\neg \sim c$ and $\Delta?\neg d$ is yes) then execute/make true

that $e$ is in $\Delta$, namely add $e$ to $\Delta$ as additional data. This is an operation that changes $\Delta$ into a new $\Delta'$.

If $\sim e$ already happens to be in $\Delta$ when we want to add $e$ to the data as a result of the application of a rule, then to maintain consistency, we take the already existing $\sim e$ out of the database $\Delta$ and insert and keep only $e$ in $\Delta$. Similarly if we want to add $\sim e$ to the database (after executing a rule of the form $Q \Rightarrow\sim e$), then we take out from the database e and replace it by $\sim e$. Let us understand the sequence $\sim\sim x$ as $x$, i.e. $\sim\sim x = x$, then we can write for any fact $x$ which we want to put into the database $\Delta$ after execution of a rule to obtain $\Delta'$: $\Delta' = (\Delta - \{\sim x\} \cup \{x\})$.

4. Data can come into the database also as input from sensors e.g. $T = 15$ (temperature is 15). In this case the new temperature value naturally over-writes the old value. We do not have to execute anything ourselves just read the number from the sensor.

5. We have a starting database $\Delta_0$.

6. Time moves in clock ticks $t = 0, 1, 2, \ldots$. Action rules are all fired in parallel at each tick of time $t$, updating the database from $\Delta_t$ to $\Delta_{t+1}$ and sensors report at each tick.

7. A set of rules and a starting database are inconsistent if at some stage tick $i$ we try to execute the rules in parallel on $\Delta_i$ and we get a demand to execute both an $x$ and a $\sim x$ at the same time.

   We shall formally define all of this in the next section. Meanwhile, we have enough to present the problem.

   A *program* is a pair of an initial database and a set of rules.

   Let us write a simple program.

   Initial Database $\Delta_0$.

   $$\Delta_0 = \{T = 10 \text{ (i.e. the sensor for temperature shows the number 10), the window is closed, heating is off}\} = \{T = 10, \sim W, \sim H\}$$

   If we want to be very mathematical and formal we should say the expression "$T = 10$" is a fact in the database. So the language contains sensor statements of the form "$T = 1$", "$T = 2$",.... Call these sensor *facts*. These are not allowed to be negated and are not allowed to appear in the right hand side of the double arrow "$\Rightarrow$". We might allow the symbol $T =?$ in the right hand side, in this case its execution is performed by reading the temperature sensor and inserting the appropriate fact in the database.

We can also allow for facts of the form "$T < 0$" and "$T > 20$", etc.

$$\begin{array}{lcl} W & = & \text{window open} \\ \sim W & = & \text{window is closed} \\ H & = & \text{heating on} \\ \sim H & = & \text{heating off} \end{array}$$

$\neg x$ means that

$\neg x$ = we look at the database and do not find $x$, i.e. $x$ is not recorded in the database.

$$M = \text{ send message to maintenance.}$$

8. Let $\Delta$ be a database and $x \in \{a, \sim a, \neg a, \neg \sim a\}$, then we write $\Delta \vdash x$ to mean as follows $\Delta \vdash x$ is not a proof theoretic symbol, but another suggestive way of writing membership in $\Delta$. We do not have logical implication, the $\Rightarrow$ is not logical implication but an action symbol:

$$\begin{array}{lll} \Delta \vdash a & \text{iff} & a \in \Delta \\ \Delta \vdash \sim a & \text{iff} & \sim a \in \Delta \\ \Delta \vdash \neg a & \text{iff} & a \notin \Delta \\ \Delta \vdash \neg \sim a & \text{iff} & \sim a \notin \Delta \\ \Delta \vdash x \wedge y & \text{iff} & \Delta \vdash x \text{ and } \Delta \vdash y \end{array}$$

9. **Rules** $R$ for our example program (so the program will be $(\Delta_0, R)$):

   (a) $T \leq 0 \Rightarrow H$
   (b) $T \leq 0 \wedge \neg H \Rightarrow M$
   (c) $T \geq 20 \Rightarrow \sim H$
   (d) $T \geq 20 \wedge \neg \sim H \Rightarrow M$
   (e) $T \geq 20 \wedge H \Rightarrow W$
   (f) $T \leq 0 \wedge W \Rightarrow \sim W$

   Recall that a program is a pair $(\Delta_0, R)$. We can let time run, $t = 0, 1, 2, \ldots$ and apply all the rules repeatedly in parallel at each stage to obtain a sequence of databases, $\Delta_0, \Delta_1, \Delta_2, \ldots$.

10. A program is *consistent* (*correct*) if we cannot have at any stage time $t$ when we apply the rules to $\Delta_t$, we get the demand from the rules for some $x$ , to put both $x$ and $\sim x$ into the database $\Delta_{t+1}$.

11. Let us check consistency correctness of the above program $(\Delta_0, R)$.

    (a) Let us agree that the temperature is read from a sensor and at any time $t$, there is exactly one number $n$ such that $T = n \in \Delta_t$.

    (b) We check that at any $t$ we cannot have for any $x$ that we are asked to execute both $x, \sim x$, i.e. we are asked to put both into $\Delta_t$.

    For $\Delta_0$ this is true. Assume true for $\Delta_t$. Check $\Delta_{t+1}$.

    If we understand the rule (where $y$ in the rule has the form $y = a$, or $\sim a$ but *not* the form $\neg z$)

$$\bigwedge_{i=1}^{k} x_i \Rightarrow y$$

    to mean that we look at $\Delta_t$ and see that $\Delta_t \vdash x_i, i = 1, \ldots, k$ then we put $y$ in $\Delta_{t+1}$ and take out $\sim y$ from $\Delta_t$ if it is there, and let $\Delta_{t+1} = \{y\} \cup \Delta_t - \{\sim y\}$, where

$$\sim y = \sim a \text{ if } y = a$$
$$\sim y = a \text{ if } y = \sim a$$

    (i.e. $\sim\sim a = a$).

    Following the above meaning of "$\Rightarrow$", the above program is consistent/correct.

12. **Checking cyber vulnerability**

    (a) Define vulnerability. By *vulnerability* we mean certain conjunctions of sentences we do not want to be in any database $\Delta_t$. If the program is correct then this will not happen, but if the program is cyber hacked into, the hacker might make it happen. So we want to identify what a hacker might do. In this case we do not want to have for any $t$,

$$\Delta_t \vdash \neg M \wedge H \wedge W \wedge T \geq 20$$

    call this $V$. (We need to assume that the temperature needs for the time clock to move forward more than two time ticks for it to drop one degree!) $V$ says that there is no record of a message sent to maintenance and the heating is on and the window is open and the temperature is 20 or above.

    When the temperature reaches 20 degrees, the program should switch off the heating (rule c). If the heating does not switch off then in the next time tick (d) and (e) will activate. If (d) is also not activated then in the next time tick we will have $\neg M \wedge H \wedge W \wedge T \geq 20$.

(b) Now that we have identified vulnerability, we pretend we are cyber hackers and ask what is needed to achieve the vulnerable set up? This we find by the process of *abduction*. We shall discuss abduction later on in the paper, but for the purpose of this orientation discussion, think of the vulnerability $V$ as a goal, and of Abduction as a Planning System for achieving the goal. The Abduction/Planning will tell us what we need and this way we expect the cyber hacker to hack our program to make it possible to get what is needed.

**Question.** What can give me

$$V = [\neg M \wedge H \wedge W \wedge T \geq 20] \text{ at stage } \Delta_t?$$

i. We need $T \geq 20$
ii. To get $W$ we need (e) to activate. So we need (c) not to activate. So we need to delete it.
iii. To get $\neg M$ we need (d) not to activate. So we need to delete rule (d) because if (e) activates then also (d) will activate.

**Answer to the question.** So a cyber attack for the $V$ vulnerability will delete rules (c) and (d). We must protect these rules or at least be informed if they are interfered with.

**Remedy.** Set up an automaton sending message if a rule is deleted.

# 3 A formal model, basic concepts

**Definition 3.1 (Syntax)**

1. (a) *Let* $Q = \{a, b, c, \ldots\}$ *be a set of distinct atoms.*
   (b) *Let "*$\sim$*" be a unary operator symbol.*
   (c) *The expression* $\sim x$ *for* $x \in Q$ *is referred to as "negated atom".*
   (d) *By the word "literal" we mean either an atom or a negated atom.*
   (e) *We let* $\sim\sim x$ *be identical to* $x$*.*
2. (a) *A literal is also referred to as a "fact"*
   (b) *A database* $\Delta$ *is a finite set of literals.*
   (c) $\Delta$ *is* consistent *iff it does not contain any* $x$ *and* $\sim x$ *together.*

*(d) Let $\Delta_1, \Delta_2$ be two databases. We use the notation*

$$\Delta_1 + \Delta_2 = \Delta_1 \cup \Delta_2 = \{y \mid y \in \Delta_1 \text{ or } y \in \Delta_2\}.$$

*(e) Let $\Delta_1 - \Delta_2$ be $\{y \mid y \in \Delta_1$ and $y \notin \Delta_2\}$.*

*(f) Let $\Theta$ be a database. Define $\sim \Theta$ to be $\{\sim y \mid y \in \Theta\}$.*

**Definition 3.2 (Queries)**

1. *Let $\Delta$ be a database and $x$ be a literal. The expression $\Delta?x$ indicates a query of $x$ from $\Delta$.*

   *We write*

   $$\begin{array}{ll} \Delta?x = 1 & \textit{iff} \quad x \in \Delta \\ \Delta?x = 0 & \textit{iff} \quad x \notin \Delta \end{array}$$

2. *Let $\Delta, \Theta$ be two databases. The query $\Delta?\Theta$ means as follows*

   $$\begin{array}{ll} \Delta?\Theta = 1 & \textit{iff} \quad \Theta \subseteq \Delta \\ \Delta?\Theta = 0 & \textit{iff} \quad \Theta \not\subseteq \Delta \end{array}$$

**Definition 3.3 (Action rules)**

1. *Let "$\Rightarrow$" be a special symbol for action. By an* action rule *we mean an expression of the form $\Theta \Rightarrow x$, where $\Theta$ is a finite set of literals and $x$ is a literal.*

2. *By a* program *we mean a pair $\mathbb{P} = (\Delta, \mathbb{R})$ where $\Delta$ is a database and $\mathbb{R}$ is a finite set of action rules.*

# 4 Checking for consistency

When we are given a program, as defined in the previous section, we need to check that this program is consistent. This means that the initial database is consistent and also as we execute the rule, we remain consistent. To make sure of that we need to define exactly how the rules are executed forward, be able to track inconsistencies and be able to fix them as we go forward. We also need to check vulnerability when our program is consistent but we leave this for the next section.

**Definition 4.1 (Forward execution for inconsistency detection)**

1. *A database $\Delta$ is* inconsistent*, if for some $y$, we have that both $y$ and $\sim y$ are in $\Delta$. Otherwise $\Delta$ is* consistent*.*

2. *Let $\mathbb{P} = (\Delta, \mathbb{R})$ be a program. We define a sequence of databases $\Delta_0, \Delta_1 \ldots$ generated by the program $\mathbb{P}$ in parallel:*

   (a) *Let $\Delta_0 = \Delta$. This is the initial data. If it is inconsistent , then stop and declare $\mathbb{P}$ as inconsistent.*

   (b) *Assume $\Delta_n$ has been defined and is consistent. Let $\Delta_n^{\mathbb{R}}$ be the set of all $x$ such that for some rule $\Theta \Rightarrow x$ we have $\Theta \subseteq \Delta_n$.*

   **Case 1.** *$\Delta_n^{\mathbb{R}}$ is not consistent. Then stop and declare that $\mathbb{P}$ is not consistent.*

   **Case 2.** *$\Delta_n^{\mathbb{R}}$ is consistent, then let $\Delta_{n+1} = (\Delta_n - \sim \Delta_n^{\mathbb{R}}) + \Delta_n^{\mathbb{R}}$.*

**Proposition 4.2** *Let $\mathbb{P} = (\Delta, \mathbb{R})$ be a program and let $\Delta_0, \Delta_1, \Delta_2, \ldots$ be a sequence of databases generated by $\mathbb{P}$.*

*Then either the sequence is finite or for some $m, n, m \neq n, \Delta_m = \Delta_n$.*

**Proof.** Let $Q$ be the set of all literals appearing in $\mathbb{P}$. Since both $\Delta$ and $\mathbb{R}$ are finite then $Q$ is finite. If the number of elements in $Q$ is $k$ then $Q$ can generate at most $2^k$ different databases. Therefore, since $\{\Delta_n\}$ is infinite, the proposition follows. ■

**Example 4.3**

1. *Let $\Delta = \{a, b\}$.*

   *Let $\mathbb{R}$ contain:*

   $$(r1) \quad \{a\} \Rightarrow \sim b$$
   $$(r2) \quad \{a, \sim b\} \Rightarrow b.$$

   *We compute the sequence of databases that $\mathbb{P} = (\Delta, \mathbb{R})$ generates in parallel.*

   **Step 1.** $\Delta_1 = \Delta$

   **Step 2.**

   $$\Delta_1^{\mathbb{R}} = \{\sim b\}$$
   $$\sim \Delta_1^{\mathbb{R}} = \{b\}$$

   *$\Delta_1^{\mathbb{R}}$ is consistent.*

   *Therefore*

   $$\begin{aligned} \Delta_2 &= (\Delta_1 - \sim \Delta_1^{\mathbb{R}}) \cup \Delta_1^{\mathbb{R}} \\ &= (\{a, b\} - \{b\}) \cup \{\sim b\} \\ &= \{a, \sim b\}. \end{aligned}$$

   **Step 3.** $\Delta_2^{\mathbb{R}} = \{\sim b, b\}$.

   *$\Delta_2^{\mathbb{R}}$ is not consistent. So we stop.*

2. *Let* $\Delta = \{a\}$.

   *Let* $\mathbb{R}$ *contain*

$$\begin{array}{ll}(r_1): & \{a\} \Rightarrow \sim a \\ (r_2): & \{\sim a\} \Rightarrow a.\end{array}$$

   *We compute the sequence of databases generated by this program. We get*

$$\begin{array}{rcl}\Delta_1 & = & \Delta \\ \Delta_1^{\mathbb{R}} & = & \{\sim a\} \\ \Delta_2 & = & \{\sim a\} \\ \Delta_2^{\mathbb{R}} & = & \{a\} \\ \Delta_3 & = & \{a\}\end{array}$$

   *We have that the program is consistent and* $\Delta_1 = \Delta_3$*, and it oscillated between* $\{a\}$ *and* $\{\sim a\}$.

We now want to discuss how to resolve and correct inconsistencies. We need auxiliary definitions.

**Definition 4.4 (**$(\Delta, \mathbb{R})$ **support)** *Let* $\Delta$ *be a database and let* $\mathbb{R}$ *be a set of rules. Write the rules as a sequence enumerated as*

$$\begin{array}{l}(r_1): \Theta_1 \Rightarrow x_1 \\ \vdots \\ (r_n): \Theta_n \Rightarrow x_n\end{array}$$

*Let* $x \in \{x_1, \ldots, x_n\}$*. We define the* $(\Delta, \mathbb{R})$ support *of* $x$ *to be the set*

$$S(\Delta, \mathbb{R}, x) = \{r_i \in \mathbb{R} \mid \Theta_i \subseteq \Delta \text{ and } r_i \text{ is } \Theta_i \Rightarrow x\}.$$

*In other words, it is all the rules which "generate"* $x$ *from* $\Delta$.

**Definition 4.5 (Inconsistency resolution rules)** *The following is a list of Inconsistency Resolution Rules (IRR) which can be used when needed.*

*Let* $\Delta$ *be a database and let* $\mathbb{R} = \{r_1, \ldots, r_n\}$ *be a set of rules. Let* $\Delta^{\mathbb{R}} = \{y \mid$ *for some rule* $(r): \Theta \Rightarrow y$*, we have* $\Theta \subseteq \Delta\}$.

*Let* $S(\Delta, \mathbb{R}, y)$ *be the support of* $y$.

*We list the following inconsistency resolution rules:*

1. *Do nothing IRR.*

   *Whenever both* $y$ *and* $\sim y$ *are in* $\Delta^{\mathbb{R}}$ *then delete both* $y$ *and* $\sim y$.[1]

[1]Remember that we are dealing with action rules here, and so the inconsistency arises from opposing instructions, one to add $y$ to the database and one to add $\sim y$. So IRR-Do Nothing says "do nothing".

2. *Specificity IRR.*
   *Whenever $y$ and $\sim y$ are in $\Delta^{\mathbb{R}}$ and the support for $y$ is more specific than the support for $\sim y$, then take $\sim y$ out of $\Delta^{\mathbb{R}}$.*

   *Where we say the support of $y$ is more specific than the support for $\sim y$ iff whenever $(r_i) : \Theta_i \Rightarrow \sim y$ is activated (i.e. $\Theta_i \subseteq \Delta$) then for some $(r_j) : \Theta_j \Rightarrow y$ we have $\Theta_j \supsetneqq \Theta_i$ and $\Theta_j \subseteq \Delta$, i.e. $\Theta_j$ is more specific than $\Theta_i$.*

3. *Strength IRR.*
   *Whenever $y$ and $\sim y$ are in $\Delta^{\mathbb{R}}$ and $|S(\Delta, \mathbb{R}, y)| > |S(\Delta, \mathbb{R}, \sim y)|$ then take out $\sim y$. Where for a set $A$, $|A|$ is the number of elements in $A$.*

4. *Priority IRR.*

   *We have a priority ordering on pairs $\{y, \sim y\}$ and whenever both $y, \sim y \in \Delta^{\mathbb{R}}$, we take out the literal of lower priority.*

**Definition 4.6 (Algorithm for resolving inconsistency using IRR rules)**

1. *We have a preference ordering on the IRR rules. Whenever we detect an inconsistency, we apply the rules in the order of preference. Note that because of the* do nothing *rule, any inconsistent $\Delta^{\mathbb{R}}$ becomes a consistent $\Delta^{\mathbb{R}}_{\text{Con}}$ after applying the rules, because even if all the other rules are inconclusive, the* do nothing *rule is always conclusive.*

2. *We can modify the forward sequence of Definition 4.1 by modifying the inductive definition of $\Delta_{n+1}$ to be*

$$\Delta_{n+1} = (\Delta_n - \sim \Delta^{\mathbb{R}}_{n,\text{Con}}) + \Delta^{\mathbb{R}}_{n,\text{Con}}.$$

**Example 4.7** *We revisit item 1 of Example 4.3. We have $\Delta = \{a, b\}$ and the rules*

$$\begin{array}{ll} (r_1): & \{a\} \Rightarrow \sim b \\ (r_2): & \{a, \sim b\} \Rightarrow b \end{array}$$

*In this case we get that $\Delta^{\mathbb{R}}_2$ is $\{\sim b, b\}$.*

*We note: $b$ has a more specific support. If we use this fact then $\Delta^{\mathbb{R}}_{2,\text{Con}} = \{b\}$ and the database $\Delta_3$ would be $\{a, b\}$ as $\sim b \in \Delta_2$ will be over-written by $b$.*

*If we use the do nothing IRR then $\Delta_3$ would be the same as $\Delta_2 = \{a, \sim b\}$.*

# 5 Checking for vulnerability

By *vulnerability* we mean certain conjunctions of sentences we do not want to be in any database $\Delta_n$, for any $n$. To achieve this, we need to check whether any of these sentences do appear at any stage, in which case we must take steps to ensure that they fail, probably by amending the initial data or the rules.

If these vulnerability sentences do not succeed we must identify what changes might reasonably make them succeed, and take measures to guard against hackers making these changes.

From the technical point of view, we need algorithms that can tell us the following:

**Vulnerability checking case 1.** Given $c$, does $c$ succeed and exactly in what ways (data and rules) does it succeed, and how can we make it not succeed.

**Vulnerability Checking case 2.** Given $c$, does $c$ not succeed and whether there exist some sensor data that will make it succeed.

## 5.1 Motivating example

**Example 5.1** *Consider the database* $\Delta = \{a, b\}$. *Consider the five rules:*

$$\begin{array}{ll} (r_0): & \{a, \sim b\} \Rightarrow c \\ (r_1): & \{a\} \Rightarrow \sim a \\ (r_2): & \{b\} \Rightarrow \sim b \\ (r_3): & \{\sim a\} \Rightarrow a \\ (r_4): & \{\sim b\} \Rightarrow b. \end{array}$$

*We ask: can we go forward and get* $c$ *in some step?*

*Going forward we find that the database oscillates between* $\Delta_1 = \Delta = \{a, b\}$ *and* $\Delta_2 = \{\sim a, \sim b\}$. *We never get* $c$.

*If we go goal directed, we can ask: can we get* $c$ *at some stage* $n$ *for some* $n$*? Well, if* $c \in \Delta$, *then* $n = 1$ *would get* $c$. *But* $c \notin \Delta$, *however rule* $(r_0)$ *can give us* $c$ *if activated at stage* $n - 1$. *So we need to ask whether both* $a$ *and* $\sim b$ *can succeed at stage* $n - 1$.

*Let us draw a tree (see Figure 1):*

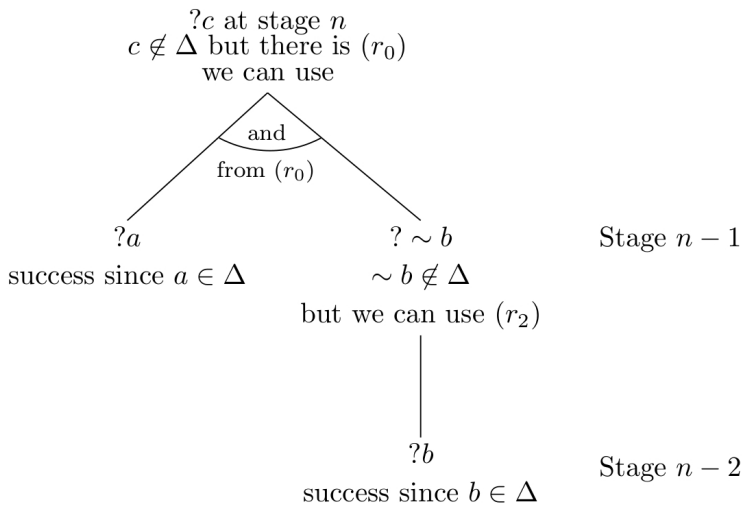


Figure 1:

*The tree shows success of the query ?c in both branches, one of depth 2 and one of depth 1, giving the impression that the query ?c should be successful. The backward computation tree, however, is wrong. We know it is wrong because if we go forward with the rules, we oscillate between* $\{a, b\}$ *and* $\{\sim a, \sim b\}$*, We never get* $\{a, \sim b\}$*, and so we never get c. So what did we do wrong? We must get all final success nodes all at the same stage. Here ?a succeeds at stage* $n-1$ *but ?b succeeds at stage* $n-2$*.*

### 5.2 Vulnerability checking case 1

Let us try a different policy.

1. Compute goal directed backwards until all branches succeed.
2. Check how many steps backwards you need for the longest path. Let it be $n$. (In our case it is $n = 2$).
3. Go forward $n+1$ steps and you either get the goal and succeed or alternatively you get inconsistency or failure or loop in which case you will know you will never get the goal.

So the backward computation serves to tell you how many steps to go forward to get a definite answer. This strategy, of course, yet has to be proved correct. This can be proved by induction.

**Definition 5.2 (Non-deterministic success tree)** *Let $\mathbb{P} = (\Delta, \mathbb{R})$ be a program and let $y$ be a literal. We define the notion of $y$* succeeds in at most *$n$-steps.*

1. *$y$ succeeds in one step if $y \in \Delta$*
2. *$y$ succeeds in exactly (respectively in at most) $n+1$ steps if for some rule $\Theta \Rightarrow y \in \mathbb{R}$. We have that for all $x \in \Theta, x$ succeeds in exactly (respectively in at most) $n$ steps.*
3. *$\mathbb{P}$ is inconsistent if for some $y$ and $\sim y$ both succeed in exactly $n$ steps.*

**Remark 5.3** *The previous definition gives us a non-deterministic goal directed algorithm for finding whether $\mathbb{P}$ is inconsistent. It is a conceptual definition. We need practical deterministic Algorithms.*

*We can approach it as follows:*

*Let $Q_0$ be a set of literals that we suspect are involved in an inconsistency. We need to find an $n$ and a $y$ in $Q_0$ such that $y$ and $\sim y$ both succeed in exactly $n$ steps.*

*Since we do not want to guess, we must for each $y$ and $\sim y$ in $Q_0$ find how many backward steps we need to succeed, take the maximal number of backward steps needed for any such $y$ or $\sim y$ to succeed and collect the set $Q_1$ containing $Q_0$, of all letters appearing in the computation tree of all $q$, from $Q_0$. We now go forward using only rules of the form $\Theta \Rightarrow y$ with $\Theta$ from $Q_0$. We then go forward up to $n+1$ steps to find if for some $m$, both $y$ and $\sim y$ are obtained at the same forward step.*

*The perceptive reader will understand that this algorithm is cheaper then just going forward with all the letters in $Q$. We go forward just with the letters which are suspect and a minimal number of required steps, suggested by the backward trees.*

*Note also that a $y$ can succeed in several $n$ steps. For example, let $\Delta = \{a\}$.*

$$\begin{aligned} (r_1): &\quad \{a\} \Rightarrow b \\ (r_2): &\quad \{b\} \Rightarrow a. \end{aligned}$$

*then $a$ can succeed in 1 step but also in 3 steps.*

*This is important because we may also have*

$$\begin{aligned} (r_3): &\quad \{a\} = c \\ (r_4): &\quad \{c\} \Rightarrow d \\ (r_5): &\quad \{a, d\} \Rightarrow y. \end{aligned}$$

*$y$ succeeds in 4 steps. For that we need $a$ to succeed in 3 steps.*

We now proceed with a series of concepts leading to the definition of decorated tree. Such trees are used in defining algorithms for finding inconsistencies and/or vulnerabilities. We need to overcome the problem of looping, and hence the need to include the history in the decoration.

Let us begin:

**Definition 5.4 (Trees)** *We define the notion of a finite tree $(T, <)$ of depth $\leq n$ (resp. exactly $n$) by induction on $n$. This tree describes the computation. The nodes of the tree represent steps in the computation and the pair $(x, y) \in <$ means that $x$ is a next step in the computation to $y$, and $y$ is the father of $x$. We have $< \subseteq T \times T$.*

1. *Let $T_\infty$ be an infinite set of distinct atoms which we use as nodes.*
2. *Let $x, y$ be nodes in the tree and assume $x < y$. We say that $x$ is an* immediate descendant *of $y$.*
3. *A tree of depth exactly one has the form*

$$T_1 = \{t\},\ < = \varnothing,\ t \in T_\infty$$

   *We say $t$ is the root of the tree as well as a bottom leaf of the tree.*

4. *Assume the notions of a tree of depth $\leq n$ and of depth exactly $n$ have been defined for $n \geq 1$. Let $(T^1, <^1), \ldots, (T^k, <^k)$ be $k$ trees of depth $\leq n$ (resp. depth exactly $n$) and assume that $T^j \cap T^i = \varnothing$, for $i \neq j$. Let $t \in T_\infty$ be a new point not appearing in any $T^j, j = 1, \ldots, k$ and let $t_i$ be the root of $(T^i, <^i)$. We define a new tree of depth $\leq n+1$ (resp. depth exactly $n+1$) as follows:*

$$\begin{array}{rcl} T & = & \{t\} \cup \bigcup_{i=1}^{k} T^i \\ < & = & \{(t_i, t) \mid i = 1, \ldots, k\} \cup \bigcup_{i=1}^{k} <^i \end{array}$$

5. *We say $t$ is the root of the new tree and its bottom leaves are all the bottom leaves of all the $T^i$.*
6. *Notice the trees are going downwards.*

**Definition 5.5 (History)**

1. *Let $\mathbb{P} = (\Delta, \mathbb{R})$ be a program. We define the notion of a sequence $H = ((x_1, r_1), (x_2, r_2), \ldots, (x_m, r_m))$ being a* legitimate history *for $\mathbb{P}$, where the pairs $(x_i, r_i)$ are comprised of a literal $x_i$ which is a goal and $r_i$ is a rule with head $x_i$ used to get $x_i$ in the respective step.*

(a) *Any unit sequence* $((y, \varnothing))$ *or the empty sequence* $\varnothing$ *is a legitimate history, for* $y$ *a literal.*

(b) *A sequence* $H = ((x_1, r_1), \ldots, (x_n, r_n)), n \geq 2$ *is a legitimate history if* $((x_1, r_1), \ldots, (x_{n-1}, r_{n-1}))$ *is a legitimate history and for some* $r : \Theta \Rightarrow x_{n-1}$ *in* $\mathbb{R}$ *we have that* $x_n \in \Theta$ *and* $r_n = r$.

**Definition 5.6 (Decorated tree)** *A decorated tree has the form* $(T, <, \delta)$, *where* $(T, <)$ *is a tree and* $\delta$ *is a function giving for each* $t \in T$ *a decoration of the form*

$$\delta(t) = (y, H, Z, A)$$

*where* $t$ *is a node in the tree indicating a step in the computation. At this step* $t$ *we want to get the goal/literal* $y$. $y$ *is a literal,* $H$ *is a legitimate history and* $Z \in \{success, failure, loop\}$ *and* $A$ *is a set of literals (the literal Abductive Set). The set* $A$ *is the set of literals we would need in order to succeed when the computation reaches the end of the tree. Of course we do not know* $A$ *at this stage but we would know* $A$ *when we reach the end of the tree and identify the nodes descendants of* $t$ *which fail or loops because of the literals decorating them at the end nodes. So to construct* $A$ *in practice we have to first construct the tree without instantiating the* $A$ *decoration and then move up the tree and give the accumulation of values to the* $A$ *decoration.*

**Remark 5.7 (Explaining the decoration)** *The trees are supposed to be computation trees for queries of the form* $\mathbb{P} = (\Delta, \mathbb{R})?y$.

*So the decoration* $\delta(t)$ *records* $y$, *the current query at* $t$. *It also records the history of the queries up to* $t$ *and what rules were used at each node above* $t$. *This is* $H$. *It also records whether the computation tree eventually succeeds or fails or loops. If it succeeds, the abduction set is* $\varnothing$. *If it fails or loops the abductive set* $A$ *tells us what literals are missing from* $\Delta$, *which are needed to make the query succeed. To be able to have an* $A$ *we need the tree to be of exactly depth* $n$. *We need* $H$ *also to know if we are looping. We use the history to terminate the tree and make it finite.*

*Look at the Example 5.1 to appreciate how we might loop.*

*The tree of Figure 2 describes the goal directed computation for the goal query* $?c$ *for this example: we record the history*

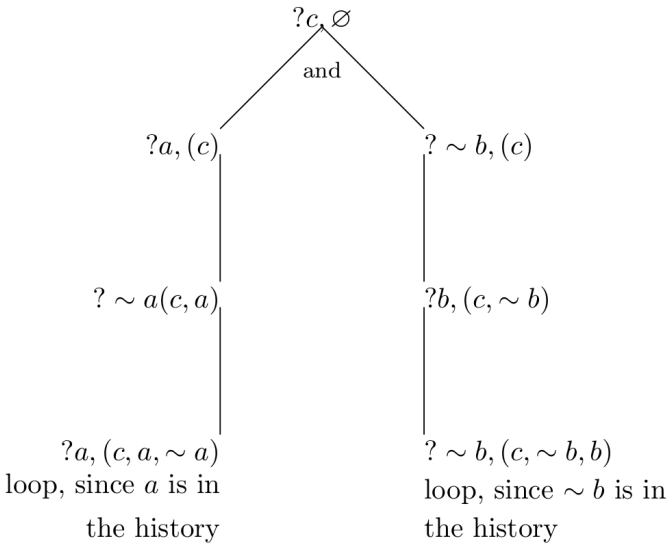


Figure 2:

*We omitted to record the rules used to keep the history simple and also because each goal can use exactly one rule.*

*Note that the tree is of exact depth 4 and so we can find an abductive set. We need all end leaves to immediately succeed. So that we need both $a$ and $\sim b$ to be in the database. So $A = \{a, \sim b\}$. The original database was $\Delta = \{a, b\}$. So we cannot just add $\sim b$, it will be inconsistent. So let us delete $b$ and add $\sim b$. So our database is now $\Delta_A = \{a, \sim b\}$. Would ?c succeed? Yes, immediately from the rule $(r_0) : \{a, \sim b\} \Rightarrow c$.*

Now that we have the concept of decorated tree, we can define the algorithm looking for inconsistencies and/or vulnerabilities.

**Definition 5.8 (Abduction for trees of depth 1)** *Let $\mathbb{P} = (\Delta, \mathbb{R})$ be a program and let $y$ be a literal. We define the set of decorated computation trees with success/failure/loop, for $\mathbb{P}?y$ together with their literal abductive sets $A$ and history $H$.*

**Case 1.** $y \in \Delta$.
*We have a tree of depth 1 of the form $(\{t\}, \varnothing), t \in T_\infty$, with the decoration $\delta(t) = (y, H,\ success, \varnothing)$. $\mathbb{P}?y$ succeeds, since $y \in \Delta$. There is no abductive set, but there is history and we may even have $y$ appears in $H$.*

*This is a case of success and was used to define the non-deterministic trees of success in Definition 5.2.*

*The abductive set is* $\varnothing$*. There is nothing to abduce. We mention this case for completeness. It is the fail and loop cases that are of interest for the construction of the abductive set. We do not need an abductive set in case of success.*

**Case 2.** $y \notin \Delta$ *and there is no rule in* $\mathbb{R}$ *with head* $y$ *(i.e. no rule of the form* $\Theta \Rightarrow y \in \mathbb{R}$*). So the tree is* $(\{t\}, \varnothing)$ *and the decoration is* $\delta(t) = (y, H,\ failure, \{y\})$*.* $\mathbb{P}?y$ *immediately fails, since* $y \notin \Delta$ *and there is no possibility of continuing the computation and searching for a success tree since there is no rule* $\Theta \Rightarrow y \in \mathbb{R}$*.*

*The abductive set is* $\{y\}$*, since this is the only way to make* $?y$ *succeed – to add* $y$ *to* $\Delta$*.*

**Case 3.** $y \notin \Delta$ *and there are rules in* $\mathbb{R}$ *with head* $y$*. The tree has one point* $(t, \varnothing)$ *with decoration* $\delta(t) = (y, H,\ loop, \{y\})$ *and we have that some* $z$ *appears in* $H$ twice *with the same rule, i.e.* $(z, \delta)$ *appears twice in the sequence* $H$*.*

*(To explain what this case does, consider the rule* $r : \{a\} \Rightarrow a$*. We ask* $?a$ *and we get the situation of Figure 3.)*

$?a, \varnothing$

$?a, ((a, r))$

$?a, ((a, r), (a, r))$, stop, loop

Figure 3:

*In this case there is no possibility of immediate success since* $y \notin \Delta$ *but also there is no certainty of immediate failure since we have* $k$ *rules we can use and may succeed. So our annotation would be: reserve judgement and continue the computation until one stops.*

**Remark 5.9 (Abduction continued, for the inductive case; trees of depth $n \geq 2$)** *We first explain this case. Consider the query $\mathbb{P}?y$ with $y \notin \Delta$ and there are some $\Theta_1 \Rightarrow y, \ldots, \Theta_k \Rightarrow y$ in $\mathbb{R}$ with head $y$.*

*We can non-deterministically choose one of the $\Theta_i \Rightarrow y$ and continue the computation and if we continue to choose favourably we will get a depth $n$ success tree according to Definition 5.2. Our tree will be annotated, but for this case of success we do not need the annotation.*

*The other possibility is that there is no success tree. This means that for* all *our non-deterministic choices, the tree will not be a success tree.*

*Of course we need to use the history as a loop checker to make the tree stop. There are two possibilities.*

1. *The tree is finite with depth $\leq n$ but* not *with depth exactly $n$. In this case there is no abductive set of literals. There is nothing to add to the database to make the original goal query succeed, except possibly to add the goal query itself.*

2. *The tree stops with depth exactly $n$. In this case the abduction set $A$ is the set of all queries in the bottom leaves. Some of the queries in the end leaves are not immediate success (that is why the original query is not a success). So these unsuccessful leaf queries either fail or loop. So if we add these queries literals to the database, we will have success. So we know what is the abduction set to add.*

*Note that for each such failed attempt of a tree of depth exactly $n$ we get a (probably different) abduction set. We are now ready for the inductive definition.*

**Definition 5.10 (Abductive set for the non-deterministic failure/loop annotated tree)** *Let $\mathbb{P} = (\Delta, \mathbb{R})$ be a program and $y$ be a literal. We define the notion of a* failure/loop tree *for $y$ from $\mathbb{P}$ in exactly $n$ steps. Note that we do not claim such a tree exists for $\mathbb{P}?y$. We are just going to say what it looks like if it exists.*

1. **Case $n = 1$.**
   *The definition as in Definition 5.8, case 2.*

2. **Case $n + 1$.**
   *This is the case where $y \notin \Delta$ but there are rules $\Theta_i \Rightarrow y$ in $\mathbb{R}$ with head $y$. Then our tree is built from a sequence of non-deterministic choices, the first one being choosing one of these rules, say $(r) : \{x_1, \ldots, x_n\} \Rightarrow y$ and continuing with choices promised by the inductive hypothesis of trees $T_n^i$ depth $n$ for the queries $\mathbb{P}?x_i$ as in Figure 4.*

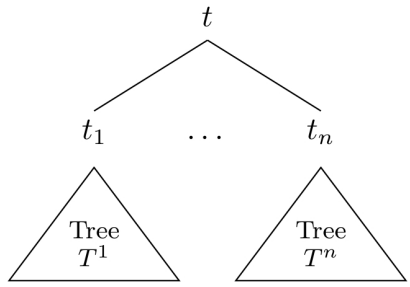


Figure 4:

$$\begin{aligned}\delta(t) &= (y, \varnothing, Z_t, A_t)\\ \delta(t_i) &= (x_i, (y, (r)), Z_{t_i}, A_{t_i})\end{aligned}$$

*where the trees with roots $t_i$ are all distinct and assumed to be correct and of depth $n$. For each node $s$ in any tree we have $A_s$ = set of all literal queries at the bottom leaves below $s$. Further, $Z(t)$ is failure iff the annotation of at least one leaf node is failure and $Z(t)$ = loop iff the annotation of at least one leaf is loop. Note that we have loop annotation if we have both failure leaves and loop leaves.*

**Remark 5.11** *Definition 5.8 and Remark 5.9 defined non-deterministic computation tree. We would like to define a complete single computation tree developing all computation options and addressing both disjunctive and conjunctive tree options. We explain through an example.*

*Consider the following database and rules.*

$$\begin{aligned}\Delta = &\quad \{a, b, x\}\\ (r_1) &\quad a \wedge b \Rightarrow c\\ (r_2) &\quad x \wedge y \Rightarrow c\end{aligned}$$

*our query is ?c.*

*A non-deterministic successful tree will choose rule $(r_1)$ and succeed. This option can be described by the tree of Figure 5*

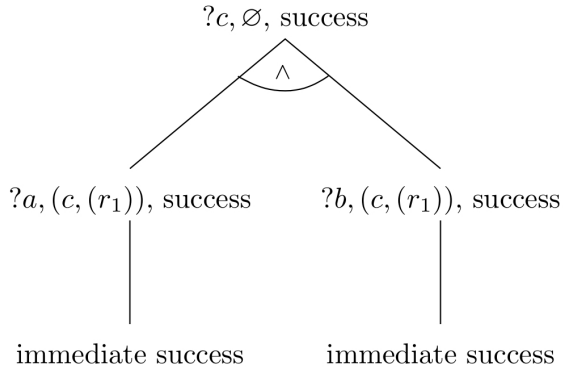


Figure 5:

*If we want a full deterministic tree showing all options including those which fail, we need to explore all the options in the same tree. So we get Figure 6. To build this tree properly we need to add two more annotations, "∨" and "∧" for the subtrees being disjunctive or conjunctive and we need also to record which rule is being used.*

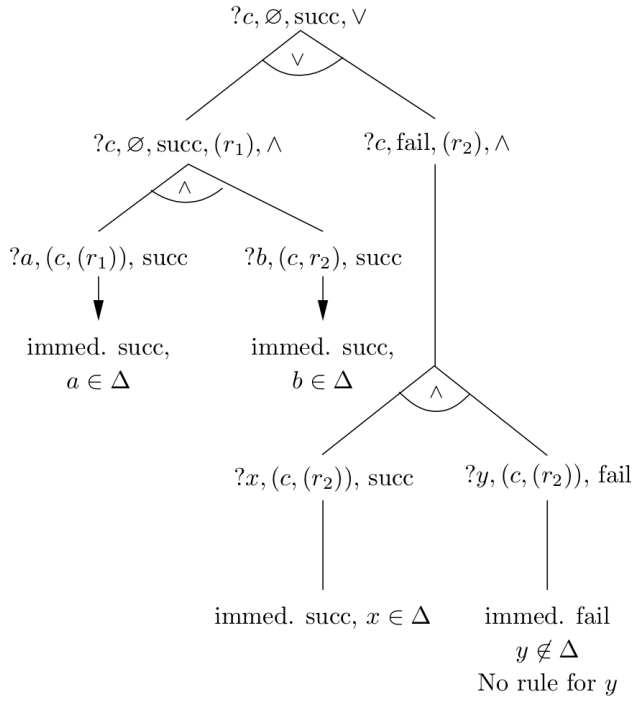


Figure 6:

**Definition 5.12 (Full deterministic computation tree with full annotation of nodes)** *The definition is by induction. We are given a database* $\Delta$*, rules* $\mathbb{R}$ *and we ask a goal* $?x$*. The induction is on the depth of annotated tree. We define the notion of a* correct computation tree*.*

1. *Tree with one node* $t$ *and annotation as follows is correct.*

$$t?x, H, Z, \text{ rule, boolean symbol}$$

   (a) $x \in \Delta, Z =$ *immediate success.*

   (b) $x \notin \Delta, Z =$ *immediate failure, and there are no rules in* $\Delta$ *of the form* $\Theta \Rightarrow x$*.*

   (c) $x \notin \Delta, Z =$ *loop, and for every rule* $(r) : \Theta \Rightarrow x$ *in* $\Delta$ *the pair* $(x, (r))$ *is in the history* $H$*.*

2. *Induction step, tree with depth* $n + 1$*.*

   (a) *The top node* $t$ *in the tree is annotated*

$$?x, H, Z, \text{ “}\vee\text{”}$$

   *(Note that there is no rule annotation) and* $x$ *is the head of rules* $(r_i) : \Theta_i \Rightarrow x, i = 1, \ldots, k$ *and the immediate descendant nodes of* $t$ *are* $s_1, \ldots, s_k$ *and these nodes are annotated* $?x, H_i, Z_i, (r_i)$*, and* $Z =$ *success (resp.* $Z =$ *failure,* $Z =$ *loop) and at least one* $Z_i$ *is success (resp. all* $Z_i =$ *failure, resp. one* $Z_i$ *at least is loop and all others are failure or loop) and the subtrees below* $s_i$ *are correct.*

   (b) *The top node* $t$ *is annotated* $x, H, Z, (r), \wedge$

$$(r) \text{ is the rule } x_1\wedge, \ldots, \wedge x_k \Rightarrow x$$

   *and the immediate descendants of* $t$ *are the nodes* $s_1, \ldots, s_k$ *and* $s_i$ *is decorated by* $x_i, H*(x, (r)), Z_i, \vee$ *and* $Z =$ *success (resp.* $Z =$ *failure, resp.* $Z =$ *loop) and all* $Z_i =$ *success (resp. some* $Z_i =$ *failure, resp. some* $Z_i =$ *loop and all other* $Z_i =$ *failure or loop) and the tree below* $s_i$ *is correct.*

**Theorem 5.13** *Let* $k$ *be a maximal length of any path in the tree of Definition 5.12 , for the goal* $?x$*. Then it takes* $k + 1$ *steps of forward execution to know for this literal* $x$ *whether it is in, out or looping.*

**Proof.** Obvious by induction on $k$. ■

**Remark 5.14** *Note that in Theorem 5.13, we will just know that after $k+1$ forward steps that the goal $?x$ is settled. So for example suppose $x$ gets into the data at step $k+1$. This does not guarantee that $x$ will stay there. It is quite possible that at a later step, say $k+3$, the goal $? \sim x$ is settles with $\sim x$ required to be in the data, and this will override $x$. In order to know exactly what is the status of $x$ Vs. $\sim x$ in with respect to the database we must apply Theorem 5.13 to each of $?x$ and $? \sim x$ and then go forward enough steps to settle both of them.*

**Remark 5.15** *Following the previous Remark 5.14, we note that for the purpose of vulnerability, we are interested only to find the steps where $?x$ is implemented, in case $x$ is a vulnerable node of the system. For example if $x =$ window open, we do not want $?x$ to succeed at any point. It does not matter if the window closes at some later stage, the important point to us is that it does not open in the presence of other literals, for example when no one at home.*

*Note that in general we need to define vulnerability. It may be that we do not mind if the window opens briefly only, as there is not enough time to be vulnerable. So we need a module defining/identifying vulnerability for the program.*

## 6 Comparison with the Literature

Our system uses a temporal operator of the form

$$x_1 \wedge \ldots x_n \Rightarrow y$$

where $x_i$ are entries in a database and if they are indeed in the data, we proceed the next step and (impose and) put $y$ in the data. The question to be asked is how does this system compare with traditional temporal logic, as used for program verification.

If "$\rightarrow$" denotes classical implication and "$O$" denotes the next operator and "$\Box$" denotes the always operator (see [10] for a survey of temporal logic for computer science), then $x \Rightarrow y$, the action binary operator we used earlier in this paper, can be written descriptively (by describing what "$\Rightarrow$" actually does) as $\Box(x \rightarrow Oy)$.

The best way to give an orientation and explain the differences is to work on a simple example and illustrate how the different points of view manifest themselves on this example.

1. **The language**
   So let our language contain two atoms.

   $\mathbf{h}$ = it is too hot
   $\sim \mathbf{h}$ = it is not too hot
   $W$ = window is open
   $\sim W$ = window is closed

   We imagine a house in the very cold north with a sensor $\mathbf{h}$ which determines the temperature every 10 minutes and updates the value $\{0 =$ not too hot and $1 =$ too hot$\}$, of $\mathbf{h}$.

2. **The program**
   Based on the value of $\mathbf{h}$ a program $\mathbb{P}$ controls a window. If at step $n$ (i.e. $10 \times n$ minutes from the start of the program) we have the sensor reporting that it is too hot, then the program opens the window. If the sensor reports that it is not too hot then the program closes the window.

3. **The rules**
   Let us write the rules for this program.

   $(r1)$: $\mathbf{h} \wedge \sim W \Rightarrow W$
   $(r2)$: $\mathbf{h} \wedge W \Rightarrow W$
   $(r3)$: $\sim \mathbf{h} \wedge W \Rightarrow \sim W$
   $(r4)$: $\sim \mathbf{h} \wedge \sim W \Rightarrow \sim W$

   Since the operational meaning of these rules is to impose action, we can write only two rules:

   $(rr1)$: $\mathbf{h} \Rightarrow W$
   $(rr2)$: $\sim \mathbf{h} \Rightarrow \sim W$

   It does not matter what the state of the window is at state $n$. At state $(n + 1)$ we impose the consequent of the rule. We close the widow or open the window as required.

   These rules are not descriptive, they are temporal action rules.

4. **The automaton**
   We now show that we can look at this system as specifying at automaton $\mathcal{A}$.

   The automaton gets input from the language $\mathbf{L}(\mathbf{h})$ which are numbers in $\{0, 1\}$. The automaton states are $\{W, \sim W\}$. Let us say the initial state is $W$. Figure 7 shows the transition system, corresponding to rules $(r1)$–$(r4)$.

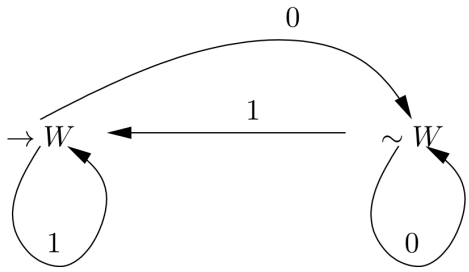


Figure 7:

Note that this is a finite automaton. There is no final state. The input is infinite stream. We need not give, for our purpose, a condition of acceptance for such streams.

5. **Sample runs**
Let us give a sample run for this automaton. Assume an infinite input vector of "heat" beginning with "0" and then keeping the rest of the values constant at "1", i.e. the vector $(0, 1, 1, \ldots)$ or $(\sim \mathbf{h}, \mathbf{h}, \mathbf{h}, \ldots)$, where 0 means "not too hot" and 1 means "too hot".

The corresponding sequence of states is $(W, \sim W, W, W, \ldots)$.

Figure 8 describes this scenario.

| Time | Time 1 | Time 2 | Time 3 | Time 4 |
|---|---|---|---|---|
| Input 1 =heat ? | $\sim \mathbf{h}$ | $\mathbf{h}$ | $\mathbf{h}$ | Continue as fixed $\mathbf{h}$ |
| States = window ? | $W$ | $\sim W$ | $W$ | Continue as fixed $W$ |

Figure 8:

We can also have the pair of vectors of Figure 9

| Time | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| heat | $\sim \mathbf{h}$ | $\mathbf{h}$ | $\mathbf{h}$ | ... |
| window | $W$ | $W$ | $W$ | ... |

Figure 9:

The sample run of Figure 9 does not satisfy the specification. At time 1 it was not hot ad yet the window was not closed at time 2.

Imagine we gave the specification to a student to implement in Java. The student came up with a Java program $J\mathbb{P}$ for our specification. We

give the package to a temporal logic verification company, who discovers a possible run (of Figure 9) which does not satisfy the specification, namely $\Box(\sim \mathbf{h} \rightarrow O \sim W)$ (meaning "always if it is not too hot then next the window is closed") does not hold in this run. What does this mean?

6. **Temporal logic**
We have atoms like $\mathbf{h}$ and $W$ and we have the logical connectives of classical logic, say $\{\neg, \wedge, \vee, \rightarrow\}$ and temporal future connectives $\{O, \Box\}$, say *next* $O$ and *always* $\Box$. We can have infinite runs, which we can describe as a function $\rho$ which gives for each $n$, values $\rho(W, n) \in \{0, 1\}$ and $\rho(\mathbf{h}, n) \in \{0, 1\}$. We define satisfaction $\rho \vDash_n A$, for any wff $A$ as follows:

For the atoms $\mathbf{h}$ and $W$, we have:
$\rho \vDash_n \mathbf{h}$ if $\rho(\mathbf{h}, n) = 1$
$\rho \vDash_n W$ if $\rho(Wn) = 1$.

For the complex formulas we have:
$\rho \vDash_n \neg A$ if $\rho \nvDash_n A$
$\rho \vDash_n A \wedge B$ if $\rho \vDash_n A$ and $\rho \vDash_n B$
$\rho \vDash_n A \vee B$ if $\rho \vDash_n A$ or $\rho \vDash_n B$
$\rho \vDash_n A \rightarrow B$ if $\rho \nvDash_n A$ or $\rho \vDash_n B$
$\rho \vDash_n OA$ if $\rho \vDash_{n+1} A$
$\rho \vDash_n \Box A$ if for all $m > n, \rho \vDash_m A$.

7. **Temporal logic verification**
To check and verify whether an implementation $J\mathbb{P}$ of a specification $\mathbb{P}$ satisfies the specification we need to take the following steps:

   (a) write IP into a program in temporal logic theory in the temporal logic language.
   (b) write a description of the runs of $J\mathbb{P}$ in temporal logic.
   (c) check whether (ii) satisfies (i) (or (ii) can prove (i)), or whatever method program verification uses.

8. **Comparison**
Note that $\mathbb{P}$ of our item (2) above is our program written in action temporal logic. It is not descriptive. The program $\mathbb{P}$ is

$(rr1)$: $\mathbf{h} \Rightarrow W$
$(rr2)$: $\sim \mathbf{h} \Rightarrow \sim W$

The temporal descriptive equivalent is

$$\Theta_{\mathbb{P}} = \Box(\mathbf{h} \rightarrow OW) \wedge \Box(\sim \mathbf{h} \rightarrow O \sim W).$$

Note that temporal logic expressions describe runs, they are not programs. So stress that the above formula is descriptive and it merely describes the runs of the action program.

9. **Security**
If the program is written in some programming language, to analyse what it does we look at its runs. In our case we look at the formula $\Theta_{\mathbb{P}}$. We define runs satisfying $\Theta_{\mathbb{P}}$ which are also suspicious. So if we have a run where the window is always open and it is always hot, i.e. $\varphi = \Box\mathbf{h} \wedge \Box W$, then something is wrong. Here we use knowledge of the world common sense. This is wrong because in cold weather with the window always open we should not have always too hot.

   So security using temporal logics of program verification examines suspicious run.

   In our case the program itself is written in temporal action logic, so we can examine $\mathbb{P}$ directly. We can ask, as we did in previous sections, what keeps the window open and apply our algorithms to discover that **h** can cause it. We can apply security measures to ensure **h** works correctly and not easily hackable.

10. **Other papers**
The following list of references [9, 10, 11], use temporal logic or other means to detect suspicious runs. They differ from our methods. Paper [10] is a survey of temporal logic and how it is used for program verification. It is included in the references for the benefit of the reader. The other two papers, [9] and [11], are sample papers showing how they approach the subject of security by looking at traces. We first quote from [9] (references omitted):

> "The theory of trace properties, which characterizes correct behavior of programs in terms of properties of individual execution paths, developed out of an interest in proving the correctness of programs. Practical model- checking tools, now enable automated verification of correctness. Verification of security, unfortunately, isn't directly possible with such tools, because some important security policies require sets of execution paths to model. But there is reason to believe that similar verification methodologies could be developed for security: The self-composition construction reduces properties of pairs of execution paths to properties of single execution paths, thereby enabling verification of a class of security policies.

> The theory of hyperproperties generalizes the theory of trace properties to security policies, showing that certain classes of security policies are amenable to verification with invariance arguments and with stepwise refinement. Prompted by these ideas, this paper develops an automated verification methodology for security. In our methodology, security policies are expressed as logical formulas, and a model checker verifies those formulas."

We now quote from [11] (references omitted):

> "Despite great progress in research on computer security, fully secure computer systems are still a distant dream. Today any large and complex computer system has many security flaws. Intrusion detection involves monitoring the system under concern to identify the misuse of these flaws as early as possible in order to take corrective measures.
>
> There are two main approaches to intrusion detection: signature-based and anomaly-based. In the signature-based approach, system behavior is observed for known patterns of attacks, while in the anomaly-based approach an alarm is raised if an observed behavior deviates significantly from pre-learned normal behavior. Both these approaches have relative advantages and disadvantages. The signature-based approach has a low false-alarm rate, but it requires us to know the patterns of security attacks in advance and previously unknown attacks would go undetected. In contrast, the anomaly-based approach can detect new attacks, but has a high false-alarm rate.
>
> In this paper, we adopt a temporal logic approach to signature-based intrusion detection. One can naturally specify the absence of a known attack pattern as a safety formula $\phi$ in a suitable temporal logic. Such a temporal logic based approach was considered using a variant of linear temporal logic (LTL) with first order variables. However we consider a more expressive logic in which one can also express attack signatures involving real-time constraints and statistical properties. We show how to automatically monitor the specification $\phi$ against the system execution and raise an intrusion alarm whenever the specification is violated. We also show how this technique can be used for simple types of anomaly-based intrusion detection. The idea is to specify the intended behavior of security-critical programs as temporal formulas involving statistical predicates,

and monitor the system execution to check if it violates the formula. If the observed execution violates the formula then an intrusion has occurred, and thus attacks can be detected even if they are previously unknown."

# 7 Conclusion

Today's IT security ecosystem, which relies on a combination of static perimeter network defence (e.g., firewalls) the use of end-host based defences (e.g. antivirus), and software patches from vendors, is fundamentally inadequate to handle IoT deployments. Against this background in this paper we have tried to face some of the challenges that the emerging IoT devices encountered. Specifically, here we defined a novel logic model to suit the special IoT characteristics and we developed novel techniques for security verification of an IoT system that mainly involves inconsistency and vulnerability.

We now proceed, describing our future research directions.

1. Notice that our consideration of the algorithms so far assume a unique initial state $\Delta$. In practice we may not know what the initial state is as this may depend on the users or circumstances. We therefore need to apply our method for each allowable initial state. However, in order to avoid explosive computation we may restrict the set of allowable initial state.

2. Our language contained classical literals $a, b, c$, their negations $\sim a, \sim b, \sim c$ and the failure symbol $\neg$.

   Our action symbols used only the current state and told us what to do for the next state. So if we wrote a rule of the form $W \Rightarrow \sim W$, it means if the window is open then in the next step close it. What if we want to say if the window is open now and at the previous step then close it? We have no memory of previous steps once we update. So we need a *yesterday operator symbol* $Ya$, reading '$a$' was in the data in the previous yesterday step.

   We can now form more atomic statements like $Ya, YYa, YYYa$,

   We can now write rules of the form $W \wedge YW \Rightarrow \sim W$, reading if the window has been opened for the two last steps, then close it.

3. Using First Order Logic (FOL) language to enable more complicated rules in the data base: we can allow for quantifiers (All $x$) and (Some $x$) and variables $x, y$ ranging over facts $a, b, c, \sim a, \sim b, \sim c, \ldots$ (Remember that $\sim\sim x = x$.)

We can write rules: (some $x$) (some $y$) $[x \wedge Y \sim x \wedge y \wedge Y \sim y) \Rightarrow$ call supervisor this is a safety rule, if too many literals (in this case 2 literals) change (indicated by the Yesterday Operator) then something is wrong.

From the implementation point of view we need to pay attention and remember two data steps, if we use only one iteration of $Y$ or more.

4. We can add a contradiction seeking module. In an intelligent home, there may be many rules. We can now detect a contradiction only if we are required to execute $a$ and $\sim a$. But what if we need to execute $a$ and $b$, where $a$ and $b$ are different but physically impossible facts? For example, in my home I cannot switch at the same time both the toaster and the kettle, it requires too much electrical current on the circuit. So in the morning, I want toast and tea, I need to do the toast first and then the tea. Sometimes I forget and discovered that the electricity is blown after 3 time ticks, not immediately. Anyway we need to record that: $a =$ toaster on and $b =$ kettle on, are contradictory. This means we need to use a classical logic proof theoretical module for detecting inconsistencies, see (4).

   This needs to be incorporated into our algorithms. We cannot avoid this and write say $a \Rightarrow \sim b$ because the two sides of $\Rightarrow$ are at different steps/times.

5. The contradictions can be described, implemented and identified within the framework of ABA — Assumption Based Argumentation [12]. This framework basically adds another implication symbol to the system (classical implication). We can write for example, $a \rightarrow\sim b$.

   Referring to the example in (3) of the toaster and kettle, we can now write $a \rightarrow \sim b$ meaning if the toaster is "on" then the kettle should not be "on" at the same time.

## Acknowledgement

This research is funded by the Israeli Ministry of science Technology and Space.

## References

[1] Alain Beauvieux. A method to check database consistency. In K Nori and S Kumar, eds. *Foundations of Software Technology and Theoretical Computer Science*, Springer 1988, pp 455-469.

[2] Gartner. Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020 http://www.gartner.com/newsroom/id/2636073

[3] https://www.alljoyn.org/

[4] HP. Internet of Things Security: State of the Union, 2014. http://www.hp.com/go/fortifyresearch/iot

[5] http://www.electronicproducts.com/Computer_Peripherals/Communication_Peripherals/AES_vs_SSL_TLS_Encryption_for_the_internet_of_things.aspx

[6] Liang, Chieh-Jan Mike and Karlsson, Börje F and Lane, Nicholas D and Zhao, Feng and Zhang, Junbei and Pan, Zheyi and Li, Zhao and Yu, Yong. SIFT: building an internet of safe things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, ACM, p. 298–309, 2015.

[7] http://threadgroup/org.

[8] Yu, Tianlong and Sekar, Vyas and Seshan, Srinivasan and Agarwal, Yuvraj and Xu, Chenren. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-Things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, ACM, p. 5, 2015.

[9] Masoud Koleini, Michael R. Clarkson and Kristopher K. Micinski. A temporal logic of security. https://arxiv.org/abs/1306.5678

[10] Stéphane Demri and Paul Gastin. Specification and Verification using Temporal Logics. *World Scientific Review Volume*. Chapter 15 in *Modern Applications Of Automata Theory*, Paperback 24 May 2012.

by Deepak D'souza (Editor), Priti Shankar (Editor), pp 457-495

[11] Prasad Naldurg, Koushik Sen, and Prasanna Thati. A Temporal Logic Based Framework for Intrusion Detection. In *Proceedings of the 24th IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, 2004.

[12] F. Toni. A tutorial on assumption-based argumentation, *Argument and Computation*, Vol: 5, 89–117, 2014.

[13] M.-C. Fiazza, M. Peroli and L. Viganò. Defending vulnerable security protocols by means of attack interference in non-collaborative scenarios. *ICT* 2:11, 2015. doi: 10.3389/fict.2015.00011

[14] Ruggero Lanotte, Massimo Merro, Riccardo Muradore, and Luca Vigano. A Formal Approach to Cyber-Physical Attacks. *arXiv*:1611.01377v2 [cs.CR], 2016.

Esther David
Department of Computer Science
Ashkelon Academic college
Ashkelon, Israel
*E-mail:* astrdod@acad.ash-college.ac.il

Dov Gabbay
Department of Computer Science, Faculty of Exact Sciences
Bar-Ilan University
Ramat-Gan, Israel
*E-mail:* dov.gabbay@kcl.ac.uk

Guy Leshem
Department of Computer Science
Ashkelon Academic college
Ashkelon, Israel
*E-mail:* leschemg@cs.bgu.ac.il

Students of CS Ashkelon
Department of Computer Science
Ashkelon Academic college
Ashkelon, Israel