Responsibility Analysis by Abstract Interpretation

Chaoqiang Deng and Patrick Cousot

Computer Science Department, New York University, USA
{deng,pcousot}@cs.nyu.edu

Abstract. Given a behavior of interest in the program, statically determining the corresponding responsible entity is a task of critical importance, especially in program security. Classical static analysis techniques (e.g. dependency analysis, taint analysis, slicing, etc.) assist programmers in narrowing down the scope of responsibility, but none of them can explicitly identify the responsible entity. Meanwhile, the causality analysis is generally not pertinent for analyzing programs, and the structural equations model (SEM) of actual causality misses some information inherent in programs, making its analysis on programs imprecise. In this paper, a novel definition of responsibility based on the abstraction of event trace semantics is proposed, which can be applied in program security and other scientific fields. Briefly speaking, an entity E_{R} is responsible for behavior $\mathcal{B}$, if and only if E_{R} is free to choose its input value, and such a choice is the first one that ensures the occurrence of $\mathcal{B}$ in the forthcoming execution. Compared to current analysis methods, the responsibility analysis is more precise. In addition, our definition of responsibility takes into account the cognizance of the observer, which, to the best of our knowledge, is a new innovative idea in program analysis.

Keywords: Responsibility · Abstract Interpretation · Static Analysis · Dependency · Causality · Program Security

1 Introduction

For any behavior of interest, especially potentially insecure behaviors in the program, it is essential to determine the corresponding responsible entity, or say, the root cause. Contrary to accountability mechanisms [40,23,15] that track down perpetrators after the fact, the goal of this paper is to detect the responsible entity and configure its permission before deploying the program, which is important for safety and security critical systems. Due to the massive scale of modern software, it is virtually impossible to identify the responsible entity manually. The only solution is to design a static analysis of responsibility, which can examine all possible executions of a program without executing them.

The cornerstone of designing such an analysis is to define responsibility in programming languages. It is surprising to notice that, although the concepts of causality and responsibility have been long studied in various contexts (law

sciences [36], artificial intelligence [30], statistical and quantum mechanics, biology, social sciences, etc. [4]), none of these definitions is fully pertinent for programming languages. Take the actual cause [19,20] as an example, its structural equations model (SEM) [10] is not suitable for representing programs: the value of each endogenous variable in the model is fixed once it is set by the equations or some external action, while the value of program variables can be assigned for unbounded number of times during the execution. In addition, the SEM cannot make use of the temporal information or whether an entity is free to make choices, which plays an indispensable role in determining responsibility.

There do exist techniques analyzing the influence relationships in programs, such as dependency analysis [1,7,37], taint analysis [31] and program slicing [39], which help in narrowing down the scope of possible locations of responsible entity. However, no matter whether adopting semantic or syntactic methods, these techniques are not precise enough to explicitly identify responsibility.

To solve the above problems, we propose a novel definition of responsibility based on the event trace semantics, which is expressive and generic to handle computer programs and other scientific fields. Roughly speaking, an entity E_{R} is responsible for a given behavior $\mathcal{B}$ in a certain trace, if and only if E_{R} can choose various values at its discretion (e.g. inputs from external subjects), and such a choice is the first one that guarantees the occurrence of $\mathcal{B}$ in that trace. Such a definition of responsibility is an abstract interpretation [11,12] of event trace semantics, taking into account both the temporal ordering of events and the information regarding whether an entity is free to choose its value. Moreover, an innovative idea of cognizance is adopted in this definition, which allows analyzing responsibility from the perspective of various observers. Compared to current techniques, our definition of responsibility is more generic and precise.

The applications of responsibility analysis are pervasive. Although an implementation of responsibility analyzer is not provided here, we have demonstrated its effectiveness by examples including access control, “negative balance” and information leakage. In addition, due to the page limit, a sound framework of abstract responsibility analysis is sketched in the extended version of this paper [13], which is the basis of implementing a responsibility analyzer. It is guaranteed that the entities that are found definitely responsible in the abstract analysis are definitely responsible in the concrete, while those not found potentially responsible in the abstract analysis are definitely not responsible in the concrete.

To summarize, the main contributions of this work are: (1) a completely new definition of responsibility, which is based on the abstract interpretation of event trace semantics, (2) the adoption of observers’ cognizance in program analysis for the first time, (3) various examples of responsibility analysis, and (4) a sound framework for the abstract static analysis of responsibility.

In the following, section 2 discusses the distinctions between responsibility and current techniques via an example, and sketches the framework of responsibility analysis. Section 3 formally defines responsibility as an abstraction of event trace semantics. Section 4 exemplifies the applications of responsibility analysis. Section 5 summarizes the related work.

2 A Glance at Responsibility

Given a behavior of interest (e.g. security policy violation), the objective of responsibility analysis is to automatically determine which entity in the system has the primary control over that behavior. Then security admins could decide either to keep or to deny the responsible entity's permission to perform the behavior of interest. Take the information leakage in a social network as an example: if the information's owner is responsible for the leakage (e.g. a user shares his picture with friends), then it is safe to keep its permission to perform such a behavior; otherwise, if anyone else is responsible for the leakage, it could be a malicious attacker and its permission to do so shall be removed. Such human decisions can only be done manually and are beyond the scope of this paper. In addition, it is worthwhile to note that responsibility analysis is not the same as program debugging, since the analyzed program code is presumed to be unmodifiable and the only possible change is on the permissions granted to entities in the system.

In order to give an informal introduction to responsibility, as well as its main distinctions with dependency, causality and other techniques in detecting causes, this section starts with a simple example, which is used throughout the paper.

2.1 Discussion of an Access Control Program Example

Example 1 (Access Control). Consider the program in Fig. 1, which essentially can be interpreted as an access control program for an object o (e.g. a secret file), such that o can be read if and only if both two admins approve the access and the permission type of o from system settings is greater than or equal to "read": the first two inputs correspond to the decisions of two independent admins, where 1 represents approving the access to o, and 0 represents rejecting the access; the third input stored in `typ` represents the permission type of o specified in the system settings, where 1 represents "read", 2 represents "read and write" (this is similar to the file permissions system in Linux, but is simplified for the sake of clarity); by checking the value of `acs` at line 10, the assertion can guarantee both admins approve the access and the permission type of o is at least 1. □

```
1:  apv = 1; //1: Approval, 0: Rejection
2:  i1 = input_1(); //Input 0 or 1 from 1st admin
3:  if (i1 == 0) {
4:    apv = 0; }
5:  i2 = input_2(); //Input 0 or 1 from 2nd admin
6:  if (apv != 0 && i2 == 0) {
7:    apv = 0; }
8:  typ = input_3(); //Input 1 or 2 from system settings
9:  acs = apv * typ;
10: assert(acs >= 1); //Check if the read access is granted
11: /* Read an object o here */
```

Fig. 1: Access Control Program Example

Here the question we are interested is: when the assertion fails (referred as "Read Failure" in the following, i.e. the read access to o fails to be granted), which entity (entities) in the program shall be responsible? The literature has several possible answers. By the definition of dependency ([1,7,37]), the value of `acs` depends on the value of `apv` and `typ`, which further depend on all three inputs. That is to say, the read failure depends on all variables in the program, thus program slicing techniques (both syntactic slicing [39] and semantic slicing [35]) would take the whole program as the slice related with read failure. Such a slice is useful in debugging in the sense that it rules out parts of the program that are completely irrelevant with the failure, and modifying any code left in the slice may prevent the failure, e.g. replacing "`acs=apv*typ`" with "`acs=2`" trivially fixes the read failure problem. However, this paper presumes the program code to be unmodifiable, hence a statement like "`acs=apv*typ`", which is fully controlled by others and acts merely as the intermediary between causes and effects, shall not be treated as responsible. In addition, the third input (i.e. the system setting of o's permission type) is also included in the slice. Although it does affect `acs`'s value, it is not decisive in this case (i.e. no matter it is 1 or 2, it could not either enforce or prevent the failure). Therefore, the dependency analysis and slicing are not precise enough for determining responsibility.

Causation by counterfactual dependency [28] examines the cause in every single execution and excludes non-decisive factors (e.g. the third input in this example), but it is too strong in some circumstances. For example, in an execution where both the first two inputs are 0, neither of them would be determined as the cause of read failure, because if one input is changed to value 1, the failure would still occur due to the other input 0.

Actual cause introduced in [19,20] is based on the structural equations model (SEM) [10], and extends the basic notion of counterfactual dependency to allow "contingent dependency". For this example here, it is straightforward to create a SEM to represent the access control program (although it is not always the case): three inputs are represented by exogenous variables, and five program variables are represented by endogenous variables, in which the value of `apv` is `i1*i2`. Consider an execution where both the first two inputs are 0, no matter what value the third input takes, the actual causes of read failure (i.e. `acs<1`) would be determined as "`i1=0`", "`i2=0`", "`apv=0`" and "`acs=0`", since the failure counterfactually depends on each of them under certain contingencies. Thus, both two admins are equally determined as causes of failure, as well as two intermediary variables. This structural-model method has allowed for a great progress in causality analysis, and solved many problems of previous approaches. However, as an abstraction of concrete semantics, the SEM unnecessarily misses too much information, including the following three important points.

(P1) *Time (i.e. the temporal ordering of events) should be taken into account.* For example, the SEM does not keep the temporal ordering of first two inputs (i.e. the information that "`i1=0`" occurs before "`i2=0`" is missed), hence it determines both of them equally as the cause of assigning `0` to `apv`, further as the cause of read failure. However, in the actual execution where first two inputs are 0, the

first input already decides the value of `apv` before the second input is entered and the assignment at line 7 is not even executed, thus it is unnecessary to take the second input as a cause of assigning `0` to `apv` or the read failure. To deal with this difficulty, Pearl's solution is to modify the model and introduce new variables [6] to distinguish whether `apv` is assigned by `i1` or `i2`. However, a much simpler method is to keep the temporal ordering of events, such that only the first event that ensures the behavior of interest is counted as the cause. Therefore, in an execution where both the first two inputs are 0, the first input ensures the read failure before the second input is entered, hence only the first input is responsible for failure; meanwhile, in another execution where the first input is 1 and the second one is 0, the second input is the first and only one that ensures the failure hence shall take the responsibility.

(P2) *The cause must be free to make choices.* For example, `acs=0` is determined as an actual cause of read failure, based on the reasoning that if the endogenous variable `acs` in SEM is assigned a different value, say `2`, then the read failure would not have occurred. But such a reasoning ignores a simple fact that `acs` is not free to choose its value and acts merely as an intermediary between causes and effects. Thus, only entities that are free to make choices can possibly be causes, and they include but are not limited to user inputs, system settings, files read, parameters of procedures or modules, returned values of external functions, variable initialization, random number generations and the parallelism. To be more accurate, it is the external subject (who does the input, configures the system settings, etc.) that is free to make choices, but we say that entities like user inputs are free to make choices, as an abuse of language.

(P3) *It is necessary to specify "to whose cognizance / knowledge" when identifying the cause.* All the above reasoning on causality is implicitly based on an omniscient observer's cognizance (i.e. everything that occurred is known), yet it is non-trivial to consider the causality to the cognizance of a non-omniscient observer. Reconsider the access control program example, and suppose we adopt the cognizance of the second admin who is in charge of the second input. If she/he is aware that the first input is already 0, she/he would not be responsible for the failure; otherwise she/he does not know whether the first input is 0 or 1, then she/he is responsible for ensuring the occurrence of failure. In most cases, the cognizance of an omniscient observer will be adopted, but not always.

2.2 An Informal Definition of Responsibility

To take the above three points into account and build a more expressive framework, this paper proposes *responsibility*, whose informal definition is as follows.

Definition 1 (Responsibility, informally). *To the cognizance of an observer, the entity E_{R} is responsible for a behavior $\mathcal{B}$ of interest in a certain execution, if and only if, according to the observer's observation, E_{R} is free to choose its value, and such a choice is the first one that guarantees the occurrence of $\mathcal{B}$ in that execution.*

It is worth mentioning that, for the whole system whose semantics is a set of executions, there may exist more than one entities that are responsible for $\mathcal{B}$. Nevertheless, in every single execution where $\mathcal{B}$ occurs, there is only one entity that is responsible for $\mathcal{B}$. To decide which entity in an execution is responsible, the execution alone is not sufficient, and it is necessary to reason on the whole semantics to exhibit the entity's "free choice" and guarantee of $\mathcal{B}$. Thus, responsibility is not a trace property (neither safety nor liveness property).

To put such a definition into effect, our framework of responsibility analysis is designed as Fig. 2, which essentially consists of three components: (1) *System semantics*, i.e. the set of all possible executions, each of which can be analyzed individually. (2) *A lattice of system behaviors of interest*, which is ordered such that the stronger a behavior is, the lower is its position in the lattice. (3) *An observation function for each observer*, which maps every (probably unfinished) execution to a behavior in the lattice that is guaranteed to occur, even though such a behavior may have not occurred yet. These three components are formally defined in section 3, and their abstractions are sketched in [13].

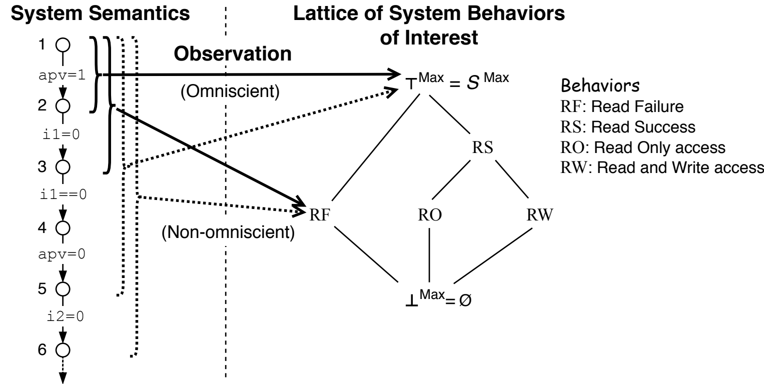

Fig. 2: Framework of Responsibility Analysis for Example 1

In this framework, if an observer's observation finds that the guaranteed behavior grows stronger after extending an execution, then the extension part of execution must be responsible for ensuring the occurrence of the stronger behavior. Consider the example in Fig. 2 which sketches the analysis for a certain execution of the access control program. Suppose $\top^{\mathsf{Max}}$ in the lattice represents "not sure if the read access fails or not" and `RF` represents the behavior of read failure, whose formal definitions are given in section 3.2. The solid arrow from executions to the lattice stands for the observation of an omniscient observer, while the dashed arrow stands for the observation of the second admin who is unaware of the first input. As illustrated in the figure, the omniscient observer finds that the execution from point 1 to point 2 can guarantee only $\top^{\mathsf{Max}}$, while the stronger behavior `RF` is guaranteed if the execution reaches point 3. Thus, to the cognizance of the omniscient observer, "`i1=0`" between point 2 and 3 is responsible for the read failure. Meanwhile, the second admin observes that all the executions upto point 5 guarantee $\top^{\mathsf{Max}}$, and `RF` is guaranteed only after point

6 is reached. Hence, to the cognizance of the second admin, "`i2=0`" between point 5 and point 6 is responsible for the read failure. For the sake of completeness, the entire desired analysis result for Example 1 is included in the following.

Example 2 (Access Control, Continued). To the cognizance of an omniscient observer: for any execution, if the first input is 0, no matter what the other two inputs are, only the first admin is responsible for the read failure; if the first input is 1 and the second one is 0, the second admin is responsible.

To the cognizance of the second admin, two cases need to be considered separately. If she/he is aware of the input of first admin, the analysis result is exactly the same as the omniscient observer. Otherwise, she/he does not know the first input: in every execution where the second input is 0, the second admin is responsible, no matter what the first and third input are; in every execution where the second input is 1, nobody shall be responsible for the failure, since whether the failure occurs or not is uncertain from the second admin's perspective. □

After finishing responsibility analysis, it is time for the security admin to configure permissions granted to each responsible entity at her/his discretion. If the behavior of interest is desired or the responsible entity is authorized, the permissions granted to the responsible entity can be kept. On the contrary, if that behavior is undesired or it is against the policy for the responsible entity to control it, the permissions granted to the responsible entity shall be confined. For instance, in the access control program, if the first two inputs are from admins who are authorized to control the access, their permissions to input 0 and 1 can be kept; if those two inputs come from ordinary users who have no authorization to deny other users' access, their permissions to input 0 shall be removed.

3 Formal Definition of Responsibility

In order to formalize the framework of responsibility analysis, this section introduces event traces to represent the system semantics, builds a lattice of system behaviors by trace properties, proposes an observation function that derives from the observer's cognizance and an inquiry function on system behaviors. Furthermore, this section formally defines responsibility as an abstraction of system semantics, using the observation function. To strengthen the intuition of responsibility analysis, the analysis of Example 1 will be illustrated step by step.

3.1 System Semantics

Generally speaking, no matter what system we are concerned with and no matter which programming language is used to implement that system, the system's semantics can be represented by event traces.

Event Trace In general, an *event* could be used to represent any action in the system, such as "input an integer", "assign a value to a variable", or even "launch the program". Take the classic While programming language as an example, there

are only three types of events: skip, assignment, and Boolean test. In order to make the definition of responsibility as generic as possible, here we do not adopt a specific programming language or restrict the range of possible events.

A *trace* σ is a sequence of events that represents an execution of the system, and its length $|\sigma|$ is the number of events in σ. If σ is infinite, then its length $|\sigma|$ is denoted as ∞. A special trace is the empty trace ε , whose length is 0. A trace σ is $\preceq$ - less than or equal to another trace σ', if and only if, σ is a prefix of σ'. The *concatenation* of a finite trace σ and an event e is simply defined by juxtaposition σe, and the *concatenation* of a finite traces σ and another (finite or infinite) trace σ' is denoted as $\sigma\sigma'$.

$$
\begin{array}{llll}
e \in \mathrm{E} & & & \text{event} \\
\sigma \in \mathrm{E}^{+\infty} & \triangleq & \bigcup_{n \geqslant 1} \{[0, n-1] \mapsto \mathrm{E}\} \cup \{\mathbb{N} \mapsto \mathrm{E}\} & \text{nonempty trace} \\
\sigma \in \mathrm{E}^{*\infty} & \triangleq & \{\varepsilon\} \cup \mathrm{E}^{+\infty} & \text{empty or nonempty trace} \\
\sigma \preceq \sigma' & \triangleq & |\sigma| \leqslant |\sigma'| \wedge \forall\, 0 \leqslant i \leqslant |\sigma| - 1 : \sigma_i = \sigma'_i & \text{prefix ordering of traces}
\end{array}
$$

The function $\mathsf{Pref}(P)$ returns the prefixes of every trace in the set P of traces.

$$
\begin{array}{ll}
\mathsf{Pref} \in \wp(\mathrm{E}^{*\infty}) \mapsto \wp(\mathrm{E}^{*\infty}) & \text{prefixes of traces} \\
\mathsf{Pref}(P) \triangleq \{\sigma' \in \mathrm{E}^{*\infty} \mid \exists \sigma \in P.\ \sigma' \preceq \sigma\} &
\end{array}
$$

Trace Semantics For any system that we are concerned with, its *maximal trace semantics*, denoted as $\mathcal{S}^{\mathsf{Max}} \in \wp(\mathrm{E}^{*\infty})$, is the set of all possible maximal traces of that system. Especially, the maximal trace semantics of an empty program is $\{\varepsilon\}$. Correspondingly, the *prefix trace semantics* $\mathcal{S}^{\mathsf{Pref}} \in \wp(\mathrm{E}^{*\infty})$ is the set of all possible prefix traces, which is an abstraction of maximal trace semantics via Pref, i.e. $\mathcal{S}^{\mathsf{Pref}} = \mathsf{Pref}(\mathcal{S}^{\mathsf{Max}})$. Besides, a trace σ is said to be *valid* in the system, if and only if $\sigma \in \mathcal{S}^{\mathsf{Pref}}$. Obviously, both maximal and prefix trace semantics do preserve the temporal ordering of events, which is missed by the SEM.

Example 3 (Access Control, Continued). For the program in Fig. 1, only two types of events are used: assignment (e.g. `apv=1`) and Boolean test (e.g. `i1==0` and `¬(acs>=1)`, where ¬ denotes the failure of a Boolean test). To clarify the boundary among events, the triangle ▷ is used in the following to separate events in the trace. The access control program has three inputs, each of which has two possible values, thus its maximal trace semantics $\mathcal{S}^{\mathsf{Max}}$ consists of 8 traces (`T1`-`T8`), each of which is represented as a path in Fig. 3 starting at the entry point of program and finishing at the exit point. E.g. `T1` = `apv=1` ▷ `i1=0` ▷ `i1==0` ▷ `apv=0` ▷ `i2=0` ▷ `¬(apv!=0&&i2==0)` ▷ `typ=1` ▷ `acs=0` ▷ `¬(acs>=1)` denotes the maximal execution where the first two inputs are 0 and the third input is 1. Meanwhile, the prefix trace semantics $\mathcal{S}^{\mathsf{Pref}} = \mathsf{Pref}(\mathcal{S}^{\mathsf{Max}})$ are represented by the paths that start at the entry point and stop at any point (including the entry point for the empty trace ε). □

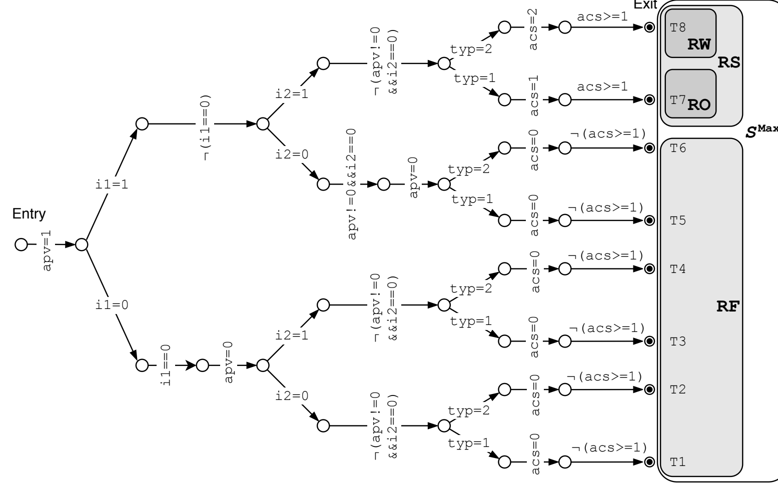

Fig. 3: Trace Semantics and Properties of Example 1

3.2 Lattice of System Behaviors of Interest

Trace Property A *trace property* is a set of traces in which a given property holds. Most behaviors of a given system, if not all, can be represented as a maximal trace property $\mathcal{P} \in \wp(\mathcal{S}^{\mathsf{Max}})$.

Example 4 (Access Control, Continued). As illustrated in Fig. 3, the behavior "Read Failure" `RF` is represented as a set of maximal traces such that the last event is `¬(acs>=1)`, i.e. $\texttt{RF} = \{\sigma \in \mathcal{S}^{\mathsf{Max}} \mid \sigma_{|\sigma|-1} = \neg\texttt{(acs>=1)}\} = \{\texttt{T1}, \texttt{T2}, \texttt{T3}, \texttt{T4}, \texttt{T5}, \texttt{T6}\}$; the behavior "Read Success" `RS` (i.e. the read access succeeds to be granted) is the complement of `RF`, i.e. $\texttt{RS} = \mathcal{S}^{\mathsf{Max}} \backslash \texttt{RF} = \{\texttt{T7}, \texttt{T8}\}$, whose subset $\texttt{RO} = \{\texttt{T7}\}$ and $\texttt{RW} = \{\texttt{T8}\}$ represent stronger properties "Read Only access is granted" and "Read and Write access is granted", respectively. ☐

Complete Lattice of Maximal Trace Properties of Interest We build a complete lattice of maximal trace properties, each of which represents a behavior of interest. Typically, such a lattice is of form $\langle \mathcal{L}^{\mathsf{Max}}, \subseteq, \top^{\mathsf{Max}}, \bot^{\mathsf{Max}}, \Cup, \Cap \rangle$, where

- $\mathcal{L}^{\mathsf{Max}} \in \wp(\wp(\mathrm{E}^{*\infty}))$ is a set of behaviors of interest, each of which is represented by a maximal trace property;
- $\top^{\mathsf{Max}} = \mathcal{S}^{\mathsf{Max}}$, i.e. the top is the weakest maximal trace property which holds in every valid maximal trace;
- $\bot^{\mathsf{Max}} = \emptyset$, i.e. the bottom is the strongest property such that no valid trace has this property, hence it is used to represent the property of invalidity;
- $\subseteq$ is the standard set inclusion operation;
- $\Cup$ and $\Cap$ are join and meet operations, which might not be the standard $\cup$ and $\cap$, since $\mathcal{L}^{\mathsf{Max}}$ is a subset of $\wp(\mathcal{S}^{\mathsf{Max}})$ but not necessarily a sublattice.

For any given system, there is possibly more than one way to build the complete lattice of maximal trace properties, depending on which behaviors are of interest. A special case of lattice is the power set of maximal trace semantics, i.e. $\mathcal{L}^{\mathsf{Max}} = \wp(\mathcal{S}^{\mathsf{Max}})$, which can be used to examine the responsibility for every possible behavior in the system. However, in most cases, a single behavior is of interest, and it is sufficient to adopt a lattice with only four elements: $\mathcal{B}$ representing the behavior of interest, $\mathcal{S}^{\mathsf{Max}} \setminus \mathcal{B}$ representing the complement of the behavior of interest, as well as the top $\mathcal{S}^{\mathsf{Max}}$ and bottom $\emptyset$. Particularly, if $\mathcal{B}$ is equal to $\mathcal{S}^{\mathsf{Max}}$, i.e. every valid maximal trace in the system has this behavior of interest, then a trivial lattice with only the top and bottom is built, from which no responsibility can be found, making the corresponding analysis futile.

Example 5 (Access Control, Continued). We assume that "Read Failure" is of interest, as well as the behavior of granting write access. As illustrated by the lattice in Fig. 2, regarding whether the read access fails or not, the top $\top^{\mathsf{Max}}$ is split into two properties "Read Failure" `RF` and "Read Success" `RS`, which are defined in Example 4 such that $\mathtt{RF} \cup \mathtt{RS} = \mathcal{S}^{\mathsf{Max}}$ and $\mathtt{RF} \cap \mathtt{RS} = \emptyset$. Furthermore, regarding whether the write access is granted or not, `RS` is split into "Read Only access is granted" `RO` and "Read and Write access is granted" `RW`. Now every property of interest corresponds to an element in the lattice, and the bottom $\perp^{\mathsf{Max}} = \emptyset$ is the meet $\cap$ of `RF`, `RO` and `RW`. In addition, if "Read Failure" is the only behavior of interest, `RO` and `RW` can be removed from the lattice. $\square$

Prediction Abstraction Although the maximal trace property is well-suited to represent system behaviors, it does not reveal the point along the maximal trace from which a property is guaranteed to hold later in the execution. Thus, we propose to abstract every maximal trace property $\mathcal{P} \in \mathcal{L}^{\mathsf{Max}}$ isomorphically into a set $\mathcal{Q}$ of prefixes of maximal traces in $\mathcal{P}$, excluding those whose maximal prolongation may not satisfy the property $\mathcal{P}$. This abstraction is called *prediction abstraction*, and $\mathcal{Q}$ is a *prediction trace property* corresponding to $\mathcal{P}$. It is easy to see that $\mathcal{Q}$ is a superset of $\mathcal{P}$, and is not necessarily prefix-closed.

$$\alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!] \in \wp(\mathrm{E}^{*\infty}) \mapsto \wp(\mathrm{E}^{*\infty}) \qquad \text{prediction abstraction}$$

$$\alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\mathcal{P}) \triangleq \{\sigma \in \mathsf{Pref}(\mathcal{P}) \mid \forall \sigma' \in \mathcal{S}^{\mathsf{Max}}.\ \sigma \preceq \sigma' \Rightarrow \sigma' \in \mathcal{P}\}$$

$$\gamma_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!] \in \wp(\mathrm{E}^{*\infty}) \mapsto \wp(\mathrm{E}^{*\infty}) \qquad \text{prediction concretization}$$

$$\gamma_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\mathcal{Q}) \triangleq \{\sigma \in \mathcal{Q} \mid \sigma \in \mathcal{S}^{\mathsf{Max}}\} = \mathcal{Q} \cap \mathcal{S}^{\mathsf{Max}}$$

We have a Galois isomorphism between maximal trace properties and prediction trace properties:

$$\langle \wp(\mathcal{S}^{\mathsf{Max}}),\ \subseteq \rangle \xleftrightarrows[\alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!]]{\gamma_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!]} \langle \bar{\alpha}_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\wp(\mathcal{S}^{\mathsf{Max}})),\ \subseteq \rangle \tag{1}$$

where the abstract domain is obtained by a function $\bar{\alpha}_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!] \in \wp(\wp(\mathrm{E}^{*\infty})) \mapsto \wp(\wp(\mathrm{E}^{*\infty}))$, which is defined as $\bar{\alpha}_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\mathfrak{X}) \triangleq \{\alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\mathcal{P}) \mid \mathcal{P} \in \mathfrak{X}\}$. The following lemma immediately follows from the definition of $\alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!]$.

Lemma 1. *Given a prediction trace property $\mathcal{Q}$ that corresponds to a maximal trace property $\mathcal{P}$, if a prefix trace σ belongs to $\mathcal{Q}$, then σ guarantees the satisfaction of property $\mathcal{P}$ (i.e. every valid maximal trace that is greater than or equal to σ is guaranteed to have property $\mathcal{P}$).*

Example 6 (Access Control, Continued). By α_{Pred}, each behavior in the lattice $\mathcal{L}^{\mathsf{Max}}$ of Example 5 can be abstracted into a prediction trace property:

- $\alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\top^{\mathsf{Max}}) = \mathcal{S}^{\mathsf{Pref}}$, i.e. every valid trace in $\mathcal{S}^{\mathsf{Pref}}$ guarantees $\top^{\mathsf{Max}}$.
- $\alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\texttt{RF}) = \{\sigma \in \mathcal{S}^{\mathsf{Pref}} \mid \texttt{apv=1} \triangleright \texttt{i1=0} \preceq \sigma \vee \texttt{apv=1} \triangleright \texttt{i1=1} \triangleright \neg\texttt{(i1==0)} \triangleright \texttt{i2=0} \preceq \sigma\ \}$, i.e. for any valid trace, if at least one of first two inputs is 0, then it guarantees "Read Failure" `RF`.
- $\alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\texttt{RS}) = \{\sigma \in \mathcal{S}^{\mathsf{Pref}} \mid \texttt{apv=1} \triangleright \texttt{i1=1} \triangleright \neg\texttt{(i1==0)} \triangleright \texttt{i2=1} \preceq \sigma\}$, i.e. for any valid trace, if first two inputs are 1, it guarantees "Read Success" `RS`.
- $\alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\texttt{RO}) = \{\sigma \in \mathcal{S}^{\mathsf{Pref}} \mid \texttt{apv=1} \triangleright \texttt{i1=1} \triangleright \neg\texttt{(i1==0)} \triangleright \texttt{i2=1} \triangleright \neg\texttt{(apv!=0\&\&i2==0)} \triangleright \texttt{typ=1} \preceq \sigma\}$, i.e. for any valid trace, if first two inputs are 1 and the third input is 1, then it guarantees "Read Only access is granted" `RO`.
- $\alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\texttt{RW}) = \{\sigma \in \mathcal{S}^{\mathsf{Pref}} \mid \texttt{apv=1} \triangleright \texttt{i1=1} \triangleright \neg\texttt{(i1==0)} \triangleright \texttt{i2=1} \triangleright \neg\texttt{(apv!=0\&\&i2==0)} \triangleright \texttt{typ=2} \preceq \sigma\ \}$, i.e. for any valid trace, if first two inputs are 1 and the third is 2, then it guarantees "Read and Write access is granted" `RW`.
- $\alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\bot^{\mathsf{Max}}) = \emptyset$, i.e. no valid trace can guarantee $\bot^{\mathsf{Max}}$. □

3.3 Observation of System Behaviors

Let $\mathcal{S}^{\mathsf{Max}}$ be the maximal trace semantics and $\mathcal{L}^{\mathsf{Max}}$ be the lattice of system behaviors designed as in Section 3.2. Given any prefix trace $\sigma \in \mathrm{E}^{*\infty}$, an observer can learn some information from it, more precisely, a maximal trace property $\mathcal{P} \in \mathcal{L}^{\mathsf{Max}}$ that is guaranteed by σ from the observer's perspective. In this section, an *observation* function $\mathbb{O}$ is proposed to represent such a "property learning process" of the observer, which is formally defined in the following three steps.

Inquiry Function First, an *inquiry function* $\mathbb{I}$ is defined to map every trace $\sigma \in \mathrm{E}^{*\infty}$ to the strongest maximal trace property in $\mathcal{L}^{\mathsf{Max}}$ that σ can guarantee.

$$\mathbb{I} \in \wp(\mathrm{E}^{*\infty}) \mapsto \wp(\wp(\mathrm{E}^{*\infty})) \mapsto \mathrm{E}^{*\infty} \mapsto \wp(\mathrm{E}^{*\infty}) \qquad \text{inquiry (2)}$$
$$\mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \sigma) \triangleq$$
$$\quad \textit{let } \alpha_{\mathsf{Pred}}[\![\mathcal{S}]\!](\mathcal{P}) = \{\sigma \in \mathsf{Pref}(\mathcal{P}) \mid \forall \sigma' \in \mathcal{S}.\ \sigma \preceq \sigma' \Rightarrow \sigma' \in \mathcal{P}\} \textit{ in}$$
$$\qquad \Cap\{\mathcal{P} \in \mathcal{L}^{\mathsf{Max}} \mid \sigma \in \alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\mathcal{P})\}$$

Specially, for an invalid trace $\sigma \notin \mathcal{S}^{\mathsf{Pref}}$, there does not exist any $\mathcal{P} \in \mathcal{L}^{\mathsf{Max}}$ such that $\sigma \in \alpha_{\mathsf{Pred}}[\![\mathcal{S}^{\mathsf{Max}}]\!](\mathcal{P})$, therefore $\mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \sigma) = \emptyset = \bot^{\mathsf{Max}}$.

Corollary 1. *Given the semantics $\mathcal{S}^{\mathsf{Max}}$ and lattice $\mathcal{L}^{\mathsf{Max}}$ of system behaviors, if the inquiry function $\mathbb{I}$ maps a trace σ to a maximal trace property $\mathcal{P} \in \mathcal{L}^{\mathsf{Max}}$, then σ guarantees the satisfaction of $\mathcal{P}$ (i.e. every valid maximal trace that is greater than or equal to σ is guaranteed to have property $\mathcal{P}$).*

Lemma 2. *The inquiry function* $\mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}})$ *is decreasing on the inquired trace* σ*: the greater (longer)* σ *is, the stronger property it can guarantee.*

Example 7 (Access Control, Continued). Using $\mathcal{S}^{\mathsf{Max}}$ defined in Example 3 and $\mathcal{L}^{\mathsf{Max}}$ defined in Example 5, the inquiry function $\mathbb{I}$ of definition (2) is such that:

- $\mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1}) = \top^{\mathsf{Max}}$, i.e. `apv=1` can guarantee only $\top^{\mathsf{Max}}$.
- $\mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1} \triangleright \texttt{i1=0}) = \texttt{RF}$, i.e. after setting the first input as 0, "Read Failure" `RF` is guaranteed.
- $\mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1} \triangleright \texttt{i1=1}) = \mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1}\triangleright\texttt{i1=1} \triangleright\neg\texttt{(i1==0)}) = \top^{\mathsf{Max}}$, i.e. if the first input is 1, only $\top^{\mathsf{Max}}$ is guaranteed before entering the second input.
- $\mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1} \triangleright \texttt{i1=1} \triangleright \neg\texttt{(i1==0)} \triangleright \texttt{i2=0}) = \texttt{RF}$, i.e. if the second input is 0, "Read Failure" `RF` is guaranteed.
- $\mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1} \triangleright \texttt{i1=1} \triangleright \neg\texttt{(i1==0)} \triangleright \texttt{i2=1}) = \texttt{RS}$, i.e. if first two inputs are 1, "Read Success" `RS` is guaranteed.
- $\mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1} \triangleright \texttt{i1=1} \triangleright \neg\texttt{(i1==0)} \triangleright \texttt{i2=1} \triangleright \neg\texttt{(i2==0)} \triangleright \texttt{typ=2}) = \texttt{RW}$, i.e. if first two inputs are 1, after the third input is set to be 2, a stronger property "Read and Write access is granted" `RW` is guaranteed. □

Cognizance Function As discussed in (P3) of section 2.1, it is necessary to take the observer's cognizance into account. Specifically, in program security, the observer's cognizance can be used to represent attackers' capabilities (e.g. what they can learn from the program execution). Given a trace σ (not necessarily valid), if the observer cannot distinguish σ from some other traces, then he does not have an omniscient cognizance of σ, and the *cognizance* function $\mathbb{C}(\sigma)$ is defined to include all traces indistinguishable from σ.

$$\mathbb{C} \in \mathrm{E}^{*\infty} \mapsto \wp(\mathrm{E}^{*\infty}) \qquad \text{cognizance (3)}$$
$$\mathbb{C}(\sigma) \triangleq \{\sigma' \in \mathrm{E}^{*\infty} \mid \text{observer cannot distinguish } \sigma' \text{ from } \sigma\}$$

Such a cognizance function is extensive, i.e. $\forall \sigma \in \mathrm{E}^{*\infty}.\ \sigma \in \mathbb{C}(\sigma)$. In particular, there is an *omniscient observer* and its corresponding cognizance function is denoted as $\mathbb{C}_o$ such that $\forall \sigma \in \mathrm{E}^{*\infty}.\ \mathbb{C}_o(\sigma) = \{\sigma\}$, which means that every trace is unambiguous to the omniscient observer.

To facilitate the proof of some desired properties for the observation function defined later, two assumptions are made here without loss of generality:

(A1) The cognizance of a trace $\sigma\sigma'$ is the concatenation of cognizances of σ and σ'. I.e. $\forall \sigma, \sigma' \in \mathrm{E}^{*\infty}.\ \mathbb{C}(\sigma\sigma') = \{\tau\tau' \mid \tau \in \mathbb{C}(\sigma) \wedge \tau' \in \mathbb{C}(\sigma')\}$.

(A2) Given an invalid trace, the cognizance function would not return a valid trace. I.e. $\forall \sigma \in \mathrm{E}^{*\infty}.\ \sigma \notin \mathcal{S}^{\mathsf{Pref}} \Rightarrow \mathbb{C}(\sigma) \cap \mathcal{S}^{\mathsf{Pref}} = \emptyset$.

To make the assumption (A1) sound, we must have $\mathbb{C}(\varepsilon) = \{\varepsilon\}$, because otherwise, for any non-empty trace σ, $\mathbb{C}(\sigma) = \mathbb{C}(\sigma\varepsilon) = \{\tau\tau' \mid \tau \in \mathbb{C}(\sigma) \wedge \tau' \in \mathbb{C}(\varepsilon)\}$ does not have a fixpoint. In practice, $\{\langle\sigma, \sigma'\rangle \mid \sigma' \in \mathbb{C}(\sigma)\}$ is an equivalence relation, but the symmetry and transitivity property are not used in the proofs.

Example 8 (Access Control, Continued). Consider two separate observers.

(i) For an omniscient observer: $\forall \sigma \in \mathrm{E}^{*\infty}.\ \mathbb{C}_o(\sigma) = \{\sigma\}$.

(ii) For an observer representing the second admin who is unaware of the first input: $\mathbb{C}(\texttt{i1=0} \triangleright \texttt{i1==0} \triangleright \texttt{apv=0}) = \mathbb{C}(\texttt{i1=1} \triangleright \neg\texttt{(i1==0)}) = \{\texttt{i1=0} \triangleright \texttt{i1==0} \triangleright \texttt{apv=0}, \texttt{i1=1} \triangleright \neg\texttt{(i1==0)}\}$, i.e. this observer cannot distinguish whether the first input is 0 or 1. Thus, for a prefix trace in which the first two inputs are 0, $\mathbb{C}(\texttt{apv=1} \triangleright \texttt{i1=0} \triangleright \texttt{i1==0} \triangleright \texttt{apv=0} \triangleright \texttt{i2=0}) = \{\texttt{apv=1} \triangleright \texttt{i1=0} \triangleright \texttt{i1==0} \triangleright \texttt{apv=0} \triangleright \texttt{i2=0}, \texttt{apv=1} \triangleright \texttt{i1=1} \triangleright \neg\texttt{(i1==0)} \triangleright \texttt{i2=0}\}$, where `apv=1` and `i2=0` are known by this observer. In the same way, its cognizance on other traces can be generated. ☐

Observation Function For an observer with cognizance function $\mathbb{C}$, given a single trace σ, the observer cannot distinguish σ with traces in $\mathbb{C}(\sigma)$. In order to formalize the information that the observer can learn from σ, we apply the inquiry function $\mathbb{I}$ on each trace in $\mathbb{C}(\sigma)$, and get a set of maximal trace properties. By joining them together, we get the strongest property in $\mathcal{L}^{\mathsf{Max}}$ that σ can guarantee from the observer's perspective. Such a process is defined as the *observation* function $\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \sigma)$.

$$\mathbb{O} \in \wp(\mathrm{E}^{*\infty}) \mapsto \wp(\wp(\mathrm{E}^{*\infty})) \mapsto (\mathrm{E}^{*\infty} \mapsto \wp(\mathrm{E}^{*\infty})) \mapsto \mathrm{E}^{*\infty} \mapsto \wp(\mathrm{E}^{*\infty})$$

$$\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \sigma) \triangleq \qquad \text{observation (4)}$$
$$\textit{let } \alpha_{\mathsf{Pred}}[\![\mathcal{S}]\!](\mathcal{P}) = \{\sigma \in \mathsf{Pref}(\mathcal{P}) \mid \forall \sigma' \in \mathcal{S}.\ \sigma \preceq \sigma' \Rightarrow \sigma' \in \mathcal{P}\} \textit{ in}$$
$$\textit{let } \mathbb{I}(\mathcal{S}, \mathcal{L}, \sigma) = \Cap\{\mathcal{P} \in \mathcal{L} \mid \sigma \in \alpha_{\mathsf{Pred}}[\![\mathcal{S}]\!](\mathcal{P})\} \textit{ in}$$
$$\Cup\{\mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \sigma') \mid \sigma' \in \mathbb{C}(\sigma)\}.$$

From the above definition, it is easy to see that, for every invalid trace σ, $\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \sigma) = \bot^{\mathsf{Max}}$, since every trace σ' in $\mathbb{C}(\sigma)$ is invalid by (A2) and $\mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \sigma') = \bot^{\mathsf{Max}}$. In addition, for an omniscient observer with cognizance function $\mathbb{C}_o$, its observation $\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}_o, \sigma) = \mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \sigma)$.

Corollary 2. *For any observer with cognizance $\mathbb{C}$, if the corresponding observation function maps a trace σ to a maximal trace property $\mathcal{P} \in \mathcal{L}^{\mathsf{Max}}$, then σ guarantees the satisfaction of property $\mathcal{P}$ (i.e. every valid maximal trace that is greater than or equal to σ is guaranteed to have property $\mathcal{P}$).*

Lemma 3. *The observation function $\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C})$ is decreasing on the observed trace σ: the greater (longer) σ is, the stronger property it can observe.*

Example 9 (Access Control, Continued). For an omniscient observer, the observation function is identical to the inquire function in Example 7. If the cognizance of the second admin defined in Example 8 is adopted, we get an observation function that works exactly the same as the dashed arrows in Fig.2:

- $\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \texttt{apv=1} \triangleright \texttt{i1=0}) = \mathbb{I}\ (\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1} \triangleright \texttt{i1=0}) \Cup \mathbb{I}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1} \triangleright \texttt{i1=1}) = \texttt{RF} \Cup \top^{\mathsf{Max}} = \top^{\mathsf{Max}}$, i.e. even if the first input is already 0 in the trace, no property except $\top^{\mathsf{Max}}$ can be guaranteed for the second admin.
- $\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \texttt{apv=1} \triangleright \texttt{i1=0} \triangleright \texttt{i1==0} \triangleright \texttt{apv=0} \triangleright \texttt{i2=1}) = \mathbb{I}\ (\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1} \triangleright \texttt{i1=0} \triangleright \texttt{i1==0} \triangleright \texttt{apv=0} \triangleright \texttt{i2=1}) \Cup \mathbb{I}\ (\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1} \triangleright \texttt{i1=1} \triangleright \neg\texttt{(i1==0)} \triangleright \texttt{i2=1}) = \texttt{RF} \Cup \top^{\mathsf{Max}} = \top^{\mathsf{Max}}$, i.e. if the second input is 1, only $\top^{\mathsf{Max}}$ can be guaranteed.
- $\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \texttt{apv=1} \triangleright \texttt{i1=0} \triangleright \texttt{i1==0} \triangleright \texttt{apv=0} \triangleright \texttt{i2=0}) = \mathbb{I}\ (\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1} \triangleright \texttt{i1=0} \triangleright \texttt{i1==0} \triangleright \texttt{apv=0} \triangleright \texttt{i2=0}) \Cup \mathbb{I}\ (\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \texttt{apv=1} \triangleright \texttt{i1=1} \triangleright \neg\texttt{(i1==0)} \triangleright \texttt{i2=0}) = \texttt{RF} \Cup \texttt{RF} = \texttt{RF}$, i.e. `RF` is guaranteed only after the second input is entered 0. ☐

3.4 Formal Definition of Responsibility

Using the three components of responsibility analysis introduced above, responsibility is formally defined as the *responsibility abstraction* α_R in (5). Specifically, the first parameter is the maximal trace semantics $\mathcal{S}^{\mathsf{Max}}$, the second parameter is the lattice $\mathcal{L}^{\mathsf{Max}}$ of system behaviors, the third parameter is the cognizance function of a given observer, the fourth parameter is the behavior $\mathcal{B}$ whose responsibility is of interest, and the last parameter is the analyzed traces $\mathcal{T}$.

Consider every trace $\sigma_{\mathrm{H}}\sigma_{\mathrm{R}}\sigma_{\mathrm{F}} \in \mathcal{T}$ where H, R and F respectively stand for *History*, *Responsible part* and *Future*. If $\emptyset \subsetneq \mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \sigma_{\mathrm{H}}\sigma_{\mathrm{R}}) \subseteq \mathcal{B} \subsetneq \mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \sigma_{\mathrm{H}})$ holds, then σ_{H} does not guarantee the behavior $\mathcal{B}$, while $\sigma_{\mathrm{H}}\sigma_{\mathrm{R}}$ guarantees a behavior which is at least as strong as $\mathcal{B}$ and is not the invalidity property represented by $\perp^{\mathsf{Max}} = \emptyset$. Therefore, σ_{R} is said to be *responsible* for ensuring behavior $\mathcal{B}$ in the trace $\sigma_{\mathrm{H}}\sigma_{\mathrm{R}}\sigma_{\mathrm{F}}$.

In particular, the length of σ_{R} is restricted to be 1 (i.e. $|\sigma_{\mathrm{R}}| = 1$), such that the responsible entity σ_{R} must be a single event and the responsibility analysis could be as refined as possible. Otherwise, if we do not have such a restriction, then for every analyzed trace $\sigma \in \mathcal{T}$ where the behavior $\mathcal{B}$ holds, the responsibility analysis may split the trace σ into three parts $\sigma = \sigma_{\mathrm{H}}\sigma_{\mathrm{R}}\sigma_{\mathrm{F}}$ such that $\sigma_{\mathrm{H}} = \varepsilon$, $\sigma_{\mathrm{R}} = \sigma$ and $\sigma_{\mathrm{F}} = \varepsilon$. In such a case, $\emptyset \subsetneq \mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \sigma_{\mathrm{H}}\sigma_{\mathrm{R}}) \subseteq \mathcal{B} \subsetneq \mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \sigma_{\mathrm{H}})$ holds, and the whole trace σ would be found responsible for $\mathcal{B}$. This result is trivially correct, but too coarse to be useful in practice.

Responsibility Abstraction α_R

$$
\begin{aligned}
&\alpha_R \in \wp(\mathrm{E}^{*\infty}) \mapsto \wp(\wp(\mathrm{E}^{*\infty})) \mapsto (\mathrm{E}^{*\infty} \mapsto \wp(\mathrm{E}^{*\infty})) \\
&\qquad \mapsto \wp(\mathrm{E}^{*\infty}) \mapsto \wp(\mathrm{E}^{*\infty}) \mapsto \wp(\mathrm{E}^{*\infty} \times \mathrm{E} \times \mathrm{E}^{*\infty}) \qquad (5)\\
&\alpha_R(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \mathcal{B}, \mathcal{T}) \triangleq \\
&\quad \textit{let } \alpha_{\mathsf{Pred}}[\![\mathcal{S}]\!](\mathcal{P}) = \{\sigma \in \mathsf{Pref}(\mathcal{P}) \mid \forall \sigma' \in \mathcal{S}.\ \sigma \preceq \sigma' \Rightarrow \sigma' \in \mathcal{P}\} \textit{ in} \\
&\quad\quad \textit{let } \mathbb{I}(\mathcal{S}, \mathcal{L}, \sigma) = \Cap\{\mathcal{P} \in \mathcal{L} \mid \sigma \in \alpha_{\mathsf{Pred}}[\![\mathcal{S}]\!](\mathcal{P})\} \textit{ in} \\
&\quad\quad\quad \textit{let } \mathbb{O}(\mathcal{S}, \mathcal{L}, \mathbb{C}, \sigma) = \Cup\{\mathbb{I}(\mathcal{S}, \mathcal{L}, \sigma') \mid \sigma' \in \mathbb{C}(\sigma)\} \textit{ in} \\
&\quad\quad\quad\quad \{\langle\sigma_{\mathrm{H}},\ \sigma_{\mathrm{R}},\ \sigma_{\mathrm{F}}\rangle \mid \sigma_{\mathrm{H}}\sigma_{\mathrm{R}}\sigma_{\mathrm{F}} \in \mathcal{T} \wedge |\sigma_{\mathrm{R}}| = 1 \wedge \\
&\quad\quad\quad\quad\quad \emptyset \subsetneq \mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \sigma_{\mathrm{H}}\sigma_{\mathrm{R}}) \subseteq \mathcal{B} \subsetneq \mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \sigma_{\mathrm{H}})\}
\end{aligned}
$$

Since $\alpha_R(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \mathcal{B})$ preserves joins on analyzed traces $\mathcal{T}$, we have a Galois connection: $\langle\wp(\mathrm{E}^{*\infty}),\ \subseteq\rangle \underset{\alpha_R(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \mathcal{B})}{\overset{\gamma_R(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \mathcal{B})}{\leftrightarrows}} \langle\wp(\mathrm{E}^{*\infty} \times \mathrm{E} \times \mathrm{E}^{*\infty}),\ \subseteq\rangle$.

Lemma 4. *If σ_{R} is said to be responsible for a behavior $\mathcal{B}$ in a valid trace $\sigma_{\mathrm{H}}\sigma_{\mathrm{R}}\sigma_{\mathrm{F}}$, then $\sigma_{\mathrm{H}}\sigma_{\mathrm{R}}$ guarantees the occurrence of behavior $\mathcal{B}$, and there must exist another valid prefix trace $\sigma_{\mathrm{H}}\sigma'_{\mathrm{R}}$ such that the behavior $\mathcal{B}$ is not guaranteed.*

Recall the three desired points (time, free choices and cognizance) for defining responsibility in section 2.1. It is obvious that α_R has taken both the temporal ordering of events and the observer's cognizance into account. As for the free choices, it is easy to find from lemma 4 that, if σ_{R} is determined by its history

trace σ_{H} and is not free to make choices (i.e. $\forall \sigma_{\mathrm{H}}\sigma_{\mathrm{R}}, \sigma_{\mathrm{H}}\sigma'_{\mathrm{R}} \in \mathcal{S}^{\mathsf{Pref}}.\ \sigma_{\mathrm{R}} = \sigma'_{\mathrm{R}}$), then σ_{R} cannot be responsible for any behavior in the trace $\sigma_{\mathrm{H}}\sigma_{\mathrm{R}}\sigma_{\mathrm{F}}$.

3.5 Responsibility Analysis

To sum up, the responsibility analysis typically consists of four steps: **I)** collect the system's trace semantics $\mathcal{S}^{\mathsf{Max}}$ (in Section 3.1); **II)** build the complete lattice of maximal trace properties of interest $\mathcal{L}^{\mathsf{Max}}$ (in Section 3.2); **III)** derive an inquiry function $\mathbb{I}$ from $\mathcal{L}^{\mathsf{Max}}$, define a cognizance function $\mathbb{C}$ for each observer, and create the corresponding observation function $\mathbb{O}$ (in Section 3.3); **IV)** specify the behavior $\mathcal{B}$ of interest and the analyzed traces $\mathcal{T}$, and apply the responsibility abstraction $\alpha_R(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \mathcal{B}, \mathcal{T})$ to get the analysis result (in Section 3.4). Hence, the responsibility analysis is essentially an abstract interpretation of the event trace semantics.

In the above definition of responsibility, the semantics and lattice of system behaviors are concrete, and they are explicitly displayed in the access control example for the sake of clarity. However, they may be uncomputable in practice, and we do not require programmers to provide them in the implementation of responsibility analysis. Instead, they are provided in the abstract, using an abstract interpretation-based static analysis that is sketched in [13].

Example 10 (Access Control, Continued). Using the observation functions created in example 9, the abstraction α_R can analyze the responsibility of a certain behavior $\mathcal{B}$ in the set $\mathcal{T}$ of traces. Suppose we want to analyze "Read Failure" in every possible execution, then $\mathcal{B}$ is `RF`, and $\mathcal{T}$ includes all valid maximal traces, i.e. $\mathcal{T} = \mathcal{S}^{\mathsf{Max}}$. Thus, $\alpha_R(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}, \mathtt{RF}, \mathcal{S}^{\mathsf{Max}})$ computes the responsibility analysis result, which is essential the same as desired in Example 2.

Furthermore, the responsibility of "granting write access" can be analyzed by setting the behavior $\mathcal{B}$ as `RW` instead, and we get the following result. To the cognizance of an omniscient observer, in every execution that both the first two inputs are 1, the third input (i.e. system setting of permission type) is responsible for `RW`. Meanwhile, to the cognizance of the second admin who is unaware of the first input, no one is found responsible for `RW`, because whether the write access fails or not is always uncertain, from the second admin's perspective. □

4 Examples of Responsibility Analysis

Responsibility is a broad concept, and our definition of responsibility based on the abstraction of event trace semantics is universally applicable in various scientific fields. We have examined every example supplied in actual cause [19,20] and found that our definition of responsibility can handle them well, in which actions like "drop a lit match in the forest" or "throw a rock at the bottle" are treated as events in the trace. In the following, we will illustrate the responsibility analysis by two more examples: the "negative balance" problem of a withdrawal transaction, and the information leakage problem.

4.1 Responsibility Analysis of "Negative Balance" Problem

Example 11 (Negative Balance). Consider the withdrawal transaction program in Fig. 4 in which the `query_database()` function gets the balance of a certain bank account before the transaction, and `input()` specifies the withdrawal amount that is positive. When the withdrawal transaction completes, if the balance is negative, which entity in the program shall be responsible for it? □

It is not hard to see that, the "negative balance" problem can be transformed into an equivalent buffer overflow problem, where the memory of size `balance` is allocated and the index at `n-1` is visited. Although this problem has been well studied, it suffices to demonstrate the advantages of responsibility analysis over dependency/causality analysis.

```
1: balance = query_database();
2: n = input(); //Positive
3: balance -= n;
```

Fig. 4: Withdrawal Transaction Program

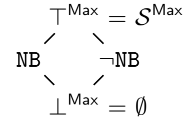

Fig. 5: Lattice of Behaviors

As discussed in section 3.5, the responsibility analysis consists of four steps. For the sake of simplicity, we consider only the omniscient observer here.

(1) Taking each assignment as an event, each maximal trace in this program is of length 3, and the program's maximal trace semantics consists of infinite number of such traces. E.g. `balance=0` ▷ `n=5` ▷ `balance=-5` denotes a maximal execution, in which the balance before the transaction is 0 and the withdrawal amount is 5 such that "negative balance" occurs.

(2) Since "negative balance" is the only behavior that we are interested here, a lattice $\mathcal{L}^{\mathsf{Max}}$ of maximal trace properties in Fig. 5 with four elements can be built, where `NB` (Negative Balance) is the set of maximal traces where the value of `balance` is negative at the end, and ¬`NB` is its complement.

(3) Using the omniscient observer's cognizance $\mathbb{C}_o$, the observation function $\mathbb{O}$ can be easily derived from the lattice $\mathcal{L}^{\mathsf{Max}}$, such that:
   - $\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}_o, \varepsilon) = \top^{\mathsf{Max}}$;
   - $\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}_o, \texttt{balance=i}) = \texttt{NB}$ where $\texttt{i} \leq 0$, i.e. if the balance before the transaction is negative or 0, the occurrence of "negative balance" is guaranteed before the withdrawal amount `n` is entered;
   - $\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}_o, \texttt{balance=i}) = \top^{\mathsf{Max}}$ where $\texttt{i} > 0$, i.e. if the balance before the transaction is strictly greater than 0, whether "negative balance" occurs or not has not been decided;
   - $\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}_o, \texttt{balance=i} \triangleright \texttt{n=j}) = \texttt{NB}$ where $\texttt{i} > 0$ and $\texttt{j} > \texttt{i}$, i.e. "negative balance" is guaranteed to occur immediately after `input()` returns a value strictly greater than `balance`;
   - $\mathbb{O}(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}_o, \texttt{balance=i} \triangleright \texttt{n=j}) = \neg\texttt{NB}$ where $\texttt{i} > 0$ and $\texttt{j} \leq \texttt{i}$, i.e. "negative balance" is guaranteed not to occur immediately after `input()` returns a value less than or equal to `balance`.

(4) Suppose the behavior $\mathcal{B} = \texttt{NB}$ and the analyzed traces $\mathcal{T} = \mathcal{S}^{\mathsf{Max}}$, the abstraction $\alpha_R(\mathcal{S}^{\mathsf{Max}}, \mathcal{L}^{\mathsf{Max}}, \mathbb{C}_o, \mathcal{B}, \mathcal{T})$ gets the following result. If `query_database()` returns 0 or a negative value, no matter what value `input()` returns, the function `query_database()` (i.e. event `balance=i`) is responsible for "negative balance", and further responsibility analysis shall be applied on the previous transactions of the database. Otherwise, if `query_database()` returns a value strictly greater than 0, the function `input()` (i.e. event `n=j`) takes the responsibility for "negative balance", thus "negative balance" can be prevented by configuring the permission granted to `input()` such that its permitted return value must be less than or equal to the returned value of `query_database()`.

4.2 Responsibility Analysis of Information Leakage

Essentially, responsibility analysis of information leakage is the same as read failure or "negative balance" problem, and the only significant distinction is on defining the behaviors of interest. Here we adopt the notion of non-interference [16] to represent the behavior of information leakage.

In the program, the inputs and outputs are classified as either *Low* (public, low sensitivity) or *High* (private, high sensitivity). For a given trace σ, if there is another trace σ' such that they have the same low inputs but different low outputs, then the trace σ is said to leak private information. If no trace in the program leaks private information (i.e. every two traces with the same low inputs have the same low outputs, regardless of the high inputs), the program is secure and has the non-interference property. Thus, for any program with maximal trace semantics $\mathcal{S}^{\mathsf{Max}}$, the behavior of "Information Leakage" `IL` is represented as the set of leaky traces, i.e. $\texttt{IL} = \{\sigma \in \mathcal{S}^{\mathsf{Max}} \mid \exists \sigma' \in \mathcal{S}^{\mathsf{Max}}.\ \mathit{low_inputs}(\sigma) = \mathit{low_inputs}(\sigma') \wedge \mathit{low_outputs}(\sigma) \neq \mathit{low_outputs}(\sigma')\}$, where functions *low_inputs* and *low_outputs* collects low inputs and outputs along the trace, respectively. The behavior of "No information Leakage" `NL` is the complement of `IL`, i.e. $\texttt{NL} = \{\sigma \in \mathcal{S}^{\mathsf{Max}} \mid \forall \sigma' \in \mathcal{S}^{\mathsf{Max}}.\ \mathit{low_inputs}(\sigma) = \mathit{low_inputs}(\sigma') \Rightarrow \mathit{low_outputs}(\sigma) = \mathit{low_outputs}(\sigma')\}$. Thus, the lattice $\mathcal{L}^{\mathsf{Max}}$ of maximal trace properties regarding information leakage can be built as in in Fig. 6. Further, the corresponding observation function $\mathbb{O}$ can be created, and the analysis result can be obtained by applying the responsibility abstraction.

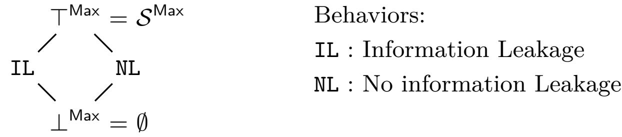

Fig. 6: Lattice of Behaviors regarding Information Leakage

Notice that we are interested in analyzing only the insecure programs in which some traces leak private information while others do not, i.e. $\texttt{IL} \subsetneq \top^{\mathsf{Max}}$. For the erroneous programs where every trace leaks private information, i.e.

$\texttt{IL} = \top^{\mathsf{Max}}$, we need to admit that our responsibility analysis cannot identify any entity responsible for the leakage, unless "launching the program" is treated as an event and it would be found responsible for leaking private information.

5 Related Work

Definition of Causality and Responsibility Hume [22] is the first one to specify causation by counterfactual dependence [29]. The best known counterfactual theory of causation is proposed by Lewis [28], which defines causation as a transitive closure of counterfactual dependencies. Halpern and Pearl [19,20,30] defines actual causality based on SEM and extends counterfactual dependency to allow "contingent dependency". Chockler and Halpern [8] defines responsibility to have a quantitative measure of the relevance between causes and effects, and defines blame to consider the epistemic state of an agent. Their application of actual causality, responsibility and blame is mainly on artificial intelligence.

Our definition of responsibility also adopts the idea of counterfactual dependence in the sense that, suppose an event σ_{R} is said to be responsible for behavior $\mathcal{B}$ in the trace $\sigma_{\mathrm{H}}\sigma_{\mathrm{R}}$, there must exist another event σ'_{R} such that, if σ_{R} is replaced by σ'_{R}, then $\mathcal{B}$ is not guaranteed (by lemma 4).

Error Cause Localization Classic program analysis techniques, e.g. dependency analysis [1,7,37] and program slicing [39,38,27,2], are useful in detecting the code that may be relevant to errors, but fail to localize the cause of error.

In recent years, there are many papers [3,18,25,17,34,33,32,24] on fault localization for counterexample traces, and most of them compare multiple traces produced by a model checker and build a heuristic metric to localize the point from which error traces separate from correct traces. Other related papers include error diagnosis by abductive/backward inference [14], tracking down bugs by dynamic invariant detection [21]. Actual causality is applied to explain counterexamples from model checker [5] and estimate the coverage of specification [9]. Besides, there are researches on analyzing causes of specific security issues. E.g. King et al. [26] employ a blame dependency graph to explain the source of information flow violation and generate a program slice as the error report.

Compared to the above techniques, this paper succeeds to formally define the cause or responsibility, and the proposed responsibility analysis, which does not require a counterexample from the model checker, is sound, scalable and generic to cope with various problems.

6 Conclusion and Future Work

This paper formally defines responsibility as an abstraction of event trace semantics. Typically, the responsibility analysis consists of four steps: collect the trace semantics, build a lattice of behaviors of interest, create an observation function for each observer, and apply the responsibility abstraction on analyzed traces. Its effectiveness has been demonstrated by several examples.

In the future, we intent to: (1) formalize the abstract responsibility analysis that is sketched in [13], (2) build a lattice of responsibility abstractions to cope with possible alternative weaker or stronger definitions of responsibility, (3) generalize the definition of cognizance function as an abstraction of system semantics, and (4) study the responsibility analysis of probabilistic programs.

Acknowledgment

This work was supported in part by NSF Grant CNS-1446511. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. P. Cousot thanks Marco Pistoia for initial discussions on responsibility while visiting the Thomas J. Watson Research Center at Hawthorne in 2005.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G.: A core calculus of dependency. In: POPL. pp. 147–160. ACM (1999)
2. Agrawal, H., Horgan, J.R.: Dynamic program slicing. In: PLDI. pp. 246–256. ACM (1990)
3. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: POPL. pp. 97–105. ACM (2003)
4. Beebee, H., Hitchcock, C., Menzie, P.: The Oxford Handbook of Causation. Oxford University Press (2009)
5. Beer, I., Ben-David, S., Chockler, H., Orni, A., Trefler, R.J.: Explaining counterexamples using causality. Formal Methods in System Design **40**(1), 20–40 (2012)
6. Chen, B., Pearl, J., Bareinboim, E.: Incorporating knowledge into structural equation models using auxiliary variables. In: IJCAI. pp. 3577–3583. IJCAI/AAAI Press (2016)
7. Cheney, J., Ahmed, A., Acar, U.A.: Provenance as dependency analysis. Mathematical Structures in Computer Science **21**(6), 1301–1337 (2011)
8. Chockler, H., Halpern, J.Y.: Responsibility and blame: A structural-model approach. J. Artif. Intell. Res. **22**, 93–115 (2004)
9. Chockler, H., Halpern, J.Y., Kupferman, O.: What causes a system to satisfy a specification? ACM Trans. Comput. Log. **9**(3), 20:1–20:26 (2008)
10. Christopher, W.J.: Structural Equation Models, From Paths to Networks. Studies in Systems, Decision and Control 22, Springer (2015)
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)
12. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. pp. 269–282. ACM Press (1979)
13. Deng, C., Cousot, P.: Responsibility analysis by abstract interpretation. Extended version of this paper, at `http://cs.nyu.edu/~deng/` (2019)
14. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI. pp. 181–192. ACM (2012)

15. Frankle, J., Park, S., Shaar, D., Goldwasser, S., Weitzner, D.J.: Practical accountability of secret processes. In: USENIX Security Symposium. pp. 657–674. USENIX Association (2018)
16. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. pp. 11–20. IEEE Computer Society (1982)
17. Griesmayer, A., Staber, S., Bloem, R.: Automated fault localization for C programs. Electr. Notes Theor. Comput. Sci. **174**(4), 95–111 (2007)
18. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. STTT **8**(3), 229–247 (2006)
19. Halpern, J.Y., Pearl, J.: Causes and explanations: A structural-model approach: Part 1: Causes. In: UAI. pp. 194–202. Morgan Kaufmann (2001)
20. Halpern, J.Y., Pearl, J.: Causes and explanations: A structural-model approach. part i: Causes. The British journal for the philosophy of science **56**(4), 843–887 (2005)
21. Hangal, S., Lam, M.S.: Tracking down software bugs using automatic anomaly detection. In: ICSE. pp. 291–301. ACM (2002)
22. Hume, D.: An enquiry concerning human understanding. London: A. Millar (1748), http://www.davidhume.org/texts/ehu.html
23. Jagadeesan, R., Jeffrey, A., Pitcher, C., Riely, J.: Towards a theory of accountability and audit. In: ESORICS. Lecture Notes in Computer Science, vol. 5789, pp. 152–167. Springer (2009)
24. Jin, H., Ravi, K., Somenzi, F.: Fate and free will in error traces. In: TACAS. Lecture Notes in Computer Science, vol. 2280, pp. 445–459. Springer (2002)
25. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: PLDI. pp. 437–446. ACM (2011)
26. King, D., Jaeger, T., Jha, S., Seshia, S.A.: Effective blame for information-flow violations. In: SIGSOFT FSE. pp. 250–260. ACM (2008)
27. Korel, B., Rilling, J.: Dynamic program slicing methods. Information & Software Technology **40**(11-12), 647–659 (1998)
28. Lewis, D.: Causation. The journal of philosophy **70**(17), 556–567 (1973)
29. Menzies, P.: Counterfactual theories of causation. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University, winter 2017 edn. (2017)
30. Pearl, J.: Causality: Models, Reasoning and Inference. Cambridge University Press, 2nd edn. (2013)
31. Pistoia, M., Flynn, R.J., Koved, L., Sreedhar, V.C.: Interprocedural analysis for privileged code placement and tainted variable detection. In: ECOOP. Lecture Notes in Computer Science, vol. 3586, pp. 362–386. Springer (2005)
32. Qi, D., Roychoudhury, A., Liang, Z., Vaswani, K.: Darwin: an approach for debugging evolving programs. In: ESEC/SIGSOFT FSE. pp. 33–42. ACM (2009)
33. Ravi, K., Somenzi, F.: Minimal assignments for bounded model checking. In: TACAS. Lecture Notes in Computer Science, vol. 2988, pp. 31–45. Springer (2004)
34. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ASE. pp. 30–39. IEEE Computer Society (2003)
35. Rival, X.: Understanding the origin of alarms in astrée. In: SAS. Lecture Notes in Computer Science, vol. 3672, pp. 303–319. Springer (2005)
36. van Sliedregt, E.: Individual Criminal Responsibility in International Law. Oxford Monographs in International Law, Oxford University Press (2012)
37. Urban, C., Müller, P.: An abstract interpretation framework for input data usage. In: ESOP. Lecture Notes in Computer Science, vol. 10801, pp. 683–710. Springer (2018)

38. Weiser, M.: Program slicing. In: ICSE. pp. 439–449. IEEE Computer Society (1981)
39. Weiser, M.: Program slicing. IEEE Trans. Software Eng. **10**(4), 352–357 (1984)
40. Weitzner, D.J., Abelson, H., Berners-Lee, T., Feigenbaum, J., Hendler, J.A., Sussman, G.J.: Information accountability. Commun. ACM **51**(6), 82–87 (2008)