Interaction-aware Actual Causation: A Building Block for Accountability in Security Protocols

Submitted in partial fulfillment of the requirements for
the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Divya Sharma

B.E., Electronics & Electrical Communications Engg., PEC University of Technology

M.S., Electrical & Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

September 2015

Thesis Committee

Professor Anupam Datta (Chair), Carnegie Mellon University
Professor Dilsun Kaynar, Carnegie Mellon University
Professor Lujo Bauer, Carnegie Mellon University
Professor Peter Spirtes, Carnegie Mellon University
Professor Joseph Halpern, Cornell University

'Causality is not an easy topic to speak about, but it is a fun topic to speak about. It is not easy because like religion, sex, and intelligence, causality was meant to be practiced, not analyzed. It is fun because, like religion, sex, and intelligence, emotions run high; examples are plenty; there are plenty of interesting people to talk to; and above all, the experience of watching our private thoughts magnified under the microscope of formal analysis is exhilarating.' – Judea Pearl, Winner of 2012 A.M. Turing Award.

Abstract

Protocols involving multiple agents and their interactions are ubiquitous. Protocols for tasks such as authentication, electronic voting, and secure multi-party computation ensure desirable security properties if participating agents follow their prescribed programs. However, if some agents choose to deviate from their prescribed programs and a security property is violated, it is important to hold agents accountable, i.e. *assign blame* for their choices and actions, and to fix deficiencies in the protocol design. Prior work in accountability has focused primarily on detecting or punishing deviations. This dissertation proposes a novel interaction-aware approach to *actual causation* (i.e., the identification of particular agents' choices to deviate, and interactions which caused a specific violation). We propose this approach as a useful building block for accountability in interacting multi-agent systems, including but not limited to security protocols.

The definitions of actual cause in this dissertation are inspired by prior work on actual causation in philosophy, law, and computer science. However, prior frameworks do not account for the program dynamics that arise in protocol-based settings and do not naturally capture *agent interactions* and *agents' choices to deviate*. Motivated by these applications and challenges, we make two main contributions.

First, we propose a theory of actual causation with choice and interaction as key components. Specifically, we define in an interacting program model, what it means for a *sequence of program expressions* (modeling choices, actions and interactions) to be an actual cause of a violation. We demonstrate that our theory significantly advances the state-of-the-art in the research area of actual causation by combining process-oriented and counterfactual-based viewpoints in prior work. A careful treatment of interaction and choice enables us to cleanly deal with a known set of issues that plague extant theories, including expressing concise interaction models and over-permissive counterfactual-based definitions.

Second, we demonstrate the value of this theory in the domain of security and privacy protocols, by proving that violations of a specific class of safety properties always have an actual cause. We also present a sound technique for establishing program actions as actual causes. Additionally, we provide a causal analysis of a representative protocol, designed to address weaknesses in the current public key certification infrastructure. Our theory clearly distinguishes between deviances and actual causes which is important from the standpoint of accountability.

Acknowledgments

Throughout my PhD, I have had the privilege of meeting several individuals who have shaped the way I think and shaped the way I have evolved as a researcher.

I would like to thank my advisor, Prof. Anupam Datta. I have benefitted tremendously from his focus on rigor, clarity of thought and the bigger picture. His feedback has helped me improve my writing skills significantly. This thesis has evolved to be one of the most challenging endeavors for me, and a topic worth thinking about for many years to come. I am thankful to Anupam for letting me take ownership of this work and his constant encouragement, over the years. It has been a great learning experience.

I am grateful to all my doctoral committee members for agreeing to serve on my committee, and for the numerous discussions and constant feedback: Prof. Dilsun Kaynar, who is also a close collaborator. I have really enjoyed dissecting our definitions together and working with her has been an immensely rewarding experience. Prof. Peter Spirtes, who introduced me to a number of philosophy related works and has always helped with the inputs on causality-related aspects. Prof. Joseph Halpern, whose work inspired the counterfactual flavor in our work and whose sharp questions and comments, helped strengthen the arguments presented in the dissertation. Prof. Lujo Bauer, whose feedback regarding applications and novelty of this work helped improve the content of the thesis significantly.

I would like to thank my collaborators, Prof. Deepak Garg and Arunesh Sinha – they had a significant influence on this research and were great mentors. Thanks to Deepak, Anupam and Dilsun for feedback on multiple iterations of this dissertation. Some of the most important ideas in this dissertation came up during brainstorming sessions with them and I have learnt most of what I know about research from all of them. I have also benefitted immensely from my interactions with other members of our research group, especially Omar Chowdhury and Michael Tschantz.

I had the privilege of collaborating with other excellent researchers. I worked with Prof. Limin Jia and Prof. Lujo Bauer on a project at the beginning of my PhD, which was a great introduction to research and a fun experience. Prof. Lorrie Cranor has been a great positive influence during my years in Cylab. I really enjoyed working on a project related to court records with Prof. Helen Nissenbaum. I would also like to thank other researchers in Cylab with whom I had encouraging interactions.

I had the opportunity to intern at two excellent research labs during my time

as a graduate student. It gave me the first taste of applying research to a product in real life. The team at Symantec Research Labs, especially Sanjay Sawhney, Sharada Sundaram and Darren Shou were a pleasure to work with. My second internship at Microsoft Research Cambridge was a great experience as well. I sincerely want to thank two people who introduced me to research during my years as an undergraduate and had a strong influence on my decision to pursue a PhD: Prof. Bernhard Plattner (ETH Zurich) and Prof. Abhay Karandikar (IIT Bombay).

I feel lucky to have great collaborators (and their better halves) who are also good friends – Arunesh, Pushpa, Deepak, Vasundhara, Limin, Omar, Amit, Dilsun, you'll be missed!

Working at Cylab has been an absolute pleasure, especially because of the outstanding Cylab staff who supported our work. Thank you, Cylab and ECE administrative staff – especially Megan, Karen, Samantha S., Tina, Toni, Kelley, Rachael, Ivan, Samantha G. Outside my own research group, the biometrics lab adopted me as one of their own. Utsav, Shreyas, Ramzi, Keshav, Sekhar and other folks in the biometrics lab were my go-to people when I ran out of moral support or food at my desk (Ramzi, I made it but I still don't have the answers!). I really enjoyed interacting with folks in CUPS lab (especially Michelle, Manya, Blase). The entire Cylab community provided one of the best working environments.

Shreyas, Sarah, Arunesh, Pushpa – thank you for opening your hearts, your kitchens and your couches so generously! In my final two years, it was great to have Ashwati as a flatmate. Our informal 'thesis writing group' (or rather Pittsburgh dining society) with Utsav, Varun and Supreeth made the last two months of thesis writing less stressful. Thanks to all the other people who made living in Pittsburgh an amazing experience, especially, Vivek, Niranjini, Divya H., GC, SKB, Sudhanshu, Lavanya; Arshi, Mudit, Krishna, Rishu, Sabah, Sulabh, Amar, Shruti; and Arun, Uday, Aranya, Ashwin, Siddharth N., Ishani, Marco, Sneha, Veda, Prof. Carla Larocca, Dr. Michele and the IGSA folks I met during my time as a performer/ dance coordinator/ board member.

I want to thank my friends and mentors from my undergraduate school, many of them were involved in finalizing the decision to pursue a PhD, especially SMB, Aastha, Kushal, Madhur, and other friends in PEC.

Thanks to Deepika M. and Arvind for always being there. I want to thank my family for their constant support despite always wondering why I chose this path, especially my parents, Indra and Raj Kumar, and my sister, Yajnika. I want to thank

my parents for many reasons, particularly for instilling in me a strong work ethic, teaching me how to be over-ambitious and most importantly, letting me do something different. My brother, Mohit, has been a great support through the good and and the bad, for all my endeavors. This thesis is dedicated to (*there has to be a list!*):

my parents, without whom none of this would have started,
Mohit and Arvind, without whom none of this would have reached an end,
the punjabi spirit, without which none of this would have sustained.

This research has been partly funded by CyLab at Carnegie Mellon University under grants CNS 1423168 and CCF 0424422 from the National Science Foundation (NSF), and Multidisciplinary Research Program of the University Research Initiative (MURI) on Science of Cybersecurity by Air Force Office of Scientific Research (AFOSR).

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

1.1.1 Motivation of problem

Accountability mechanisms complement preventive security and privacy mechanisms by detecting policy violations after they occur, identifying agents to blame for violations, and punishing the violators. Ensuring accountability for security protocols is essential in a wide range of settings. For example, protocols for authentication and key exchange [2], electronic voting [3, 4], auctions [5], contract signing [4, 6] and secure multiparty computation [7] are widely used multi-agent protocols that require strong accountability guarantees due to the involvement of trusted third parties. These multi-agent protocols ensure desirable security properties if participating agents follow their prescribed programs. However, if some of these agents choose to deviate from their prescribed programs and a security property is violated, it is important to analyze the violation. This is crucial for two reasons: first, to hold agents accountable (*assign blame*) for their choices and actions, and second, to fix deficiencies (*protocol debugging*) in the protocol design which enabled the violation. The importance of accountability in multi-agent systems has been recognized in prior work [4, 6, 8, 9, 10, 11, 12].

While the research community has recognized the problem of accountability and its complexity, none of the existing works provide an approach which connects agents' actions to the violation they are held accountable for. Prior work in accountability has focused primarily on detecting or punishing deviations, regardless of whether the deviations and resulting interactions actually led to the violation.

1.1.2 Motivation of our approach

This dissertation proposes *actual causation* (i.e., the identification of particular agents' choices to deviate and their interactions, which caused a specific violation) as a building block for accountability in decentralized multi-agent systems, including but not limited to security protocols and ceremonies.

We consider a simple protocol example in order to illustrate the key points of our approach. In the movie *Flight* [13], a pilot drinks alcohol and snorts cocaine before flying a commercial plane, and the plane goes into a locked dive in mid-flight. While the pilot's behavior is found to be deviant in this case—he does not follow the prescribed protocol (program) for pilots—it is found to not be the actual cause of the plane's dive. The actual cause was a deviant behavior by the maintenance staff—they did not replace a mechanical component that should have been replaced. Ideally, the maintenance staff should have inspected the plane prior to take-off according to their prescribed protocol.

This example is useful to illustrate several key ideas that influence the formal development in this dissertation. First, it illustrates the importance of capturing the *actual interactions* among agents in a decentralized multi-agent system with non-deterministic execution semantics. The events in the movie could have unfolded in a different order but it is clear that the actual cause determination needs to be done based on the sequence of events that happened in reality. For example, had the maintenance staff replaced the faulty component *before* the take-off, the plane may not have gone into a dive.

Second, the example motivates us to hold accountable agents who exercised their *choice* to execute a deviant program that actually caused a violation. The maintenance staff had the choice to replace the faulty component or not. Our model provides natural constructs to express these choices which are crucial for the violation.

Third, the example demonstrates that the task of replacing the component could consist of *multiple steps*, for instance, the program to be followed by the maintenance staff. It is important to identify which of those steps were crucial for the occurrence of the dive. Thus, we focus on formalizing *program actions* executed in sequence as actual causes of violations rather than individual, independent events as formalized in prior work.

Fourth, the example highlights the interaction-aware approach in constructing counterfactual scenarios for program-based settings. We focus on counterfactual-based definitions of actual causation where a causal judgment is made based on what could have happened in alternative worlds. Our definition considers alternative scenarios where parts of certain programs would not have executed or certain choices could have been made differently. For the example, we test whether the plane would have gone into a nose dive, if only some of the choices and

interactions of the staff and the pilot were replayed, as in the actual setting.

Finally, the example highlights the difference between deviance and actual causation. This difference is important from the standpoint of accountability. In particular, the punishment for deviating from the prescribed protocol could be suspension or license revocation whereas the punishment for actually causing a plane crash in which people died could be significantly higher (e.g., imprisonment for involuntary manslaughter). The first four ideas, reflecting our program-based treatment, are the most significant points of difference from prior work on actual causation [14, 15, 16, 17, 18], while the last idea is a significant point of difference from prior work in accountability [4, 6, 19, 20]. The first four ideas reflect our interaction-aware and choice-aware approach to actual causation.

Prior work on accountability in computer security and cryptography has focused on deviant detection and punishment and has not studied actual causation as a building block for blame-assignment [4, 6, 10, 12, 19, 21]. This gap can result in inappropriate or unjustified blame assignment as well as the inability to distinguish between joint and independent causes [4, 12, 22, 23] — weaknesses that our formalization addresses[1].

The problem of causation is ubiquitous. The central contribution of this dissertation is **a formal definition of actual causation which is interaction-aware and choice-aware and finds actual causes at the fine-grained level of program actions.** This dissertation makes a significant contribution to the fields of security, formal methods and analytical philosophy – ours is the first work which proposes an interaction-aware approach to actual causation in order to connect actions to occurrence of violation in interacting systems. The approach proposed in this dissertation has significance not only for the problems of accountability in the above mentioned areas, but also in other fields where determining actual causes in (non-deterministic) interacting systems is important.

1.2 Overview of our approach

In this section, we provide an outline of our actual cause definition.

Terminology. Throughout this dissertation, we use the terms counterfactual and non-determinism. We first explain how we interpret these commonly used terms.

- Processes are used to model different types of systems. For instance, in certain cases processes model interactive programs, while in other cases processes can be used to model naturally occurring behavior (as in most of the physical sciences) [24]. In certain cases,

[1] See Chapters 4, 6 for a detailed comparison with related work.

individual steps which constitute a process can be specified, this is especially true when processes model programs. In this dissertation, we focus on such processes and their interactions.

- A process might be deterministic, probabilistic or non-deterministic. Deterministic processes produce the same outputs (results) when re-run with the same inputs. For example, the physical process underlying determination of force from mass and acceleration is a deterministic process. Probabilistic processes may produce different results when re-run with the same inputs because of uncertainty about their execution. For instance, the computational process underlying a modern cryptographic algorithm for encryption is a probabilistic process — when the same message is input to such an algorithm it produces different results in accordance with a probabilistic transition function. Non-determinism is an abstraction introduced by computer scientists to model probabilistic systems whose exact transition function probabilities are not known. They are defined using a possibilistic transition function that defines for each system state the possible next states, while ignoring the probabilities of the transitions. For instance, this abstraction is often used in interacting systems (e.g., distributed algorithms, security protocols) where the exact order of interactions are not known. Instead the scheduler is modeled using non-determinism to capture all possible execution orders.
- The word *counterfactual* refers to settings which are counter to the given fact, i.e. alternative possibilities. Counterfactual settings can be obtained by allowing arbitrary modifications to the outcome of the involved processes (for instance, in settings where the processes being modeled are fixed), or by allowing modifications to parts of the process and its inputs (for instance, in case of programs). In this dissertation, we adopt the latter strategy for constructing counterfactual scenarios.

Process-oriented view. In this dissertation, we focus on the *processes* rather than individual events, i.e. variable assignments for process outcome (as is traditionally done in the actual cause literature). We believe that this process-oriented view is better suited to security settings where agents exercise their choice to follow prescribed *programs* or deviate from them, rather than unrelated individual steps. If we were to solely focus on the outcome of the program, then this level of detail cannot be captured.

We model a process as a sequence of program expressions. These program expressions capture choices, other actions and interactions amongst processes at a fine-grained level, which is significant for our goal of accountability. This design choice is important to understand the effect of modifying the constituent steps of a process in the model (for instance, in case of

interactive programs), as well as in fields where analyzing the modification made to the model is also significant (for instance, models for processes in the natural sciences).

Some (or all) of the expressions in a process, may execute and result in a sequence of executed program expressions. Each executed expression may return a value, however for the purpose of accountability we focus on the general sequence of program expressions and not the outcome values. We call this sequence of uninstantiated expressions a log. The log may or may not satisfy certain properties of interest. We propose a definition of cause that can be used to identify a subsequence of the log, as a cause of a property being true (or occurrence of an event). We call such a subsequence as *causal sequence*.

Our goal is to identify the relevant parts of all interacting processes (corresponding to the relevant sequence of executed program expressions) rather than the entire program, which led to occurrence of a certain property. The level of abstraction of program expressions is useful for accountability for two reasons. First, for a specific violation V (which is a property), even though a program P might be deviant, and some expressions of program P might have executed, but those expressions may not be a part of the causal sequence that leads to the violation. Formalizing program expressions rather than entire programs, allows us to provide a basis for exonerating such programs from blame for violation V. Second, differentiating between choices and other actions provides us a way to easily identify the relevant choices and blame the agents who made those choices[2].

1.2.1 Definition outline

We envision our definition to be applicable in settings where agents execute their programs concurrently with other programs in an interacting system and a reliable log of their actions is available. Let l be a log of agents' executed program expressions (modeling the interactions and the internal choices). All agents are supposed to collectively satisfy a property if they follow the given programs. Let V be a set of traces, representing a violation of the intended property. For our definition, we assume that a system of interacting agents is completely specified and if a system goal is not met, then we have access to the log l. The definition presented here further assumes this log of events to be complete.

The basic idea of our definition is to identify a sequence of program expressions executed in $\vec{\alpha}$ to be an actual cause of the violation V, while implicitly establishing the irrelevance of other agents' choices and actions to the violation.

[2]This might also be relevant in a setting where the programs are fixed and internal choices are the only parameters that can vary across executions.

Constructing counterfactual scenarios. We consider a set of counterfactual scenarios where the agents made the same choices as the choices selected in $\vec{\alpha}$, executed the same actions as in $\vec{\alpha}$ and the interactions amongst all the agents are preserved as in $\vec{\alpha}$. We test the irrelevance of the other actions and choices (i.e., expressions not in $\vec{\alpha}$) by eliminating these expressions. In order to construct these counterfactual scenarios, we modify the processes to remove the expressions not included in the putative causal sequence, in an interaction-aware manner. Next, we generate feasible executions of the model and test for the effect on all executions resembling the log. The output values generated by processes may vary due to the non-determinism in the execution semantics as described above[3].

Informally speaking, we say that $\vec{\alpha}$ is an actual cause of V on a log l if the following properties hold:

- *Occurrence*: The violation has occurred on the log l. That is, the corresponding execution trace[4] is in V.
- *Sufficiency*: Eliminate all expressions not in $\vec{\alpha}$ from interacting processes and consider all executions similar to the log. Then, $\vec{\alpha}$ suffices to obtain the same violation on all resulting executions, even if the eliminated expressions were to return arbitrary values, i.e. we test for all possible values which could be returned by the eliminated expressions.
- *Minimality*: No proper subsequence of $\vec{\alpha}$ satisfies both the conditions above.

Revisiting the *Flight* example. In the *Flight* example introduced at the beginning of the chapter, the log records the pilot's choice and action to fly the plane, as well as the staff's choice to deviate and the malfunctioning of the plane. There are two potential causal sequences, first, the choice made by the pilot to drink and consequently fly the plane, and second, the choice made by the maintenance staff to not replace the component and allowing the plane to fly. For both these sequences, *Occurrence* condition is satisfied since the plane malfunctioned. Let us consider the sequence involving only the pilot's actions. For the *Sufficiency* condition, we will preserve the pilot's choices and actions as on the log. For all other actions and choices not on the log (i.e. the staff's program), we will test all possible scenarios that could have resulted. We can show that in at least one of the cases, had the staff replaced the component,

[3]In contrast with prior theories of actual causation, our counterfactual scenarios test for all possible returned values for the removed expressions as opposed to existential quantification over values for expressions excluded from the causal sequence.

[4]we explain the difference between a log and a trace, formally, in Chapter 2. Informally speaking, a log is the sequence of un-instantiated program expressions where the execution trace contains the instantiated values for all the executed expressions. For instance, if a trace contains the expression `send x`, it will also specify the value of x. In contrast, the log of this trace will only contain the un-instantiated expression: `send x`.

then the malfunction would not have happened[5]. Similarly, when we test the second sequence consisting of only the staff's actions, we will test all possible scenarios that could have resulted from the pilot's actions. In this case, we find that as long as the maintenance staff chose to not replace the component and allow the plane to fly – the plane will malfunction in all the cases, irrespective of whether the pilot is sober or not. Therefore, this second sequence satisfies the sufficiency condition. We can also show that this sequence is minimal since we need both the staff's choice and interactions (for instance, with the crew to allow the take-off) to demonstrate the sufficiency clause. This example illustrates that the causal analysis can be used as a *building block* for evidence to show that the staff, and not the pilot, should be held accountable for the plane's malfunction, even though both could be punished for the deviation.

Process calculus formalism. The settings where we are interested in answering the questions formulated above, primarily involve protocols where multiple agents make choices and interact. In such settings, different executions may arise even if the programs executed by agents are fixed. The *non-determinism* in the execution semantics could arise due to several factors, including, choosing different inputs, a different interleaving of actions or different synchronizations across communicating programs.

The desire to model interaction, capture program dynamics and non-deterministic execution semantics, motivates us to use a process calculus to formalize our actual cause definition. Process calculi [25] have been widely used for modeling interacting or communicating systems. Process calculus contains natural constructs to model both choice and interaction[6]. We highlight several advantages of using a process calculus framework for the questions we focus on. These include capturing sequential and interaction-based dependencies[7].

A note on the term 'actual causation'. Consider a simple example: John smoked and was diagnosed with cancer. Finding an answer to the question, 'did John's smoking cause John's cancer?' will fall under the purview of finding the *actual cause* (or token cause) of John's cancer. In contrast, finding an answer for the question 'does smoking cause cancer?' falls under the category of *type causation* (or general causation). It is common to distinguish actual causation, which is the relation between variable values, from type causation, which is the relation between random variables. Type causation [26, 27] is concerned with relations between random variables and is considered by several philosophers to be the pre-requisite for actual causation.

[5] As per the movie [13], the pilot's intoxication level was not enough to lead to any malfunction by itself.

[6] Another advantage of using process calculi for our causal analysis is that traditionally process calculi have been used for demonstrating correctness of protocols and have been of importance in proving other properties about concurrent execution of programs.

[7] See Chapter 2 for a detailed description.

There is, however, no consensus on how type causation and actual causation are related [28, 29]. In this dissertation, we focus on *actual causes of violations*, i.e. given a specific violation, we find its causes[8].

1.3 Prior work in actual causation

1.3.1 Counterfactual-based actual causation theories

Causation has been of interest to philosophers and ideas from philosophical literature have been introduced into computer science by the seminal works of Halpern and Pearl [14, 27, 30] as well as Spirtes, Glymour and Schienes [26]. In particular, counterfactual reasoning is appealing as a basis for study of actual causation. Indeed, actual causation is a building block for causal explanations [31], which can be used to provide an account of why a violation happened. It is also a building block for blame assignment in influential theories of moral and legal blame [32, 33, 34, 35, 36, 37].

Much of the definitional activity in philosophy, law, and computer science has centered around the question of what it means for an event to be an actual cause of another event. Notably, Hume [38] identified actual causation with counterfactual dependence—the idea that c is an actual cause of e if had c not occurred then e would not have occurred. The counterfactual interpretation of actual causation has been developed further and formalized in a number of influential works (see, for example, [14, 16, 17, 27, 39, 40, 41, 42]). This concept has generated significant interest in philosophy and law in part because of its connection with issues of moral and legal responsibility (see [32, 33]). Indeed, that is also why we view actual causation as a useful building block for accountability in security settings[9].

1.3.2 Process-based causation theories

Another prominent line of work commonly referred to as 'causal process theories' is based on the notion that causation should be understood in terms of continuous causal processes and interactions between them, rather than causal relations between discrete events [44]. This line of work, attributed primarily to Salmon, originated in his work on explaining physical processes. According to Salmon, a process is anything with constancy over time. His theory

[8]Specifically, in this work we focus on the theories which have been proposed for actual causation, using the structural equation framework, and employ counterfactual based reasoning. See Chapter 4 for details.

[9]A different characterization of causal relation has been proposed in terms of 'production', i.e. c is a cause of e if c generates event e or c helps to bring about e [43]. In this dissertation, we focus on counterfactual-based theories of actual causation.

makes a fundamental distinction between a causal process and a pseudo process where a causal process is defined as one that is capable of transmitting a local modification of a characteristic. Prior work in this field has primarily focused on differentiating causal processes from pseudo processes [44, 45, 46].

1.3.3 Challenges

Formalizing actual cause as a building block for accountability in multi-agent interacting systems raises new conceptual and technical challenges beyond those addressed in the literature on events as actual causes as well as process-based theories of actual causation. Prior work in counterfactual-based actual causation finds individual events (i.e. variable assignments for processes) as causes of a violation. On the other hand, prior work in process causation-based theories does not focus on finding causes for certain properties being true. In particular, event-based causation frameworks do not account for the program dynamics that arise in such settings and lack constructs that naturally capture *agent interactions*. Additionally, in prior counterfactual-based actual causation literature, the processes are deterministic and cannot naturally capture the program dependencies in a decentralized multi-agent system such as the ones we are interested in. To the best of our knowledge, prior work in actual causation has not encoded non-deterministic systems or does not provide a mathematical formulation for such a theory, which makes the direct application of these frameworks to security protocols challenging[10].

1.4 Thesis statement

Our thesis statement is the following:

> *An interaction-aware and choice-aware approach is essential to define actual causation in interacting systems. Additionally, this approach provides a useful building block for accountability (i.e., to assign blame for and explain violations) in such systems.*

This dissertation focuses on formalizing program actions as actual causes and developing analysis techniques for inferring causes of security violations.

[10] We discuss the differences with counterfactual-based theories in detail in Chapter 4. To the best of our knowledge, process based-theories do not have a mathematical formulation at a similar level of detail as counterfactual-based theories described using structural equations, making it difficult to perform a deeper comparison. We discuss these theories briefly in Chapter 4.

1.5 Summary of contributions

This dissertation makes the following contributions in the fields of actual causation and accountability in security and privacy protocols:

1. **Contributions to theory of actual causation:**

   Here are the central contributions and each of these addresses weaknesses in prior literature in actual causation:

   Definitional:

   (a) **Modeling interaction and choice.** We propose a theory of actual causation in interacting systems with choice and interaction as the key components. In contrast with prior work on counterfactual-based actual causation, our theory treats these constructs differently from other constructs in the formal language.

   (b) **Combining process-oriented view and counterfactual-based view.** A central aspect of our theory of causation lies in blending ideas from process-based and counterfactual-based traditions of actual causation. We adopt a process-based view and focus on interactions and finding processes (i.e. program expressions) as causes. On the other hand, our definition uses counterfactuals to find actual causes. With the help of examples from actual causation literature, we demonstrate that our process-oriented and interaction-aware approach to actual causation has several useful features:

      i. **Finding causal sequences** (*What is a cause?*). We capture a causal sequence as opposed to individual events (i.e. variable assignments for process outcomes), since during our causal analysis we preserve dependencies (choices and interactions) within the programs and the dependencies that are introduced due to interactions.

      ii. **Program expressions vs variable assignments as causes** (*What constitutes a causal sequence?*: process-oriented view). We focus on identifying relevant program expressions in a process as opposed to pinpointing an entire process or its outcome. Our fine-grained causal determination is useful for accountability.

      iii. **Testing counterfactual scenarios** (*How to identify a causal sequence?*). In our process-oriented theory, counterfactuals are programs[11] instead of variable-value pairs as in prior work. These programs are constructed in a specific man-

[11] i.e. absence or presence of program subexpressions

ner from the programs that actually executed. They are useful to identify the causal sequence for a violation, i.e. to identify parts of the programs that actually caused the violation and eliminate other parts that were irrelevant for the violation. This approach to constructing counterfactual scenarios allows us to retain the interaction structure of the log and not remove individual events, which could lead to spurious causal inferences. In prior work, counterfactual scenarios are constructed by modifying the value of any variable in the model and propagating the effect to its dependent variables. In our case, modifying the value of a dependent variable in a causal sequence, imposes constraints on all the variables in the causal sequence. This distinction allows us to provide a more fine-grained causal sequence as opposed to specifying the relevant outcome value.

Modeling:

(a) **Using process-calculus.** Our motivation stems from providing accountability for security protocols where modeling interaction and capturing non-deterministic settings is a key component. We focus on causal sequences and processes where capturing ordered sequences of events is relevant. Process calculus frameworks provide natural interaction primitives which have been used to model and reason about protocol settings extensively and allow us to focus specifically on processes and interactions. Hence process calculus frameworks make a natural choice for our formalization. Existing formalisms in actual causation literature lack operators to naturally capture such constructs.

Consequences:

(a) Definitional: **Distinguishing between joint and independent causes.** A joint cause refers to all the individual program expressions (possibly from different processes) which are part of a causal sequence. Independent causes of a violation occur when, intuitively, two distinct sequences of program expressions have an equal claim to be regarded as the cause of an effect[12]. Our definition cleanly handles this separation which is a recognized challenge for prior actual cause definitions [14, 17, 27, 30, 48].

(b) Modeling: **Expressing non-deterministic interacting systems concisely.** Our process calculus-based framework can express general models for non-deterministic interacting systems more concisely than the formalism used in prior work in counterfactual-

[12]This is related to the idea of symmetric overdetermination from prior work [47].

based actual causation[13]

(c) Modeling: **Handling preemption examples concisely.** In contrast with independent causes, preemption occurs when one cause has a preference and the other cause 'merely waits in reserve' [49], i.e., both causes are not considered symmetric or equivalent. Our definition can handle preemption with concise representation by using existing constructs in our formalism, as compared to prior frameworks[14].

2. **Contributions to accountability, security and privacy:**

This dissertation makes a fundamental contribution by identifying and formalizing actual causation as the basis via which one can link relevant deviations to violations for assigning blame. We demonstrate the value of our formalism for the domain of security and privacy protocols in two ways. First, we prove that violations of a precisely defined class (subset) of safety properties always have an actual cause. Thus, our definition applies to relevant security properties. Second, we provide a cause analysis of a representative protocol designed to address weaknesses in the current public key certification infrastructures [50]. Our theory clearly distinguishes between deviances and actual causes which is important from the standpoint of accountability.

In addition, we discuss how our framework can serve as a building block for causal explanations (protocol debugging) and exoneration (i.e., soundly identifying agents who should not be blamed for a violation).

Scope of work. In this dissertation, we focus on actual causation, which, in our setting, corresponds to finding causes of violations on a single log of events[15]. In particular, we consider counterfactual based theories of actual causation in prior work. We motivate our framework in the larger context of providing a building block for accountability, i.e. blame assignment and explanations. We recognize that blame-assignment goes beyond deviance and actual cause determinations — identifying how to combine deviance and actual cause determinations for accurate blame-assignment remains an open question.

Causal analysis is an intuitive building block for answering some very natural questions that have direct relevance to accountability such as (i) *why* a particular violation occurred, (ii) *what* component in the protocol is blameworthy for the violation and (iii) *how* the protocol could have been designed differently to preempt violations of this sort. Answering these questions requires an in-depth study of, respectively, explanations, blame-assignment, and protocol

[13] i.e. structural-equation based frameworks used in actual causation [14, 17, 18, 30].
[14] i.e. structural-equation based frameworks used in actual causation [14, 17, 18, 30].
[15] This is in contrast with inferring more general relations.

design, which are interesting problems in their own right, but are not the explicit focus of this dissertation. Instead, we focus on a formal definition of causation that we believe formal studies of these problems will need. Roughly speaking, explanations can be used to provide an *account* of the violation, *blame assignment* can be used to hold agents *accountable* for the violation, and protocol design informed by these would lead to protocols with better accountability guarantees.

General applicability of these ideas. The applicability of ideas proposed in this thesis goes beyond security protocols and philosophy – the analysis techniques proposed in this dissertation are relevant for reasoning about causation in any non-deterministic interacting system. Our framework naturally captures choice and interaction between communicating processes, and finds a sequence of actions implementing these constructs, as an actual cause of an effect.

Our analysis is applicable to fields where the processes (modeled via programs) generating an outcome are not fixed, and their execution might be non-deterministic. For instance, our analysis is applicable in fields such as engineering and artificial sciences [51], or in any non-deterministic system which involves (human) choice and interaction where agents have a choice to execute different programs. Hence we focus on finding a causal sequence containing program expressions, as opposed to instantiated variable values for process outcome. This is in contrast with the natural sciences, where the processes generating outcomes are stable and the focus is on understanding how the outcomes of specific processes contribute to a global effect. Ideas in actual causation are of relevance for blame exoneration or assigning responsibility [52], legal reasoning for tort and criminal cases [42], fault diagnosis [53], as well as for different applications in AI such as finding explanations, troubleshooting and prediction [27].

1.6 Structure of the dissertation

The rest of the dissertation is split into two parts.

Part 1 develops an interaction-aware theory of actual causation.

- Chapter 2 provides a detailed overview of our process calculus framework. We describe how we can encode structural equations using our process calculus framework, with some examples.
- In Chapter 3, we provide an interaction-aware definition of actual causation. We demonstrate how to construct counterfactual scenarios in an interaction-aware manner, for our protocol-based settings and justify other design choices.
- In Chapter 4, we relate our definition to process causation theories and four prominent

definitions for counterfactual-based actual causation in AI and Philosophy. We highlight distinctive features of our formalism. Additionally, we consider examples which have been problematic for prior theories, including preemption examples, distinguishing between joint and independent causes as well as over-permissive counterfactual-based analyses. We encode these examples in our process calculus framework and highlight the differences in the approaches.

Part 2 presents an instance of the interaction-aware theory of actual causation and applies it to the setting of security protocols. This part finds program actions as actual causes using an interaction-aware theory. The definition proposed in Chapter 3 is a generalization of the definition presented in this part.

- In Chapter 5, we formally introduce our model and provide a definition which finds program actions as actual causes of violations. Section 5.1 describes a representative example which we use throughout the chapter to explain important concepts. We formalize program actions as causes with an interaction-aware approach, developed in Part 1.
- In Chapter 6, we discuss how our definition can be used as a building block for explanations and blame assignment, and hence, accountability. We provide examples showing that not all deviants are actual causes and vice versa.
- Chapter 7 summarizes the dissertation and concludes with interesting directions for future research.

Remark. Chronologically, the theory presented in the second part of this dissertation was developed before the general theory presented in Part 1. Part 2 was developed specifically for the domain of security protocols and provides relevant analysis methods and a case study. In Chapter 5, we discuss how the definition developed in Part 2 is related to the definition presented in Part 1.

Some parts of this dissertation appear in a conference paper in the proceedings of the IEEE 28th Computer Security Foundations Symposium (Verona, Italy, July 2015) as *'Program Actions as Actual Causes: A Building Block for Accountability'* [54] and in the full version of that paper [55].

Part I

Interaction-aware Theory of Actual Causation

Chapter 2

Overview of the Formalism

Chapter goal. In this chapter, we introduce the process calculus formalism. We introduce the syntax and operational semantics (step-wise execution). Next, we briefly describe the syntax and semantics for the structural equation framework used in actual causation literature [14, 16, 17, 18, 29, 56]. We demonstrate the correspondence and describe why process calculus is better suited for modeling interactions and processes. We demonstrate how examples modeled using structural equations can be encoded in our framework.

2.1 Syntax

In the next part we introduce the syntax for our process calculus.

Describing examples from prior work. In this chapter, we use examples from prior work in actual causation. For each example, we provide the following details:

- The high-level description of the example.
- The encoding of the example in our process calculus framework: The processes are marked with a subscript P i.e. the process corresponding to an entity A is given by A_P in our process calculus encoding. Note that an entity can execute multiple programs and these can be distinguished in our formalism since each program will have a unique identifier[1]. For this part of the dissertation, unless specified, A_P refers to the unique program executed by entity A.

[1]Our causal analysis only focuses on the programs and not the entities executing the programs, hence a single entity can execute multiple programs.

Encoding examples using process calculus framework. As mentioned in the introduction, we are interested in understanding which choice values and which expressions in the processes are a cause of the outcome. Specifically, we are interested in pinpointing the relevant parts of the process as opposed to the entire process. For this purpose, we need to reason about various components of a process. Therefore, instead of focusing on variable assignments for process outcomes, we focus on modeling the processes and their interactions.

Let us consider an example from prior work in actual causation, suggested by Hopkins [48].

Example 1 (Loader)**.** *Two shooters A and B both shoot at a target H simultaneously. Another agent E loads the gun for A before A shoots. We are interested in finding the cause of the target being hit.*

For this example, we assume that there is a process for each of E, A, B and H which is given by E_P, A_P, B_P and H_P respectively. We define a function Σ which records the substitutions for all the free/ input variables in each of these processes. Each process has an identifier in the set $\mathcal{I}$ (for instance A, B, H in the above example) and $\mathcal{P}$ is a finite set of programs executed by E, A, B, H (for instance E_P, A_P, B_P, H_P in the above example). Therefore an *initial configuration* of a system can be described by the triple $\langle I, \mathcal{P}, \Sigma \rangle$.

In order to write these programs, we introduce a simple syntax for communication with `send` and `recv` (receive) primitives. For instance, for a program E_P interacting with another program A_P, we write $\texttt{send}(A_P, m)$ to indicate that the message m was intended for A_P. Similarly, in A_P's program we write $\texttt{recv}(E_P)$ to indicate that the message m was received from E_P. Our communication primitives are untargeted which is important for modeling security protocols, as well as expressing general models concisely. We also introduce a simple term language for variables and for performing boolean operations over variables such as conjunction or disjunction. We introduce an assignment operator, similar to the one used in structural equations, in order to assign the outcome of boolean operations to specific variables. For instance, the value received by E_P can be assigned to a variable m' using the following program expression: $m' = \texttt{recv}(A_P)$.

For our running example, the process E_P sends a value (say e) to A_P where the value denotes whether or not E loads the gun for A. A_P receives a value e from E's program and then sends a value (say s) to H_P where the value denotes whether or not A shoots. Note that s depends on both the value received from E_P and whether or not A decides to shoot. Similarly, B_P sends a value (say b) to H_P (the value denotes whether or not B shoots). H_P receives both values and uses these values to decide whether or not the target is hit (say $h = 1$ or $h = 0$).

Preserving temporal ordering and dependencies. In order to express multiple expressions in a single program which execute in a specific order, we introduce a sequencing operation. Unlike the process for B, A is not at liberty to shoot since the gun must have been loaded by E before A can shoot. We can capture such dependencies using a conditional operator ($?\ :$) in our term language[2].

$c\ ?\ d:\ g$ denotes that *if* c *then* d *else* g. Therefore if a indicates whether or not A decides to shoot and e indicates whether or not E loaded the gun, then we can say that whether or not A actually shoots (denoted by s) depends on:

$s = (e == 1)?\ a:\ 0$

This ternary operator is evaluated in the following manner: *if* the value of e is 1, *then* $s = a$ *else* $s = 0$, i.e. if E loads the gun, then A can choose to shoot or not shoot, however if E did not load the gun, then A cannot shoot.

Here is how the programs described above will look:

E_P	A_P	B_P	H_P
$\texttt{send}(A_P, e);$	$e = \texttt{recv}(E_P);$		
	$s = (e == 1)?\ a:\ 0;$		
	$\texttt{send}(H_P, s);$		$s = \texttt{recv}(A_P);$
		$\texttt{send}(H_P, b);$	$b = \texttt{recv}(B_P);$
			$h = s \vee b;$

In this case, we have aligned the communication actions across different programs, so that the readers can see how the `send` and `recv` actions are paired across different programs, as well as fixed the scheduling. A_P receives a value from E_P, H_P receives from A_P first, and then from B_P in this scenario – in Section 2.4, we describe how a more general model can be expressed.

Similarly, we can denote other boolean operations on the terms. For instance, in order for the target to be hit, if both A and B had to shoot, then we can model the scenario by replacing $h = s \vee b$ with $h = s \wedge b$ in Example 1. Let us look at another example which involves conjunction.

Example 2 (Forest fire – conjunctive scenario)**.** *A forest fire F is caused when* both *lightning strikes (L) and a match is lit (M) by an arsonist. If both the lightning strike and the arsonist drops a lit match, and the forest burns– what is the cause?*

Here is how the programs will look:

[2]This is a common operator used in the C language.

M_P	L_P	F_P
$\texttt{send}(F_P, u_M);$	$\texttt{send}(F_P, u_L);$	$m = \texttt{recv}(F_P);$ $l = \texttt{recv}(L_P);$ $f = m \wedge l;$

In this setting, M_P sends a value m (fixed as part of its background) to F_P. Similarly, L_P sends a value l to F_P. F_P receives both values and computes the conjunction.

2.1.1 Adding choice and asymmetric disjunction

Since, we are interested in assigning blame, we would like to distinguish between inputs which are fixed as part of the background, and variables over which agents have a choice. Additionally, if symmetric operators such as disjunction are used to evaluate output values, then the disjunction could be resolved due to any of the variables being true. For instance if $h = a \vee b$ and $a = b = 1$, then h could be 1 due to either value being 1. In cases where additional information is available, we would like to distinguish between which of the disjuncts was evaluated in the actual context.

We introduce a choice operator, $\oplus$, in order to model an internal choice made within the programs. For the second case mentioned above, we introduce an asymmetric disjunction operator $[]$, which is used to indicate that the disjunct via which the evaluation resulted in 1 is relevant and evidence capturing the evaluation is available. This is especially relevant for a class of preemption based examples (Section 4.4.3). We explain both the constructs in detail below.

Internal choice. In our example, writing $a = 0 \oplus 1$ indicates that the agent executing program A_P can either choose a to be 0 or 1, i.e. A can choose to shoot or not.

Here is how the programs will look for Example 1 if we explicitly model the choice over the initial variable values. Note that since we assume boolean variables for this example, therefore $s = (e == 1)?\ a :\ 0;$ can be encoded as $s = e \wedge a$[3].

[3]The conditional operator is especially useful if the variables are not boolean.

E_P	A_P	B_P	H_P
$e = 0 \oplus 1;$	$a = 0 \oplus 1;$	$b = 0 \oplus 1;$	
$\texttt{send}(A_P, e);$	$e = \texttt{recv}(E_P);$		
	$s = e \wedge a;$		
	$\texttt{send}(H_P, s);$		$s = \texttt{recv}(A_P);$
		$\texttt{send}(H_P, b);$	$b = \texttt{recv}(B_P);$
			$h = s \vee b;$

Notice that here we express both choices in the model. This is because we want our model of programs to be as general as possible in terms of expressing the values that different variables can take.

Asymmetric disjunction operator. $[]$ is used to model an asymmetric disjunction operator, which also records the specific disjunct which led to the value being 1. A symmetric disjunction operator $\vee$ does not model how the disjunction is satisfied when multiple disjuncts are true. However in certain cases, causal inference requires recording the path via which a violation happened. This is especially useful in cases such as preemption (Section 4.4.3) where a violation could potentially occur due to multiple causes on the log, however only one of the processes is responsible.

In order to understand the usage of $[]$, consider an alternative description of the running example (Example 1) with a slight modification: only one of the shooters can hit the target. Temporal asymmetry or other factors might be involved in determining which of the bullets is relevant– we can express this asymmetry via the asymmetric disjunction operator[4]. In this case, here is how the programs look for Example 1. For notational convenience, we add line numbers for all program expressions:

E_P	A_P	B_P	H_P
$1 : e = 0 \oplus 1;$	$1 : a = 0 \oplus 1;$	$1 : b = 0 \oplus 1;$	
$2 : \texttt{send}(A_P, e);$	$2 : e = \texttt{recv}(E_P);$		
	$3 : s = e \wedge a;$		
	$4 : \texttt{send}(H_P, s);$		$1 : s = \texttt{recv}(A_P);$
		$2 : \texttt{send}(H_P, b);$	$2 : b = \texttt{recv}(B_P);$
			$3 : h = s \,[]\, b;$

[4]In cases where such information is not relevant or not available, the symmetric disjunction can be used.

The asymmetric disjunction evaluates the outcome in the same manner as a symmetric disjunction, except that it additionally records which of the disjuncts was evaluated when both disjuncts were true.

Implications for the modeler. A disjunction can be modeled as $h = a \vee b$ or $h = a \,[]\, b$. One of the considerations while choosing how to model an example, is to consider whether it only matters *what* the final value of h is or whether it also matters *how* the value of h was obtained. This distinction is of significance when we consider cases like symmetric overdetermination as opposed to preemption [57]. If the focus is only on the final value of the equation, then there could possibly be independent causes in our framework. However, if the specific manner in which h was evaluated is critical, in that case our framework will only find one of the disjuncts as relevant – in particular since we give a definition of actual causation so we will find the *disjunct evaluated on the log* as relevant. If both a and b are 1 in the equation $h = a \,[]\, b$, then the actual sequence of events will note how the value was obtained. This design choice depends on the amount of information available regarding the actual context, which can be used to give a more fine grained causal analysis. Otherwise, using the symmetric disjunction will find both the relevant sequences which lead to $a = 1$ and $b = 1$ as independent causes.

The semantic interpretation for the symmetric and asymmetric disjunction operators is different, since they give rise to different execution sequences. In the next section, we describe how each of these operators can be evaluated.

Syntax. Before describing the semantics, we give the final syntax. We model programs in a simple language that can represent concurrent processes[5]. The language contains sequential expressions, e, that execute concurrently in programs and communicate with each other through `send` and `recv` commands as discussed in this section.

Our syntax is given using the A-normal form [58] where every term contains only one connective and all operands contain only variables. The syntax consists of values v for variables x, actions α and expressions e. Values v include boolean values, numerical values and all other return values (such as keys or cipher text). Variables, x, denote messages that may be passed through expressions or across programs.

An expression is a sequence of actions, α. An action may do one of the following: execute a primitive function on values $v_1, v_2, \ldots$, or send or receive a message to another program; (writ-

[5]For the purpose of this part of the dissertation, we limit attention to this simple expression language, without recursion or branching. Our definition of actual cause is general and applies to any formalism of (nondeterministic) interacting agents, but the auxiliary definitions of log projection and the function dummify introduced later must be modified.

ten $\texttt{send}(v)$ and $\texttt{recv}()$, respectively). For our examples in the first part of the dissertation, we only use the binary operators shown below for α, however in a more general case, it can be used for other side effect free actions [6]. Note that depending on the kind of variables, we could introduce additional operations other than the boolean connectives (See Example 4).

$$
\begin{array}{lrcl}
\text{Values} & v & ::= & x \mid \texttt{true} \mid \texttt{false} \mid 1 \mid 2 \ldots \\
\text{Actions} & \alpha & ::= & v \mid \texttt{send}(v) \mid \texttt{recv}() \mid v_1 \wedge v_2 \mid v_1 \vee v_2 \mid \\
 & & & \neg v \mid v_1 ?\, v_2 :\, v_3 \mid v_1 \oplus\, v_2 \mid v_1 [] \, v_2 \mid \ldots \\
\text{Expressions} & e & ::= & v \mid (b : x = \alpha); e_1 \mid \texttt{assert}(v); e_1
\end{array}
$$

Each expression e is labeled with a unique line number, written b. Line numbers help define traces later. Every action and expression in the language evaluates to a term and potentially has side-effects. The term returned by action α is bound to x in evaluating e_2 in the expression $(b : x = \alpha); e_2$.

In our model, send and recv are untargeted in the operational semantics: A message sent by a program A_P can be received by any other program. The first argument to send and recv specifies the *intended* target. Action $\texttt{send}(v)$ always returns 0 to its continuation. When the communication pair is clear from context, we directly mention the receiver for the send action and the sender for the recv action. In security settings, we consider more general settings with different synchronizations. For the examples in this part, we restrict the synchronizations in order to model the examples given in structural equation framework.

We abbreviate $(b : x = \alpha); x$ to $b : \alpha$ and $(b : x = \alpha); e$ to $(b : \alpha); e$ when x is not free in e. As an example, the following expression receives two messages, computes a new value (through a boolean operator $\vee$) and sends the resulting message to M_3.

$$
\begin{array}{l}
1 : m_1 = \texttt{recv}(M_1); \;\; //\text{receive message, bind to } m_1 \\
2 : m_2 = \texttt{recv}(M_2); \;\; //\text{receive message, bind to } m_2 \\
3 : n = m_1 \vee m_2; \;\; //\text{generate term, bind to } n \\
4 : \texttt{send}(M_3, n); \;\; //\text{send } n \text{ to } M_3
\end{array}
$$

Next we turn our attention to expressing a particular execution of the model that we would like to analyze for causes. In Example 1, even though the model is general and allows us to express different executions, we need to express an unambiguous execution of the model for causal analysis. Once we consider *interacting processes*, this step requires fixing various *synchronizations* across processes as well as the internal choices. In the next part, we discuss how to formalize an execution of the system we modeled in our process calculus framework.

[6] Refer to Chapter 5

2.2 Operational semantics for process calculus framework

In this section, we describe the execution model and rules for operational semantics (or step-wise execution) for our syntax. Details of the complete syntax and semantics for our process calculus framework can be found in the Appendix A.

Non-determinism in the execution semantics. In the previous section, we briefly alluded to the non-determinism in the execution semantics of our process calculus. There can be several sources of non-determinism in process execution:

- the order of execution across different programs could lead to a different sequence of events. Processes can execute their internal or communication actions in any order (as long as each process respects its internal sequential order), leading to multiple possible interleaving of actions across programs.
- the initial state might be one of the several possible alternatives, this could be due to externally provided inputs or internal choices in programs leading to a different execution.
- The evaluation of asymmetric disjunction operator (if any) could lead to a different execution of the system[7].

The rules for step-wise evaluation of the expressions in our language are called *operational semantics*, which we describe next. The operational semantics define how a collection of programs execute simultaneously. For instance, if a program sends a message (i.e. executes a `send` action), the operational semantics provide rules for how the action gets evaluated and how it changes the relevant variable values. The operational semantics provide us with a formal and rigorous process for discussing different states of the system as expressions are evaluated. We then turn our attention to formally defining configurations, logs and labels.

Configurations. A configuration $\mathcal{C} = \langle \mathcal{I}, \mathcal{P}, \Sigma \rangle$ records which program is currently executing which action and how the variable values gets updated. In the sequel, we identify the triple $\langle I, \mathcal{P}, \Sigma \rangle$ with the configuration defined by it. We also use a configuration's identifiers to refer to its programs. When the system is in a certain configuration, and an action is executed, we say that the current configuration $\mathcal{C}$ *reduced* to a new configuration $\mathcal{C}'$. The reduction relation is denoted by $\mathcal{C} \rightarrow \mathcal{C}'$. The rules for evaluating specific expressions in our language can be found in the appendix.

[7]If symmetric operators such as disjunction are used to evaluate output values, then the disjunction could be resolved due to any of the variables being true.

For our running example, the initial configuration would be $\langle I, \mathcal{P}, \Sigma \rangle$ where $I = \{A, B, E, H\}$, $\mathcal{P}$ is given in Figure 2.1. If we do not provide an internal choice over variables a, b, e, then Σ will contain the ground values for these variables. We can think of Σ as part of the background context which is fixed, and not decided via program's internal choices.

We denote the execution of a specific line number b within a program i by a tuple $\langle i, b \rangle$ called its label. Annotating the reduction arrow with a *label* r makes the locus of a reduction explicit. This is written as $\mathcal{C} \xrightarrow{r} \mathcal{C}'$. We define the labels in more detail below.

Labels. In Example 1, if we model the programs without the choice operators, here is how the programs will look:

E_P	A_P	B_P	H_P
$1 : \mathtt{send}(A_P, e);$	$1 : e = \mathtt{recv}(E_P);$		
	$2 : s = e \wedge a;$		
	$3 : \mathtt{send}(H_P, s);$		$1 : s = \mathtt{recv}(A_P);$
		$1 : \mathtt{send}(H_P, b);$	$2 : b = \mathtt{recv}(B_P);$
			$3 : h = s \vee b;$

Since each program expression has a line number, we can use the tuple $\langle$ *Program identifier, Line number* $\rangle$ to identify the expressions being evaluated. For instance, if we want to discuss the evaluation of H_P computing the value of h at line 3, we can write it as $\langle H_P, 3 \rangle$. We call such a tuple as a *label*. Note that the label only indicates which program and which expression were evaluated; it does not indicate the instantiation or final value of h. If we write the programs with internal choice, we will need to additionally specify the choice resolution in the labels i.e. $\langle E_P, 1, \mathtt{r} \rangle$, $\langle A_P, 1, \mathtt{r} \rangle$ and $\langle B_P, 1, \mathtt{r} \rangle$ in order to represent a specific execution.

E_P	A_P	B_P	H_P
$1 : e = 0 \oplus 1$	$1 : a = 0 \oplus 1$	$1 : b = 0 \oplus 1$	
$2 : \mathtt{send}(A_P, e);$	$2 : e = \mathtt{recv}(E_P);$		
	$3 : s = e \wedge a;$		
	$4 : \mathtt{send}(H_P, s);$		$1 : s = \mathtt{recv}(A_P);$
		$2 : \mathtt{send}(H_P, b);$	$2 : b = \mathtt{recv}(B_P);$
			$3 : h = s \vee b;$

More formally, we denote the execution of a specific line number b within a program i as a tuple $\langle i, b \rangle$. We call this as a *label* of a reduction. A label can be one of the following:

- an *internal* label which indicates the program i and the line number b that executed, i.e. $\langle i, b\rangle$. If the expression contains the internal choice operator $\oplus$ or the asymmetric disjunction ([]), then additionally the label indicates which way the choice was resolved `l` (left) or `r` (right) or which of the disjuncts was evaluated (if both are true)[8].
- a *synchronization* label which indicates how a `send` action in one program synchronized with the corresponding `recv` action along the specified channel. For instance, if A_P sends a message to H_P and b, b' are the respective line numbers in the programs for A_P and H_P, then the label will be denoted by $\langle\langle A_P, b\rangle, \langle H_P, b'\rangle\rangle$.

Traces and logs. We call a finite sequence of reductions as a *trace*. Traces are denoted with the letter t and contain information about the labels as well as the substitutions. Note that the trace contains instantiated values for all variables and actions.

We are interested in finding choices and *uninstantiated program expressions* in the syntax as a cause. Since a trace contains additional information which is not required for our causal analysis, we find it convenient to introduce the notion of a *log*.

Definition 1 (Log). *Given a trace t, the log of the trace, $\log(t)$, is the sequence of labels on the trace, i.e. $r_1, \ldots, r_m$ where r_i can be an internal label or a synchronization label.*

In other words a log denotes the sequence of uninstantiated expressions in an execution. The letter l denotes logs[9].

For instance, one log for the given set of interacting programs (without choice operator) is $\langle E_P, 1\rangle, \langle\langle E_P, 2\rangle, \langle A_P, 1\rangle\rangle, \langle A_P, 2\rangle, \langle\langle A_P, 3\rangle, \langle H_P, 1\rangle\rangle, \langle\langle B_P, 1\rangle, \langle H_P, 2\rangle\rangle, \langle H_P, 3\rangle$. Here, the temporal ordering indicates that the expression labeled 1 in E_P's program is executed. Next, an interaction occurs between E_P and A_P. Similarly, the expressions for B_P and H_P are also indicated. Note that we represent the sequence of labels above such that the corresponding send and receive labels are paired to indicate synchronization between the respective program expressions. Therefore, instead of writing $\langle E_P, 1\rangle$ and $\langle A_P, 1\rangle$ separately, we represent the synchronization of the send and receive actions by writing these labels jointly as:

$\langle\langle E_P, 1\rangle, \langle A_P, 1\rangle\rangle$.

Given $a = 1, b = 1, e = 1$, we can compute the value of h by considering the log. Here, the value of $h = 1$.

Given a set of interacting programs, there can be several traces which differ in their *order of actions* and initial values. Consider the values of a and b chosen in the programs in Figure 2.1:

[8]Note that we can handle more general cases as well via nesting the recorded choices. See Example 7, Section 4.4.3.1.

[9]Here we assume access to all programs and logs.

the value of h will vary accordingly. For instance, if the value of a and b are both chosen as 0, then keeping the same interaction structure will still yield a different value for the variable as shown in Figure 2.1. Here, we denote the log assuming that E loads the gun, and both A and B choose to shoot, i.e. the internal choice is resolved as $a = b = e = 1$.

Note that in Figure 2.1, the log fixes the interleaving of actions across programs for E_P, A_P, B_P and H_P. Consider an alternative log in which the order in which E, B, A make their internal choices is different. We would like our causal determination to be independent of the relative interleaving of internal events. Therefore, we define *projection of a log* below.

E_P	A_P	B_P	H_P	LOG l
1: $e = 0 \oplus 1$;				$\langle E_P, 1, \mathtt{r}\rangle$
	1: $a = 0 \oplus 1$;			$\langle A_P, 1, \mathtt{r}\rangle$
		1: $b = 0 \oplus 1$;		$\langle B_P, 1, \mathtt{r}\rangle$
2: $\mathtt{send}(A_P, e)$;	2: $e = \mathtt{recv}(E_P)$;			$\langle\langle E_P, 2\rangle, \langle A_P, 2\rangle\rangle$
	3: $s = e \wedge a$;			$\langle A_P, 3\rangle$
	4: $\mathtt{send}(H_P, s)$;		1: $s = \mathtt{recv}(A_P)$;	$\langle\langle A_P, 4\rangle, \langle H_P, 1\rangle\rangle$
		2: $\mathtt{send}(H_P, b)$;	2: $b = \mathtt{recv}(B_P)$;	$\langle\langle B_P, 2\rangle, \langle H_P, 2\rangle\rangle$
			3: $h = s \vee b$;	$\langle H_P, 3\rangle$

E_P	A_P	B_P	H_P	LOG l'
	1: $a = 0 \oplus 1$;			$\langle A_P, 1, \mathtt{r}\rangle$
		1: $b = 0 \oplus 1$;		$\langle B_P, 1, \mathtt{r}\rangle$
1: $e = 0 \oplus 1$;				$\langle E_P, 1, \mathtt{r}\rangle$
2: $\mathtt{send}(A_P, e)$;	2: $e = \mathtt{recv}(E_P)$;			$\langle\langle E_P, 2\rangle, \langle A_P, 2\rangle\rangle$
	3: $s = e \wedge a$;			$\langle A_P, 3\rangle$
	4: $\mathtt{send}(H_P, s)$;		1: $s = \mathtt{recv}(A_P)$;	$\langle\langle A_P, 4\rangle, \langle H_P, 1\rangle\rangle$
		2: $\mathtt{send}(H_P, b)$;	2: $b = \mathtt{recv}(B_P)$;	$\langle\langle B_P, 2\rangle, \langle H_P, 2\rangle\rangle$
			3: $h = s \vee b$;	$\langle H_P, 3\rangle$

Figure 2.1: Example 1: Two different logs l, l' for the same set of programs

Projections of a log. Given a log l and a program i, the *projection* of l to i, written $l|_i$ is the subsequence of all labels in l that mention i i.e. $l|_i$ contains all actions executed by program i on log l. We call a log l' a *projected sublog* of the log l, if for every program i dropping some labels from $l|_i$ results in $l'|_i$. More formally, for every i, the sequence $l'|_i$ is a subsequence of the sequence $l|_i$.

The definition of projected sublog allows the relative order of events in two different non-communicating programs to differ in logs l and l' but Lamport's happens-before order of actions [59] in l' must be preserved in l. This is important, for instance when the internal actions for the shooter A and B are evaluated in a specific order on the log, however, their relative

order w.r.t. each other is not crucial.

Note that for logs l and l' in Figure 2.1, even though the logs differ, the projections for both the logs coincide. The definition of a projected sublog allows us to collectively analyze the causes of a set of logs, rather than a single one. The projections for both logs are given below. Here t represents the concrete trace from which we obtain the log.

$\boxed{\log(t)|_E}$
$\langle E_P, 1, \mathbf{r}\rangle$
$\langle\langle E_P, 2\rangle, \langle A_P, 2\rangle\rangle$
$\boxed{\log(t)|_A}$
$\langle A_P, 1, \mathbf{r}\rangle$
$\langle\langle E_P, 2\rangle, \langle A_P, 2\rangle\rangle$
$\langle A_P, 3\rangle$
$\langle\langle A_P, 4\rangle, \langle H_P, 1\rangle\rangle$
$\boxed{\log(t)|_B}$
$\langle B_P, 1, \mathbf{r}\rangle$
$\langle B_P, 2\rangle$
$\langle\langle B_P, 3\rangle, \langle H_P, 2\rangle\rangle$
$\boxed{\log(t)|_H}$
$\langle\langle A_P, 2\rangle, \langle H_P, 1\rangle\rangle$
$\langle\langle B_P, 2\rangle, \langle H_P, 2\rangle\rangle$
$\langle H_P, 3\rangle$

2.3 Structural equation framework for actual causation

Structural Equation Modeling (SEM) based techniques have been extensively used in several fields such as economics, statistics etc [26, 27, 29][10]. In recent years, this framework has been used to model causal networks. Here we describe the framework as used by actual causation literature.

Causal models. Prior work has a long tradition of defining actual cause using causal models, which are used to encode counterfactual relationships amongst variables [27, 29, 56]. A structural model signature $\mathcal{S} = \{\mathcal{U}, \mathcal{V}, \mathcal{R}\}$ where $\mathcal{U}$ is called the set of exogenous or background variables. This represents the set of background variables or the context that is considered

[10]Originally path analysis by using SEMs was intended to focus on interpreting data rather than solving causation. Each equation in the framework represents a causal relationship between a set of variables and the form of the equation demonstrates the assumptions made by the modeler. In particular, some of the assumptions include that the variables are measured without any error. Using the data, these models derive quantitative causal conclusions and statistical measures to assert whether the assumptions are reasonable.

external to the causal process. $\mathcal{V}$ is the set of endogenous variables that corresponds to the conditions that are potential causes. These variables can be affected by the background variables and other endogenous variables. $\mathcal{R}$ represents the range of permissible values for both of these sets, respectively. A causal model M is an ordered pair $\langle \mathcal{S}, \mathcal{F} \rangle$ where $\mathcal{S}$ represents a structural model signature and $\mathcal{F}$ gives a set of modifiable structural equations which specify the relationships amongst these variables.

Notation. Throughout this dissertation, when we talk about endogenous variables as used in prior work, we indicate them with subscript of R, for instance, an event A being 0 or 1 will be represented as A_R to indicate that we are referring to the random variable for structural equations. The exogenous variables are denoted with subscripts of u (for instance u_A, u_B) to indicate that they are a part of the context. In our process calculus framework, when we talk about a process for an identifier A, we denote it by A_P. When we talk of variables in our process calculus framework, we simply write those as lowercase letters $a, b, \ldots$.

Structural equations are used to model the effect of causal influence of some background variables and endogenous variables on other endogenous variables. The structural equations are similar in spirit to assignment statements in programming languages. The left hand side denotes an endogenous variable whose value is given by an expression which contains other endogenous variables and exogenous variables. These equations are asymmetric in nature. For instance if A_R, M_R are variables in $\mathcal{V} \in \mathcal{S}$ (we denote the random variables in structural equation framework with the subscript R) and u_L is a background variable in $\mathcal{U}$ then $A_R \leftarrow M_R \vee \ L_R$ is a structural equation which describes the effect of M_R and L_R on A_R. However a change in the value of a variable on the left hand side (A_R) does not affect the values of the variables on the right hand side (M_R, L_R). These structural equations can encode counterfactual relations since the equation predicts the value of A_R above when M_R and L_R are instantiated to different values.

2.3.1 Semantics for structural equations

The semantics for structural equations, as used in actual causation has been discussed in prior work [14, 17]. Here we give the relevant details. Given a signature $\mathcal{S}$, a primitive event in a structural equation framework is defined as specific assignment for a random variable $X \in \mathcal{V}$, for instance $X = x$ where $x \in \mathcal{R}(X)$. Then, a causal formula is of the form $[Y_1 \leftarrow y_1, Y_2 \leftarrow y_2, \ldots]\varphi$ where φ is a combination of primitive events and Y_i's are distinct endogenous variables in $\mathcal{V}$. Additionally $y_i \in R(Y_i)$ [14]. An evaluation of a causal formula is obtained by setting a particular context i.e. a specific assignment for the exogenous variables in $\mathcal{U}$. Once we assume

an assignment for variables in $\mathcal{U}$, then we can always find a unique solution for the structural equations[11].

The equations in $\mathcal{F}$ can also be represented as directed acyclic graphs, where the variables form the nodes and the edges are formed in the following manner: a variable is the parent of another if it appears on the right hand side in its structural equation (each variable appears on the left hand side in only one equation). For a more detailed overview of structural equations, see structural equations framework [14, 16, 17, 29, 48].

The structural equations, as used in the above-described manner are deterministic in nature. Therefore, evaluating any one variable, given its parents, would always lead to the same instantiation. Moreover, the order in which the variables are evaluated, does not affect the final instantiation since the corresponding graph is acyclic. Therefore an evaluation of the system is completely specified by the assignment to the exogenous variables as well as the structural equations.

Let us consider the running example again.

Example 1, Loader

Two shooters A and B both shoot at a target H simultaneously. Another agent E loads the gun for A before A shoots. We are interested in finding the cause of the target being hit.

In this case assume that an exogenous variable E_R models whether or not E loads the gun for A. Similarly A_R models whether or not A shoots and the random variable B_R models whether or not B shoots. Then we would express the model by the following structural equations:

$$\begin{aligned} A_R &= u_A \\ B_R &= u_B \\ E_R &= u_E \\ H_R &= (A_R \wedge E_R) \vee B_R \end{aligned}$$

Here $\{u_A, u_B, u_E\}$ are exogenous variables in $\mathcal{U}$. $\{A_R, B_R, E_R, H_R\}$ represent the endogenous variables in $\mathcal{V}$. The structural equations above denote the influence of exogenous and endogenous variables.

Let us look at another example which involves conjunction.

Example 2, Forest fire – conjunctive scenario

A forest fire F is caused when *both* lightning strikes (L) and a match is lit (M) by an arsonist. If both the lightning strike and the arsonist drops a lit match, and the forest burns– what is the cause?

[11] provided the system of equations is recursive

Here F_R is a random variable that models whether or not a fire occurs. M_R is a random variable that captures if the match is lit and L_R captures whether lightning strikes or not. $\{u_M, u_L\}$ are exogenous variables $\mathcal{U}$ in the context.

In this case, the structural equations will be:

$$\begin{aligned} L_R &= u_L \\ M_R &= u_M \\ F_R &= M_R \wedge L_R \end{aligned}$$

In our model, we can encode the processes which instantiate these three random variables as processes M_P, L_P, F_P. Variables m, l, f denote whether or not the match is lit, the lightning strikes and the fire occurs, based on the outcome of these three processes, respectively.

Encoding structural equation framework using process calculus. We can encode a set of structural equations by considering each endogenous variable to be instantiated via a process. We consider an equation for an exogenous variable A_R to represent a process for evaluating the outcome value for the corresponding process A_P. The different variables on the right hand side for equation for A_R will indicate the interaction between the variables on the left hand side and the right hand side of the structural equation. Therefore the boolean operations over the random variables in the structural equations can be represented as boolean operations over the corresponding program variables in the process calculus framework. We denote the correspondence in Figure 2.2.

2.4 Why process calculus?

Prior work assumes a given context and the structural equations are fixed. Typically for structural equation framework, the equations encode one unique setting with input values (exogenous variables) and explicitly changing the context value involves changing the set of structural equations [14, 16]. For instance, in the structural equation encoding for Example 1, prior work assumes that the value of u_A, u_E and u_B is given to us, i.e. it assumes that whether or not E loads the gun, A shoots or B shoots is fixed. Further, the interaction and scheduling primitives also need to be defined in terms of random variables and structural equations and are fixed.

However, we are interested in identifying the choices made within the programs, the program expressions and the interactions which are relevant for the violation. Let us consider a more practical version of Example 1. Assume that B is a user trying to log into her account and A is an adversary who has stolen the password for B's account from an external repository E. In this case, whoever's request reaches the Server first, gains access to the account. If an

Structural equation framework for actual causation	**Process calculus**
Endogenous variables ($\mathcal{V}$)	Outputs for processes corresponding to the structural equation for every variable
Exogenous variables ($\mathcal{U}$)	We have exogenous variables and Σ fixes their values
Structural equations	Programs in $\mathcal{P}$. The interactions between programs are implemented according to structural equations: processes corresponding to endogenous/ exogenous variables on RHS of the structural equation, send messages to the process for the endogenous variable on the LHS of the structural equation.
$\wedge, \vee, \ldots$ (Boolean operators)	Implemented via boolean operations over variable values received by processes, or constants in Σ
No direct encoding (test for all exogenous values)	Internal choice operator
No direct encoding (record the manner in which disjunction is satisfied)	Asymmetric disjunction operator

Figure 2.2: Correspondence between SEM for actual causation and process calculus framework

adversary gains access to the account, it is a violation of the security property that only the legitimate user should get access to her account[12].

Typically in process calculus, the `send` and `recv` actions are untargeted, i.e. a general model can express the fact that either A's message or B's message could have reached first. In Figure 2.3, we denote a simplified version of this example (ideally the requests will be checked in more detail and access will only be granted to whoever sends the correct password first). Here the `Match` primitive checks the password against the previously stored password in the Server's memory for the associated account and `Allow` primitive denote allowing access to process P1 or P2. E_P denotes E's program where the password is sent to the Adversary A_P. We log the choice of the adversary in order to access an account (represented by a). Similarly, we log B's choice to access the account. Modeling choices explicitly is useful for assigning blame. We then model the send actions by both A and B. In Figure 2.3, both the logs use the same model of programs and differ in the scheduling. Further, we can also model a case where only one of the synchronizations occur– in this case if the Adversary A synchronizes with the Server first, then B's message will be ignored. The same model can also be used to express

[12]In Part 2 of the dissertation, we provide a detailed causal analysis of an authentication example.

different logs resulting from different choices for a and b.

E_P	A_P	B_P	H_P
1: $\texttt{send}(A_P, pwd)$;	1: $a = 0 \oplus 1$; 2: $pwd = \texttt{recv}(E_P)$; 3: $s = (a, pwd)$; 4: $\texttt{send}(H_P, s)$;	1: $b = 0 \oplus 1$; 2: $t = (b, pwd)$; 3: $\texttt{send}(H_P, b)$;	1: $pwd1 = \texttt{recv}(P1)$; 2: $pwd2 = \texttt{recv}(P2)$; 3: $access = Match(pwd1)? \texttt{Allow}(P1) : (Match(pwd2)? \texttt{Allow}(P2))$;

LOG l	LOG l'
$\langle A_P, 1, \mathbf{r}\rangle$	$\langle A_P, 1, \mathbf{r}\rangle$
$\langle B_P, 1, \mathbf{r}\rangle$	$\langle B_P, 1, \mathbf{r}\rangle$
$\langle\langle E_P, 1\rangle, \langle A_P, 2\rangle\rangle$	$\langle\langle E_P, 1\rangle, \langle A_P, 2\rangle\rangle$
$\langle B_P, 2\rangle$	$\langle A_P, 3\rangle$
$\langle A_P, 3\rangle$	$\langle B_P, 2\rangle$
$\langle\langle A_P, 4\rangle, \langle H_P, 1\rangle\rangle$	$\langle\langle B_P, 3\rangle, \langle H_P, 1\rangle\rangle$
$\langle\langle B_P, 3\rangle, \langle H_P, 2\rangle\rangle$	$\langle\langle A_P, 4\rangle, \langle H_P, 2\rangle\rangle$
$\langle H_P, 3\rangle$	$\langle H_P, 3\rangle$

Figure 2.3: Example 1: A modified example to explain general models of interaction.

Due to the presence of different primitives for interaction, choice and scheduling, process calculus naturally encodes interacting systems concisely, as shown above.

The structural equation encoding for this example will need several variables to describe the general interaction model, including:

- Exogenous variable (for pwd)
- Variables for choices a, b – prior frameworks primarily express only one instantiation for the exogenous variables so either the choices will need to be encoded as endogenous variables, or a different set of structural equations need to be included for all possible choices.
- Scheduling variables (order of internal actions and interactions) for programs E_P, A_P, B_P, H_P.
- Variables denoting interactions for all possible pairs. For instance, in order to denote that A and H synchronized in one step of the log, we will need to set the corresponding variable as 1 and all other variables denoting possible synronizations as 0.
- Variables denoting execution of expressions: Depending on the complexity of the program being modeled, whether or not a certain expression is executed would need to be

indicated via a variable. in case of our log, omitting the corresponding label indicates that the expression did not execute.

Further, for constructing counterfactual scenarios in our causal analysis, we treat interaction and scheduling primitives differently. Structural equations, as used in prior actual causation literature, model all primitives as random variables of the same type[13].

These observations demonstrate why process calculus is a more natural choice for encoding interacting systems. In real life protocols such as authentication protocols, the number of interacting programs and the possible synchronizations and scheduling possibilities, are significantly larger than the example described above. Using the structural encoding as described would be significantly less concise for expressing a general model of interaction.

2.5 Examples

We consider two examples. We indicate the structural equations and a corresponding model using process calculus framework.

Consider an instance of disjunctive causes [14, 60] where a forest fire can occur due to either a match being lit by an arsonist or a lightning strike. We describe Example 3 in Figure 2.4 and model it in two different ways: one where both the lightning and the arsonist can strike at the same time (Figure 2.4a) and a model where only one of these could occur first (Figure 2.4b). In this case, we consider factors like oxygen in the air, dryness of wood as constants given to the model [14].

Similarly, a case of conjunctive cases can be modeled by changing the boolean operator from $\vee$ to $\wedge$ in Figure 2.4a.

Let us consider another example [14, 48]. In this example, a vote passes (X) if either of the two voters, $V1$ or $V2$ vote in favor. A machine M tabulates the votes and x is 1 if $M \geq 1$. The structural equations and the corresponding encoding in process calculus is given in Figure 2.5. The program for $V1$ indicates that first a value of $v1$ is chosen internally. Then this value is sent to M. Similarly, $V2$ chooses a value for $v2$ and sends to M. M receives both values and sends the sum to X, which decides whether or not the vote passes. In this case, we can model all the contexts where $V1$ votes or not and when $V2$ votes or not. The dependence of x on m can be captured via the conditional operator. Column (b) denotes the projections of the log for individual processes for V_1, V_2, M, X. On the given log, M receives $V1$'s choice first and then $V2$'s choice[14].

[13] The claims in this dissertation focus on structural equations as used in prior work in actual causation.
[14] As discussed in the previous section, we can denote a more general model for scheduling.

Modeling-based features of the formalism. We would like to highlight three points in the formalism when compared with the modeling in the structural equations.

1. *Expressing general models of interaction more concisely:* By modeling internal choice, we allow for the same model to be used for several different 'contexts.' That is, if $V1$ decided to not vote or $V2$ decided to not vote or if both of them decided to not vote- all of these scenarios can be depicted with the same model as we have given above- the log would specify a different internal choice resolution.
2. *Capturing sequential and interaction-based dependencies:* Using our sequencing operator as well as the interaction primitives, we can capture dependencies within a program as well as across programs. The log, additionally, records the temporal ordering of evaluation of programs.
3. *Modeling non-determinism in the execution semantics:* Process calculi frameworks can model non-determinism in how the programs evaluate to different traces. We can represent all traces which arise due to the non-deterministic execution of programs. This feature of our formalism is crucial for the sufficiency clause in our definition, as described in the next section.

Remarks.

- The syntax in our framework only allows choices over data variables and does not specify choices over synchronizations. For instance, if B decides to not shoot, this choice can be modeled in two ways: first, B_P chooses a value 0 and sends a message to H_P, which is interpreted as not shooting. Alternatively, B_P could not send any message at all. In the latter case, we need to define an explicit timeout action which bounds the time that a process will wait for a message. Instead in our formalization, we allow choices over data variables and keep the synchronization structure fixed as on the log. This is an important consideration when the examples are being modeled.
- The log records the sequence of events which occurred in the actual context. The distinction between using asymmetric and symmetric disjunction stems from the amount of evidence available (as part of the log) in order to make a causal inference.

In the next section, we describe our definition and demonstrate how we can apply our definition to examples.

Example 3 (Disjunctive causes). *[14] A forest fire could be caused by either lightning or a match stuck by an arsonist. Structural equations:*

$$\begin{aligned} ML_R &\leftarrow u_{ML} \\ L_R &\leftarrow u_L \\ F_R &\leftarrow L_R \vee ML_R \end{aligned}$$

(a)

Program for L
$1 : l = 0 \oplus 1;$
$2 : \texttt{send}(F_P, l);$

Program for ML
$1 : ml = 0 \oplus 1;$
$2 : \texttt{send}(F_P, ml);$

Program for F
$1 : l = \texttt{recv}(L_P);$ //input from L
$2 : ml = \texttt{recv}(ML_P);$ //input from ML
$3 : f = l \vee ml;$

(b)

$\log(t)|_L$
$\langle L_P, 1, \mathtt{r}\rangle$
$\langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle$

$\log(t)|_{ML}$
$\langle ML_P, 1, \mathtt{r}\rangle$
$\langle\langle ML_P, 2\rangle, \langle F_P, 2\rangle\rangle$

$\log(t)|_F$
$\langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle$
$\langle\langle ML_P, 2\rangle, \langle F_P, 2\rangle\rangle$
$\langle F_P, 3\rangle$

Violation: All traces where $f = 1$.
Observation: This model allows independent causes of the violation.

(a) Example: Forest fire – disjunctive scenario. Here a symmetric operator is used to evaluate f.

(a)

Program for L
$1 : l = 0 \oplus 1;$
$2 : \texttt{send}(F_P, l);$

Program for ML
$1 : ml = 0 \oplus 1;$
$2 : \texttt{send}(F_P, ml);$

Program for F
$1 : l = \texttt{recv}(L_P);$ //input from L
$2 : ml = \texttt{recv}(ML_P);$ //input from ML
$3 : f = l \,[]\, ml$

(b)

$\log(t)|_L$
$\langle L_P, 1, \mathtt{r}\rangle$
$\langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle$

$\log(t)|_{ML}$
$\langle ML_P, 1, \mathtt{r}\rangle$
$\langle\langle ML_P, 2\rangle, \langle F_P, 1\rangle\rangle$

$\log(t)|_F$
$\langle\langle L_P, 1\rangle, \langle F_P, 1\rangle\rangle$
$\langle\langle ML_P, 2\rangle, \langle F_P, 2\rangle\rangle$
$\langle F_P, 3, \mathtt{r}\rangle$

Violation: All traces where $f = 1$.
Observation: This model does not allow independent causes of the violation. In this case, the label for $\langle F_P, 3, \mathtt{r}\rangle$ clearly indicates which disjunct was used in order to evaluate the value of h.

(b) Example: Forest fire – disjunctive scenario. Here an asymmetric operator is used to evaluate f. In this case the matchstick being lit has more influence in the actual log.

Figure 2.4: Example 3: Forest fire – disjunctive scenario

Example 4 (Voting Machine). *Consider a case where two voters $V1$ and $V2$ cast votes. A measure is passed if either of the votes is 1. Let X denote whether or not the vote passes. Additionally there is a voting machine which tabulates the votes, let M represent the total number of votes. Now $M = V1 + V2$ and $X = 1$ iff $M \geq 1$.*

Structural equations:

$$\begin{aligned} X_R &= 1 \text{ iff } M_R = 1 \\ M_R &\leftarrow V1_R + V2_R \end{aligned}$$

(a)

Program for $V1$
$1 : v1 = 0 \oplus 1;$
$2 : \texttt{send}(M, v1);$

Program for $V2$
$1 : v2 = 0 \oplus 1;$
$2 : \texttt{send}(M, v2);$

Program for M
$1 : v1 = \texttt{recv}(V1);$
$2 : v2 = \texttt{recv}(V2);$
$3 : m = v1 + v2;$
$4 : \texttt{send}(X, m);$

Program for X
$1 : m = \texttt{recv}(M);$
$2 : x = (m \geq 1)?\ 1 :\ 0;$

(b)

$\log(t)|_{V1}$
$\langle V1, 1, \mathbf{r}\rangle$
$\langle\langle V1, 2\rangle, \langle M, 1\rangle\rangle$

$\log(t)|_{V2}$
$\langle V2, 1, \mathbf{r}\rangle$
$\langle\langle V2, 2\rangle, \langle M, 2\rangle\rangle$

$\log(t)|_{M}$
$\langle\langle V1, 2\rangle, \langle M, 1\rangle\rangle$
$\langle\langle V2, 2\rangle, \langle M, 2\rangle\rangle$
$\langle\langle M, 3\rangle$
$\langle\langle M, 4\rangle, \langle X, 1\rangle\rangle$

$\log(t)|_{X}$
$\langle\langle M, 4\rangle, \langle X, 1\rangle\rangle$
$\langle X, 2\rangle$

Violation: All traces where $x = 1$.
Observation: We can model both choices for $V1$ and $V2$.

Figure 2.5: Example 4: Voting machine

"Karma refers to the spiritual principle of cause and effect where intent and actions of an individual (cause) influence the future of that individual (effect)."

— Gita, Encyclopedia Britannica, Wikipedia

Chapter 3

Defining Choices and Actions as Actual Causes

Chapter goal. In this section, we provide our definition of actual cause. We first define the terms *property* and *causal sequence* in more detail. Next, we use Example 1 to motivate our clauses.

Defining property and causal sequence. We formulate a property as a set of traces. In the sequel, let V denote the property of interest[1]. Consider a trace t starting from the initial configuration $\mathcal{C}_0 = \langle I, \mathcal{P}, \Sigma \rangle$. If $t \in V$, then t violates the property $\neg V$[2].

The programs for our running example (Example 1) are given in Figure 3.1 (part a) and the log is given in part (b). From the log, we can infer that the choices for the variables are: $e = 1$, $a = 1$ and $b = 1$. For our running example, we want to identify which of the internal choice values for e, a or b and the actions shown in part (b) of the figure, are a cause of $h = 1$[3] Therefore, we are interested in finding the program expressions (modeling the choices and interactions) on the log which are a cause of $h = 1$. We denote this sequence of program expressions by $\vec{\alpha}$ and we call it a *causal sequence*. The sequence $\vec{\alpha}$ is a subsequence of the log and includes expressions where internal choices are made, other actions within a program and interactions

[1] V stands for 'violations', a convention from the security domain.

[2] For readers more familiar with counterfactual tradition in causation in philosophy, the occurrence condition will be of the form trace $t \in V$, i.e. trace t contains a violation.

[3] Note that our current model does not include a global state, therefore when we define a violation, we will need to specify the interaction or predicate which is affected by the local variable assuming a certain value. This can easily be captured in the above model by introducing a program X_P, and adding an interaction between H_P and X_P such that, H_P sends the value h to X_P. Now the violation will specify the value sent by H. For the examples in this part, we omit the addition of an extra program, for ease of description and focus on referring to a violation as 'all traces where $h = 1$.'

across programs.

Example 1, Loader

Two shooters A and B both shoot at a target H simultaneously. Another agent E loads the gun for A before A shoots. We are interested in finding the cause of the target being hit.

Structural equations:

$$\begin{aligned} A_R &= u_A \\ B_R &= u_B \\ E_R &= u_E \\ H_R &= (A_R \wedge E_R) \vee B_R \end{aligned}$$

(a)	**(b)**	**(c)**	**(d)**
Program for E	$\log(t)\|_E$	$\vec{\alpha}\|_E$	$\vec{\alpha}'\|_E$
$1 : e = 0 \oplus 1$	$\langle E_P, 1, \mathbf{r}\rangle$		$\langle E_P, 1, \mathbf{r}\rangle$
$2 : \mathtt{send}(A_P, e);$	$\langle\langle E_P, 2\rangle, \langle A_P, 2\rangle\rangle$		$\langle\langle E_P, 2\rangle, \langle A_P, 2\rangle\rangle$
Program for A	$\log(t)\|_A$	$\vec{\alpha}\|_A$	$\vec{\alpha}'\|_A$
$1 : a = 0 \oplus 1$	$\langle A_P, 1, \mathbf{r}\rangle$		$\langle A_P, 1, \mathbf{r}\rangle$
$2 : e = \mathtt{recv}(E_P);$	$\langle\langle E_P, 2\rangle, \langle A_P, 2\rangle\rangle$		$\langle\langle E_P, 2\rangle, \langle A_P, 2\rangle\rangle$
$3 : s = e \wedge a;$	$\langle A_P, 3\rangle$		$\langle A_P, 3\rangle$
$4 : \mathtt{send}(H_P, s);$	$\langle\langle A_P, 4\rangle, \langle H_P, 1\rangle\rangle$		$\langle\langle A_P, 4\rangle, \langle H_P, 1\rangle\rangle$
Program for B	$\log(t)\|_B$	$\vec{\alpha}\|_B$	$\vec{\alpha}'\|_B$
$1 : b = 0 \oplus 1$	$\langle B_P, 1, \mathbf{r}\rangle$	$\langle B_P, 1, \mathbf{r}\rangle$	
$2 : \mathtt{send}(H_P, b);$	$\langle\langle B_P, 2\rangle, \langle H_P, 2\rangle\rangle$	$\langle\langle B_P, 2\rangle, \langle H_P, 2\rangle\rangle$	
Program for H	$\log(t)\|_H$	$\vec{\alpha}\|_H$	$\vec{\alpha}'\|_H$
$1 : s = \mathtt{recv}(A_P);$	$\langle\langle A_P, 4\rangle, \langle H_P, 1\rangle\rangle$		$\langle\langle A_P, 4\rangle, \langle H_P, 1\rangle\rangle$
$2 : b = \mathtt{recv}(B_P);$	$\langle\langle B_P, 2\rangle, \langle H_P, 2\rangle\rangle$	$\langle\langle B_P, 2\rangle, \langle H_P, 2\rangle\rangle$	
$3 : h = b \vee s;$	$\langle H_P, 3\rangle$	$\langle H_P, 3\rangle$	$\langle H_P, 3\rangle$

Violation: All traces where $h = 1$.

Causal analysis: Parts **(c)** and **(d)** show two causal sequences. B's shooting forms one causal sequence. Independently, E loading the gun and A shooting forms the second causal sequence.

Figure 3.1: Example 1, Loader: causal analysis (two causal sequences)

Causal sequence as evidence from the log. We want our causal determination to be based on evidence. Therefore, to establish the choices and interactions in $\vec{\alpha}$ to be a cause of violation V on l, we must first provide evidence of the violation on the log l. For example, in Figure 3.1, let $\vec{\alpha}$ contain the choices made by E, A and B whether or not to shoot and the interactions amongst these. Since $e = a = b = 1$ on the log, we can infer that $h = 1$.

Constructing counterfactual scenarios by modifying program expressions and choices. In this part, we explain how to modify the programs, so that we can test whether the execution of program expressions corresponding to $\vec{\alpha}$ is sufficient for the violation.

A key component of constructing counterfactual scenarios in prior work, has been identifying components of the model to hold fixed. Over-permissiveness in generating counterfactual scenarios by allowing unrestricted modification to the model can lead to counter-intuitive causal determinations in several cases [29, 61] (also discussed in Section 4.3).

Our formalism provides natural constructs to express *sequencing, interaction, choice and time*. As a consequence, we can construct counterfactual worlds in an interaction-aware manner such that we preserve dependencies across related variables in the same program and preserve the temporal ordering as on the log. In our process calculus framework, removing actions arbitrarily can lead to a halting of a program execution. For instance in Figure 3.1, removing the expression on line number 1 for program E_P would cause the program execution for E to halt, since the next expression requires the value of e before proceeding. Similarly, only removing the `send` action in line 2 would cause the concurrent execution of the programs to halt, since A_P would need to receive a value before proceeding. Therefore, our interventions need to be aware of *interaction* and *dependency structure* of the program.

If an interaction is omitted from $\vec{\alpha}$, then we intervene such that the progress of the program sending a value is not hindered. For the variable in the receiving program, we test all cases where the variable could have taken any value. For instance in case of our running example as shown in Figure 3.2, if we consider a case where there is no interaction between E and A, then we will construct contingencies, *for all possible values* of e which was getting instantiated due to the interaction[4].

We call this process of replacing actions with 'dummy alternatives' as a *dummifying transformation*. The dummifying transform is a function that takes as input the programs which executed on the log and a sublog $\vec{\alpha}$, and modifies the programs in the initial configuration in the following manner:

- Identify a subsequence $\vec{\alpha}$ of the log l.

[4]Traces can filter out unreasonable contingencies as well.

- Modify the programs in the initial configuration such that it removes all expressions not included in $\vec{\alpha}$.
- Generate all traces that could result from this modified set of programs and pick those which contain the same choices and interaction structure as $\vec{\alpha}$.

Note in both $\vec{\alpha}$ and $\vec{\alpha}'$ in Figure 3.1, we fix certain internal choices and interactions. On $\vec{\alpha}$, had B chosen to not shoot (i.e. $b = 0$), then $h = 0$ and the target would not have been hit. Similarly, had B not sent its choice to H, $h = 0$. The basic idea is to test all internal choices and interactions to ensure that they are not merely progress enablers and their execution as on the log is critical to reproduce the same outcome as on the log. In order to do so, we need to test all executions that emerge from an initial configuration where B made a different choice (in the given allowed range of values) and where B did not send the message to H.

Intervening as we described above, can lead to several traces, some of which may not share any resemblance with the log. Next, we discuss how we restrict the counterfactual scenarios we consider for testing sufficiency of a causal sequence.

Constraining choices and interaction structure as on the log. In non-deterministic models such as ours, if there is no specific assumption about scheduling, it is typical to consider the trace set arising from concurrent executions of all programs, this includes all possible interleaving of actions. In the *Sufficiency* condition of our actual cause definition, we would like to focus on the hypothetical scenarios which resemble the conditions under which the log was generated and a property was violated.

It is important to note that there is a spectrum of consistency conditions that need to be investigated depending on how strong a coupling we find appropriate between the log and the traces to consider in *Sufficiency*. We describe two of the alternatives before providing our design choice.

1. **Execute same programs as the log:** One way to connect the log and and trace set in *Sufficiency* condition could be to constrain the programs in the initial configuration for the *Sufficiency* condition to be the same as that as on the log, i.e., only consider traces in which the same program prefixes execute as on the log. Considering our example (Figure 3.1), this approach would constrain all traces in *Sufficiency* condition to consider the execution of programs for A, B, E, H until the same label as on the log. This approach is problematic because traces are prefix closed, i.e. prefix of a trace is also a valid trace. We will end up considering a trace in *Sufficiency* where neither A nor B sends a message to H, or E has not yet sent a message to A. This will count as a valid counterfactual trace. Consequently our cause definition would not find either B's action or A and E's actions

as causes of the target being hit, which is counter-intuitive.

2. **Execute same choices as on the log:** This option would help to fix the choices, however it does not constrain the interaction structure, for instance in case of asymmetric disjunction. Similarly, only preserving the interaction structure as on the log would not suffice since the internal choices could vary the data transmitted along the interaction structure.

These alternatives reveal that defining consistency is a challenge as maintaining too strong a coupling with the log could lead to ignoring relevant traces. On the other hand, if we do not constrain the traces in the *Sufficiency* condition to be similar to the log, we could end up considering traces on which no violation occurs, due to a spurious action. Therefore, in our definition, when considering hypothetical scenarios, we choose to constrain both the relevant choices and interactions on the log. For those choices and other program expressions, which are not a part of the sequence $\vec{\alpha}$, we remove the expression by carefully considering how it interacts with the actions *in the causal sequence*, and test for all possible alternatives.

Minimality. Since we consider a sequence of actions on the log, therefore our definition also contains a minimality clause in order to prevent redundant actions and choices from being a part of the cause.

3.1 Actual cause definition

Our definition of actual cause identifies a subsequence of program expressions on the log as the cause of a violation $t \in V$. For our running example, if the violation is formulated as all traces where the value of $h = 1$ (i.e. the target is hit), then our goal is to find the choices and actions in Figure 3.1 which were a cause of $h = 1$. In this part, we formally discuss $\vec{\alpha}$ and how to construct counterfactual scenarios.

More formally, in the definition below, we identify a sublog $\vec{\alpha}$ of $\log(t)$, such that the program choices and actions in $\vec{\alpha}$ are actual causes and the actions in $log(t)\backslash\vec{\alpha}$ are progress enabling actions which only contribute towards the *progress* of actions in $\vec{\alpha}$, that cause the violation. In other words, the actions not considered in $\vec{\alpha}$ contain all labels whose actual returned values are irrelevant.

Briefly, here's how the definition works. We first pick a candidate projected sublog $\vec{\alpha}$ of $\log(t)$. This includes expressions where internal choices are made as well as other actions within a program. For our example, let us first consider the interactions between B_P and H_P on $log(t)$ as $\vec{\alpha}$ (shown in figure 3.1). We consider counterfactual traces obtained from initial

configurations in which program actions omitted from $\vec{\alpha}$ (i.e. actions for A_P and E_P as well their interactions with H_P) are replaced by actions that do not have any effect other than enabling the program to progress (referred to as no-op or non-operational expressions). If a violation appears in all such counterfactual traces, then this sublog $\vec{\alpha}$ is a good candidate. Of all such good candidates, we choose those that are minimal. Let us consider the log $\vec{\alpha}$ as shown in part c in Figure 3.1. In this case, we can evaluate that $h = 1$ since $b = 1$. Therefore, had B made the same choice as on the log *and* H_P only interacted with B_P, a violation would have resulted.

Similarly, consider another subsequence of choices and actions on the log: $\vec{\alpha}'$ as shown in part (d) in Figure 3.1. In this case, we can evaluate that $h = 1$ since $a = 1$ and $e = 1$. Therefore, had A and E made the same choices as on the log *and* H_P only interacted with A_P, a violation would have resulted.

Dummification. The key technical difficulty in writing this definition is removing program expressions omitted from $\vec{\alpha}$. We cannot simply erase any such action because the action is expected to return a term which is bound to a variable used in the action's continuation. Furthermore, the potential values which could be returned for an action representing an internal choice are restricted. Hence, our approach is to substitute the variables binding the returns of actions which have been removed with arbitrary terms t. In case of internal choices, we substitute the variables binding the returns of 'removed' choice-expressions with all possible allowed choices (we denote it by function f'). In the figures, we denote this by a 'no-op' since the process essentially removes an action and makes it *non-operational*.

Formally, we assume a function $f : I \times \text{LineNumbers} \to \text{Terms}$ that in program i, for line number b, suggests a suitable term $f(i, b)$ that must be returned if the action from line b in program i is replaced with a no-op.

For choice expressions, our interventions replace the expression excluded from $\vec{\alpha}$ with a restricted set of values. We define a subset of values $\text{Choices} \subseteq \text{Terms}$, which includes the allowed choices specified in a choice expression. We assume a function $f' : I \times \text{LineNumbers} \to \text{Choices}$ that for line number b in program i picks one of the allowed choices. For instance, if a can be instantiated to any value between $\{1, 9\}$, then replacing a with the output of f corresponds to replacing a with any value between $\{1, 9\}$. However, if $a = 0 \oplus 1$ and this expression is not included in $\vec{\alpha}$, then replacing a with the output of f' corresponds to replacing a with only 0 or 1.

In our cause definition we universally quantify over f and f', thus obtaining the effect of a no-op. For technical convenience, we define a syntactic transform called $\mathsf{dummify}()$ that takes an initial configuration, the chosen sublog $\vec{\alpha}$ and the functions f, f', and produces a new initial

configuration obtained by erasing actions not in $\vec{\alpha}$ by terms obtained through f and replacing the internal choices with f'.

Definition 2 (Dummifying transformation)**.** *Let $\langle I, \mathcal{P}, \Sigma\rangle$ be a configuration and let $\vec{\alpha}$ be a log. Let $f : I \times \mathit{LineNumbers} \to \mathit{Terms}$ and $f' : I \times \mathit{LineNumbers} \to \mathit{Choices}$ where $\mathit{Choices} \subseteq \mathit{Terms}$.*

The dummifying transform $\mathsf{dummify}(I, \mathcal{P}, \Sigma, \vec{\alpha}, f, f')$ *is the initial configuration $\langle I, \mathcal{D}, \Sigma\rangle$, where for all $i \in I$, $\mathcal{D}(i)$ is $\mathcal{P}(i)$ modified as follows:*

- *If a choice expression does not appear in $\vec{\alpha}$, then replace the output of the choice with function f' where f' picks one of the allowed values.*
- *If a choice expression appears in $\vec{\alpha}$, then replace the output of the choice with the value chosen on the log.*
- *If a* `send` *action does not appear in $\vec{\alpha}$, then replace the action in the corresponding program with 0.*
- *If an action contains an asymmetric disjunction operator, then replace the output with the disjunct chosen on the log.*
- *If an action is not a* `send` *or a choice expression and does not appear in $\vec{\alpha}$, then replace the action $(b : x = \alpha); e$ with the value $e[f(i, b)/x]$ in the corresponding program.*

The dummifying function above explains how each of the constructs in our syntax are intervened on. If an expression is in $\vec{\alpha}$, then the corresponding program expression is executed as on the log. If an expression involving internal choice is dummified, we replace it by possible choices and test the obtained traces. If a disjunct with asymmetric disjunction is dummified then we choose the same disjunct as chosen on the log. If an interaction is dummified, then the `send` action returns 0 to the continuation and the variable instantiated by corresponding `recv` action is treated as a free variable. For every other construct, if the corresponding expression is not included in $\vec{\alpha}$, then we treat the corresponding variable in the problem as a free variable.

Note that we also require $\vec{\alpha}$ to be a sublog of $\log(t)$, hence we can't arbitrarily change the dependency graph.

We demonstrate how the above defined dummifying function works on our running example. In Figure 3.2, we consider $\vec{\alpha}$ from Figure 3.1 and describe how the dummifying transform works. Note that in our cause definition, we *universally quantify over dummified values* which allows us to test the dependence of the violation on the removed expression.

Example 1, Loader

(a) Original programs:	**(b)** Log:	**(c)** Causal sequence:	**(d)** Dummified programs:
Program for E	$\log(t)\|_E$	$\vec{\alpha}\|_E$	**Program for E**
$1 : e = 0 \oplus 1$	$\langle E_P, 1, \mathtt{r}\rangle$		$1 : e =$ 0 OR 1
$2 : \mathtt{send}(A_P, e);$	$\langle\langle E_P, 2\rangle, \langle A_P, 2\rangle\rangle$		$2 : no - op$
Program for A	$\log(t)\|_A$	$\vec{\alpha}\|_A$	**Program for A**
$1 : a = 0 \oplus 1$	$\langle A_P, 1, \mathtt{r}\rangle$		$1 : a =$ 0 OR 1 ;
$2 : e = \mathtt{recv}(E_P);$	$\langle\langle E_P, 2\rangle, \langle A_P, 2\rangle\rangle$		$2 : e =$ 0 OR 1 ;
$3 : s = e \wedge a;$	$\langle A_P, 3\rangle$		$3 : s =$ 0 OR 1 ;
$4 : \mathtt{send}(H_P, s);$	$\langle\langle A_P, 4\rangle, \langle H_P, 1\rangle\rangle$		$4 : no - op$
Program for B	$\log(t)\|_B$	$\vec{\alpha}\|_B$	**Program for B**
$1 : b = 0 \oplus 1$	$\langle B_P, 1, \mathtt{r}\rangle$	$\langle B_P, 1, \mathtt{r}\rangle$	$1 : b = 1$
$2 : \mathtt{send}(H_P, b);$	$\langle\langle B_P, 2\rangle, \langle H_P, 2\rangle\rangle$	$\langle\langle B_P, 2\rangle, \langle H_P, 2\rangle\rangle$	$2 : \mathtt{send}(H_P, b);$
Program for H	$\log(t)\|_H$	$\vec{\alpha}\|_H$	**Program for H**
$1 : s = \mathtt{recv}(A_P);$	$\langle\langle A_P, 4\rangle, \langle H_P, 1\rangle\rangle$		$1 : s =$ 0 OR 1 ;
$2 : b = \mathtt{recv}(B_P);$	$\langle\langle B_P, 2\rangle, \langle H_P, 2\rangle\rangle$	$\langle\langle B_P, 2\rangle, \langle H_P, 2\rangle\rangle$	$2 : b = \mathtt{recv}(B_P);$
$3 : h = b \vee s;$	$\langle H_P, 3\rangle$	$\langle H_P, 3\rangle$	$3 : h = b \vee s;$

Violation: All traces where $h = 1$.

Causal analysis: The causal sequence $\vec{\alpha}$:
$\langle B_P, 1, \mathtt{r}\rangle, \langle B_P, 3\rangle, \langle\langle B_P, 3\rangle, \langle H_P, 2\rangle\rangle, \langle H_P, 3\rangle$, i.e. our definition will find B's choices and actions as a cause. Independently, our definition will also find E and A's choices and actions as a cause.

Figure 3.2: Example 1, Loader: Causal analysis

As seen in Figure 3.2, the send actions not in $\vec{\alpha}$ are replaced by (effective) no-ops, the choice expressions not in $\vec{\alpha}$ are replaced by 0 or 1 (output of f') whereas the rest of the actions not in $\vec{\alpha}$ are instantiated by function f. Note that in this case, since the variables are boolean, we see the same options for instantiations for recv and choice expressions, i.e. for f and f'. The resulting traces from these programs will consider all the values given in part d. Notice that irrespective of value of e or a, the value of h will always be 1 given that $b = 1$ and the interaction structure in $\vec{\alpha}$ is followed. Hence, these choices and interactions are *sufficient* to cause the violation. We test for such contingencies via the *Sufficiency* clause. We also note that we could dummify fewer actions and also include the send action from A_P and include it in the

cause set. However, our goal is to find a minimal cause set, hence we add a *Minimality* clause.

We now present our main definition of actual cause.

Definition 3 (Actual Cause of Violation)**.** *Let t be a trace from the initial configuration $\langle I, \mathcal{P}, \Sigma \rangle$, and t contains a violation V. Let $\vec{\alpha}$ be a projected sublog of $\log(t)$. We say that $\vec{\alpha}$ is the actual cause of violation V on t if the following hold:*

1. *(**Sufficiency'**) Let $\mathcal{C}_0' = \mathsf{dummify}(I, \mathcal{P}, \Sigma, \vec{\alpha}, f, f')$ and let T be the set of traces starting from $\mathcal{C}_0'$ whose logs contain $\vec{\alpha}$ as a projected sublog. Then, for all values of f and f', T is non-empty and every trace in T has the violation V, i.e, $T \subseteq V$.*
2. *(**Minimality'**) No proper sublog of $\vec{\alpha}$ satisfies condition 1.*

By ensuring that $\vec{\alpha}$ is a sublog of $\log(t)$, we ensure that synchronizations different from the log are not considered in our analysis.

Remarks. At the end of the actual cause definition, we obtain one or more sequences of actions $\vec{\alpha}$. These sequences are deemed the independent actual causes of the violation on t. In our running example, both $\vec{\alpha}$ and $\vec{\alpha}'$ are independent causes of the violation.

Our definition identifies a sequence of choices and program actions as causes of a violation. However, in some applications it may be necessary to ascribe programs as causes. This can be straightforwardly handled by lifting the above definition: A program i (or $\mathcal{P}(i)$) is a cause if one of its expressions appears in $\vec{\alpha}$.

In our running example, the choices and interactions for $\{\vec{\alpha}|_B, \vec{\alpha}|_H\}$ and for $\{\vec{\alpha}|_E, \vec{\alpha}|_A, \vec{\alpha}|_H\}$ are causes of the violation. In other words, B's shooting action is a cause of the target being hit. E's loading the gun and A's shooting action are another independent cause of the violation.

3.2 Examples

Format for encoding of examples. For every example in the rest of the dissertation, we first provide the description in text. Next, we provide the structural equations where the variables are marked with subscript R. The process calculus encoding of the example contains four columns.

- Part **(a)** provides the model of programs corresponding to the example description.
- Part **(b)** specifies the log.
- Part **(c)** provides the causal sequence. The causal sequence is a subsequence of the log.
- Part **(d)** provides the dummified programs. These programs are obtained from the original programs given in part **(a)**, by dummifying the expressions omitted from the causal

sequence.

In Chapter 2, we omitted the last two columns as the focus was on describing the formalism.

Disjunctive causes (Example 3). In Figure 3.3, we show how our definition applies to the example based on disjunctive causes from the previous chapter. In this case, a forest can be burned due to either a matchstick being lit (ML_P), or a lightning strike (L_P). $\vec{\alpha}$ consists of the choice made by L_P and the interaction between L_P and F_P. While observing the dummified programs, we can see that the value of f will be 1, irrespective of the value of ml. Similarly, if we had chosen another subsequence of the log, $\vec{\alpha}'$ which contained the choice made by ML_P and its interaction with F_P, then we would get another independent cause of the violation, $f = 1$.

Example 3, Disjunctive causes

(a)	**(b)**	**(c)**	**(d)**
Original programs:	Log:	Causal sequence:	Dummified programs:
Program for L	$\log(t)\|_L$	$\vec{\alpha}\|_L$	**Program for L**
$1 : l = 0 \oplus 1;$	$\langle L_P, 1, \mathbf{r}\rangle$	$\langle L_P, 1, \mathbf{r}\rangle$	$1 : l = 1;$
$2 : \mathtt{send}(F_P, l);$	$\langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle$	$\langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle$	$2 : \mathtt{send}(F_P, l);$
Program for ML	$\log(t)\|_{ML}$	$\vec{\alpha}\|_{ML}$	**Program for ML**
$1 : ml = 0 \oplus 1;$	$\langle ML_P, 1, \mathbf{r}\rangle$		$1 : ml =$ 0 OR 1 ;
$2 : \mathtt{send}(F_P, ml);$	$\langle\langle ML_P, 2\rangle, \langle F_P, 2\rangle\rangle$		$2 : no - op$
Program for F	$\log(t)\|_F$	$\vec{\alpha}\|_F$	**Program for F**
$1 : l = \mathtt{recv}(L_P);$	$\langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle$	$\langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle$	$1 : l = \mathtt{recv}(L_P);$
$2 : ml = \mathtt{recv}(ML_P);$	$\langle\langle ML_P, 2\rangle, \langle F_P, 2\rangle\rangle$		$2 : ml =$ 0 OR 1 ;
$3 : f = l \vee ml;$	$\langle F_P, 3\rangle$	$\langle F_P, 3\rangle$	$3 : f = l \vee ml;$

Violation: All traces where $f = 1$.

Causal analysis: The causal sequence $\vec{\alpha}$: $\langle L_P, 1, \mathbf{r}\rangle, \langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle, \langle F_P, 3\rangle$, i.e. our definition finds both L's choices and actions as well as ML's choices and actions as independent causes.

Figure 3.3: Example 3, Disjunctive scenario: causal analysis

However, note that if we had conjunction instead of disjunction, i.e. both l and ml need to be 1 for f to take a value of 1, then $\vec{\alpha}$ will contain both the choices and interactions for L_P and ML_P as shown in Figure 3.4. Note that unlike the disjunctive case, both L_P and ML_P's choices and interactions are crucial in order to set f to 1. If either of l or ml is 0, then $f = 0$.

We can show that both the choices and interactions as shown in $\vec{\alpha}$ are needed.

Example 2, Forest fire – conjunctive scenario

(a)

Original programs:

Program for L

$1 : l = 0 \oplus 1;$

$2 : \texttt{send}(F_P, l);$

Program for ML

$1 : ml = 0 \oplus 1;$

$2 : \texttt{send}(F_P, ml);$

Program for F

$1 : l = \texttt{recv}(L_P);$

$2 : ml = \texttt{recv}(ML_P);$

$3 : f = l \wedge ml;$

(b)

Log:

$\log(t)|_L$

$\langle L_P, 1, \mathtt{r}\rangle$

$\langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle$

$\log(t)|_{ML}$

$\langle ML_P, 1, \mathtt{r}\rangle$

$\langle\langle ML_P, 2\rangle, \langle F_P, 2\rangle\rangle$

$\log(t)|_F$

$\langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle$

$\langle\langle ML_P, 2\rangle, \langle F_P, 2\rangle\rangle$

$\langle F_P, 3\rangle$

(c)

Causal sequence:

$\vec{\alpha}|_L$

$\langle L_P, 1, \mathtt{r}\rangle$

$\langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle$

$\vec{\alpha}|_{ML}$

$\langle ML_P, 1, \mathtt{r}\rangle$

$\langle\langle ML_P, 2\rangle, \langle F_P, 2\rangle\rangle$

$\vec{\alpha}|_F$

$\langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle$

$\langle\langle ML_P, 2\rangle, \langle F_P, 2\rangle\rangle$

$\langle F_P, 3\rangle$

(d)

Dummified programs:

Program for L

$1 : l = 1;$

$2 : \texttt{send}(F_P, l);$

Program for ML

$1 : ml = 1;$

$2 : \texttt{send}(F_P, ml);$

Program for F

$1 : l = \texttt{recv}(L_P);$

$2 : ml = \texttt{recv}(ML_P);$

$3 : f = l \wedge ml;$

Violation: All traces where $f = 1$.

Causal analysis: The causal sequence $\vec{\alpha}$:
$\langle L_P, 1, \mathtt{r}\rangle, \langle ML_P, 1, \mathtt{r}\rangle, \langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle, \langle\langle ML_P, 2\rangle, \langle F_P, 2\rangle\rangle, \langle F_P, 3\rangle$, i.e. our definition finds both L's choices and actions as well as ML's choices and actions as a joint cause.

Figure 3.4: Example 2, Conjunctive scenario: causal analysis

Voting machine example (Example 4). In Figure 3.5, we analyze the voting machine example which was proposed as a counterexample to HP2001 definition and Hall's definition. Our definition finds $\vec{\alpha}$ as shown in the figure as a cause. The causal sequences are:

- $\langle V1_P, 1, \mathtt{r}\rangle, \langle\langle V1_P, 2\rangle, \langle M_P, 1\rangle\rangle, \langle\langle M_P, 3\rangle, \langle X_P, 1\rangle\rangle, \langle X_P, 1\rangle$.
- $\langle V2_P, 1, \mathtt{r}\rangle, \langle\langle V2_P, 2\rangle, \langle M_P, 2\rangle\rangle, \langle\langle M_P, 3\rangle, \langle X_P, 1\rangle\rangle, \langle X_P, 1\rangle$.

That is, our definition finds both $V1, M$'s choices and actions as well as $V2, M$'s choices and actions as independent causes. Our definition find the sequence of expressions involving V_1's choice and its interaction with M and X as a cause. Similarly, our definition also finds the subsequence of actions involving V_2's choice and its interaction with M and X as a cause.

Example 4, Voting Machine

(a)

Original Programs:

Program for $V1$

$1 : v1 = 0 \oplus 1;$

$2 : \texttt{send}(M, v1);$

Program for $V2$

$1 : v2 = 0 \oplus 1;$

$2 : \texttt{send}(M, v2);$

Program for M

$1 : v1 = \texttt{recv}(V1);$

$2 : v2 = \texttt{recv}(V2);$

$3 : \texttt{send}(X, v1 + v2));$

Program for X

$1 : m = \texttt{recv}(M);$

$2 : x = (m \geq 1)?\ 1 :\ 0;$

(b)

Log:

$\log(t)|_{V1}$

$\langle V1, 1, \texttt{r}\rangle$

$\langle\langle V1, 2\rangle, \langle M, 1\rangle\rangle$

$\log(t)|_{V2}$

$\langle V2, 1, \texttt{r}\rangle$

$\langle\langle V2, 2\rangle, \langle M, 2\rangle\rangle$

$\log(t)|_{M}$

$\langle\langle V1, 2\rangle, \langle M, 1\rangle\rangle$

$\langle\langle V2, 2\rangle, \langle M, 2\rangle\rangle$

$\langle\langle M, 3\rangle, \langle X, 1\rangle\rangle$

$\log(t)|_{X}$

$\langle\langle M, 3\rangle, \langle X, 1\rangle\rangle$

$\langle X, 2\rangle$

(c)

Causal sequence:

$\vec{\alpha}|_{V1}$

$\langle V1, 1, \texttt{r}\rangle$

$\langle\langle V1, 2\rangle, \langle M, 1\rangle\rangle$

$\vec{\alpha}|_{V2}$

$\vec{\alpha}|_{M}$

$\langle\langle V1, 2\rangle, \langle M, 1\rangle\rangle$

$\langle\langle M, 3\rangle, \langle X, 1\rangle\rangle$

$\vec{\alpha}|_{X}$

$\langle\langle M, 3\rangle, \langle X, 1\rangle\rangle$

$\langle X, 2\rangle$

(d)

Dummified Programs:

Program for $V1$

$1 : v1 = 1;$

$2 : \texttt{send}(M, v1);$

Program for $V2$

$1 : v2 =$ 0 OR 1;

$2 : no - op$

Program for M

$1 : v1 = \texttt{recv}(V1);$

$2 : v2 =$ 0 OR 1;

$3 : \texttt{send}(P, v1 + v2));$

Program for X

$1 : m = \texttt{recv}(M);$

$2 : x = (m \geq 1)?\ 1 :\ 0;$

Violation: All traces where $x = 1$.

Causal analysis: The causal sequence shown here:
$\langle V1_P, 1, \texttt{r}\rangle, \langle\langle V1_P, 2\rangle, \langle M_P, 1\rangle\rangle, \langle\langle M_P, 3\rangle, \langle X_P, 1\rangle\rangle, \langle X_P, 1\rangle$. That is, our definition finds $V1, M$'s choices and actions as a cause.

Figure 3.5: Example 4, Voting machine (Subsets of Z): causal analysis

Remarks.

- In all of these examples, we see that the dummification operation ends up having a similar effect on both the expressions containing internal choice operator, as well as other actions. This is because the allowed values for each of these variables is 0 or 1. In more general settings where variables can take multiple values, the dummification operator would treat the internal choice as well as other actions differently.
- Our definition does not output a complete process as part of a causal sequence; it outputs a subset of the process which is relevant for the violation.

Chapter 4

Relationship with Prior Work in Actual Causation

Chapter goal. Causation has been studied by philosophers, statisticians, social scientists, AI researchers and in numerous other fields since the 1700s [38]. In this chapter, we place our work in the larger context of research in actual causation. We describe earlier theories of actual causation, both in philosophy and AI-based approaches. We first give a brief overview of prior work in process based actual causation. We then discuss four definitions from counterfactual-based actual causation literature, that we find most closely related to our work, and describe the connection with our work. We look at examples used to justify prior theories, describe these in our formalism and highlight the differences in the approaches.

Background. Causal analysis is useful for several applications including fault analysis, understanding causes of outbreaks and troubleshooting, legal reasoning in tort and criminal cases as well as finding explanations [27]. Causation has also been a subject of significant interest in Artificial Intelligence (AI) due to its applicability to finding strategy, prediction and manipulations. In particular, researchers have been interested in questions about how causal information can be acquired in an automated manner and how this information should be processed [27]. On the other hand, philosophers have focused on understanding what empirical evidence warrants a cause-effect relationship and what inferences can be drawn from these relationships [62]. These are a few considerations that have motivated different kinds of research questions for causation over the years.

Broadly, research on causation has focused on three different types of causal inference prob-

lems [27, 29]: Finding causes of effects, *or* finding effects of causes [1] *or* structure learning/ search problems[2]. There has been significant work in the field of inferring causal relations from observational data and representing the relations [See [26, 27, 28] for an overview][3]. Specifically, the research directions have focused on two different yet related aspects, i.e., learning the causal structure using probabilistic correlations and causal assumptions [26] and interpreting the causal structure in order to identify the effects of specific manipulations within the framework [27, 63]. Actual causation is an instance of causes-of-effects problem.

Scope of this section. In this dissertation, we focus on prior work in understanding the causal structure, after the structure has been inferred using causal modeling techniques [26, 27]. There are two relevant lines of work, process causation theories and counterfactual-based theories. We discuss process causation theories briefly. Due to the absence of a mathematical formulation, it is difficult to perform a deeper comparison with the work in this area. We relate the main concepts with our work. Next, we discuss the counterfactual-based theories. In this work we focus on the theories which have been proposed for actual causation, using the structural equation framework. Structural equation models are of significance in several fields as discussed earlier, here we only describe the formalism which has been used in actual causation literature[4].

For problems based on causes-of-effects, it is common to distinguish *actual causation* (or token causation) from *type causation* (relation between variables). Type causation is concerned with relations between random variables and is considered by several philosophers to be the pre-requisite for actual causation. There is, however, no consensus on how type and actual causation are related [28, 29]. The underlying causal relationships can be deterministic or probabilistic. Traditionally, probabilistic approaches have been proposed for type causation whereas actual causation has seen a tradition of counterfactual based approaches[5].

[1] See [63] for details of intervention theory

[2] See [26] for an overview

[3] *'The name 'causal modeling' is often used to describe the new interdisciplinary field devoted to the study of methods of causal inference. This field includes contributions from statistics, artificial intelligence, philosophy, econometrics, epidemiology, psychology, and other disciplines.'[28]*

[4] For a complete overview, see [26, 27]

[5] Eells developed a probabilistic theory of token causation [64]. In general, probabilistic causation based theories, roughly, causes raise the probabilities of their effect. There are identified issues with probabilistic approaches to actual causation, for instance, probability lowering causes, common cause [28].

4.1 Process-based causation theories

There has been significant work in process causation theories for defining actual causation. In contrast with event-based notion of actual causation, causal process theories interpret causation in terms of continuous processes and interactions between them. This line of work is attributed primarily to Salmon. Salmon aimed for scientific explanations that would be in keeping with actualist requirements of empiricism, therefore, his theory does not use counterfactuals [45].

According to Salmon, a process is anything with constancy over time. His theory makes a fundamental distinction between a causal process and a pseudo process where a causal process is defined as one that is capable of transmitting a local modification of a characteristic. For example, a ball moving in he air is a causal process: if we make a cut on the ball modifying its surface, this modification is transmitted by the ball as it moves in the air. This can be contrasted with the example of a spot of light moving across a wall, which is classified as a pseudo process. We can modify the shape of the light spot by distorting the surface of the wall but this modification is not transmitted across the wall as the spot moves. A causal interaction involves the mutual modification of two intersecting processes. For example, the collision of two moving balls, which are two causal processes, is a causal interaction. Both of the balls undergo a modification in momentum as a result of the collision. As the balls move, this modification is transmitted over time.

Even though the intended application domains differ, there appears to be a conceptual analogy between causal process theories and our theory of actual causation. Our programs are generic descriptions of how computation should evolve over time locally, in the absence of interaction, much like physical causal processes of Salmon. Since our intended domain of information processing systems is discrete in nature, processes described by our programs execute in discrete steps, and do not have continuous dynamics. However, they still capture the essence of a causal process that explains how one state of the system leads to another as the system progresses over time. Our programs also allow description of how computation continues from the point of synchronization of two processes, capturing the notion of interaction of in Salmon's theory.

Had our theory and causal process theories been developed for similar domains, a more direct analogy would have been possible. For example, we could envision an extension of our process calculus framework that applies to cyber-physical systems, where capturing the continuous evolution of system state over time is as essential as capturing discrete steps that change system state instantaneously [65, 66, 67]. We believe that such a framework would enable the formal definition of the notion of a causal process and interaction from the philosophy litera-

ture.

It is acknowledged that a causal process is necessary for relating two events as cause and effect but identifying a causal process between two events does not tell us which features of the process are causally relevant to the outcome that we want to explain. Suppose that ball A sinks after being hit by a moving ball B where the moving ball B's surface had been modified with a chalk stain before being thrown. We know that ball B is a causal process because it is capable of transmitting the chalk mark. However, our intuition tells us that the chalk mark on B is not causally relevant to A's sinking. To the best of our knowledge, interpreting causal process theories such that they yield intuitive causal relevance determinations (for instance, the marking of B with the chalk is deemed causally irrelevant to A's sinking) is still a subject of debate. [44, 46]. Our theory may offer some insights into this debate since identifying causal (ir)relevance is a key focus of ours.

4.2 Counterfactual-based actual causation theories

In 1748, Hume [38] identified actual causation with counterfactual dependence—the idea that c is an actual cause of e if had c not occurred then e would not have occurred. While this simple idea does not work if there are independent causes, the counterfactual interpretation of actual causation has been developed further and formalized in a number of influential works (see, for example, [14, 16, 17, 27, 39, 40, 41, 42]).

Actual causation has also inspired interest in legal settings. In particular, Hart and Honoré [33] originally proposed the NESS test (Necessary Elements of a Sufficient Set test) for causation in the legal literature and this approach was worked out in more detail in subsequent work by Wright [42]. The test states that A is a cause of B if it is a necessary element of a set which is sufficient to cause B. This work does not give a clear definition of a sufficient set and is also restricted to deeming single conjuncts (of events) as causes as per the formalization given by Halpern [68][6]. The NESS test shares similarities with prior work by Mackie [41] on INUS test which states that event A is a cause of event B if A is 'an insufficient but necessary part of a condition which is itself unnecessary but sufficient' for B. Further investigations into these definitions and attempts to formalize them in different settings have given rise to several definitions of actual cause in the literature that have subtle differences [27, 40]. One example is Halpern's formalization of the NESS test [68].

We build our definition on similar ideas of counterfactual dependence, as discussed in Chap-

[6]This is problematic in our security setting as we wish to detect collusion and blame multiple agents jointly in such cases.

ter 3.

In this section we discuss four definitions as proposed by Hitchcock (H2001) [17], Hall (H-account) [16], and variants of Halpern-Pearl's definitions (HP2001, HP2005) [14, 30] along with a recent modification to HP definition by Halpern (H2015) [18]. We only discuss the relevant details and interpretations here [18, 29, 56, 61]. For more details we refer the readers to the original papers. Hall and Hitchcock's definition were given using causal paths whereas Halpern and Pearl's definitions directly encode the clauses as constraints on variable values, in the structural equation framework.

Review of causal models. Every theory of actual causation that we discuss in this section was given using causal models, which were discussed in Chapter 2. As a brief recap, a causal model M is an ordered pair $\langle \mathcal{S}, \mathcal{F} \rangle$ where $\mathcal{S}$ represents a set of variables and $\mathcal{F}$ gives a set of structural equations which specify the relationships amongst these variables. $\mathcal{S}$ contains both endogenous variables ($\mathcal{V}$) and exogenous variables ($\mathcal{U}$), as well as the ranges of values for these variables.

Common template. For the causal analysis, the set of endogenous variables $\mathcal{V}$ is split into two disjoint sets $\vec{Z}, \vec{W}$ where the putative cause corresponds to values for a set of variables $\vec{X} \subseteq \vec{Z}$. Here $\vec{X} = \vec{x}$ denotes values for a set of variables. $\vec{Z}$ corresponds to the variables on the path from $\vec{X}$ to the violation φ (which is $\vec{Y} = \vec{y}$ and $Y \in \mathcal{V}$) and $\vec{W}$ corresponds to variables which are not on the path between $\vec{X}$ and $\vec{Y}$. A context u in the model M instantiates all the exogenous variables. Using these values, and the structural equations, we can then derive the values for all the endogenous variables in $\mathcal{V}$. We follow the notation from prior work where writing $M, u \models (X = x)$ denotes that in a model M and context u, the variable X takes the value x. Similarly, writing $M, u \models [W = w](X = x)$ denotes that in a model M and context u, when variable W is fixed at the value w, the variable X takes the value x. Prior work allows interventions on the outcome of any structural equation in the model, i.e., variable values can be set arbitrarily. As a consequence, given a model M, a context u and set of interventions, the variables could take on non-actual values, i.e. values not in the original context u.

Intervening on arbitrary variables can lead to multiple counterfactual scenarios which may or may not be relevant. Each of the definitions either restrict contingencies or prevent contingencies. Following the convention used by Westlake and Livengood [29, 56], we discuss the template for the definitions and then instantiate parts of the template by enumerating different restrictions for different definitions. Most of the definitions have one or more of these three conditions.

An event $\vec{X} = \vec{x}$ is a cause of another event $\vec{Y} = \vec{y}$ if the following conditions hold:

1. *Occurrence*: Given the model M and context u, this condition states that the violation $\vec{Y} = \vec{y}$ actually happened on the log and the putative cause $\vec{X} = \vec{x}$ also holds, i.e. $M, u \models (\vec{X} = \vec{x}) \wedge (\vec{Y} = \vec{y})$.
2. *Necessity and Sufficiency*:
   (a) Necessity: When $\vec{X}$ is changed to $\vec{x'}$ and certain off-path variables $\vec{W}$ are restricted to values $\vec{w'}$ (possibly non-actual), then $\vec{Y} \neq \vec{y}$, i.e. modifying $\vec{X}$ affects $\vec{Y}$.
   (b) Sufficiency: When $\vec{X}$ is restored to its actual value $\vec{x}$ and the off-path variables $\vec{W}$ are restricted to values $\vec{w'}$, *and* the variables $\vec{Z}$ on the path from $\vec{X}$ to $\vec{Y}$ are restricted to their actual values $\vec{z'}$, then $\vec{Y} = \vec{y}$ in all cases. In other words, restoring the putative cause $\vec{X}$ to their actual values on the log, and restoring a subset of other variables to their values on the log, is sufficient to restore $\vec{Y}$ to its value on the log.
3. *Minimality*: $\vec{X}$ is minimal.

Each of the following definitions tests for occurrence and minimality conditions above. The main difference arises from the restrictions in *Necessity and Sufficiency* conditions. For instance, mapping back to the NESS definition discussed earlier, clause $2b$ above corresponds to sufficiency and clause 3 corresponds to necessity. Since our goal is to find the choices (which are a subset of the exogenous values) and the sequence of program expressions which led to a violation, our definition has additional considerations in these clauses.

Remarks. Note that our definition does not contain an explicit clause for necessity. This holds because we do not consider absence of expressions as a cause. For instance, if an expression was not evaluated on the log, our definition will not consider it as a part of $\vec{\alpha}$ since our causal analysis starts with the log. However, the absence of an action can be modeled via choices over data variables, as discussed at the end of the previous chapter (Chapter 2). This restriction does not affect our analysis when we consider security protocols since we capture a large class of safety properties (which require presence of an action), however this is a point of difference from prior work. We discuss this point in detail in Section 4.2.5.2.

Next, we describe the prior definitions in actual causation given using causal models and discuss the relationship with our work.

4.2.1 Hitchcock 2001 (H2001)

In his framework, Hitchcock uses structural equations to represent counterfactual dependence. Roughly, a variable Y counterfactually depends on another variable X in a system of structural

equations, iff, in the given context: $X = x$ and $Y = y$, and there exist values $x' \neq x$ and $y' \neq y$ such that changing the value of X from x to x', changes the value of Y from y to y'. This corresponds to the necessity condition in the template defined above. Hitchcock's definition has two variants. We describe one of the variants below. The second variant is closer in spirit to Hall's definition and is discussed in the next subsection.

Hitchcock's causal theory (referred to as H2001 in the sequel) is based on path analysis where he defines $X = x$ to be a cause of $Y = y$ if there exists a path from X to Y such that, if we fix the values for all off-path variables W to w' (possibly non-actual values), then Y counterfactually depends on X. H2001 distinguishes between a chain of counterfactual dependence and an *active route* between two events, which is crucial for his definition of causation. In order to find an active route from X to Y, we change the value of X and keep its successors fixed and evaluate if Y changes. Hitchcock's analysis focuses on finding a single path of counterfactual dependence between X and Y – this path is called a causal path.

In terms of the template for actual cause definition given above, we fix the values for those variables which are on a path between X and Y other than the direct path being tested[7]. The necessity and sufficiency condition can then be formalized as shown below [17, 18]:

There exists a value x' and $\vec{w}$ represents the actual values for variables in $\vec{W}$ i.e. $(M, \vec{u})| = [\vec{W} = \vec{w}]$ such that

- $(M, \vec{u})| = [\vec{X} = \vec{x'}, \vec{W} = \vec{w}]\ \vec{Y} \neq \vec{y}$

Hitchcock also defines a notion of a *weakly active route* where variables in W are allowed to take non-actual values. In this case, Hitchcock gives a definition which allows variables in $\vec{W}$ to take on non-actual values, however none of these variations affect the values of remaining variables in $\vec{Z}$. This definition is similar to Hall's definition (H-account).

Relationship with our theory. H2001 fixes the values of off-path variables in order to mask their influence on the effect and only test the direct route. Since we focus on uninstantiated program expressions, we do not fix the return values of any of the expressions. Rather, we remove the irrelevant program expressions and test whether the violation occurs along a similar path as on the log. Our definition also captures the idea of testing the same 'path' as on the log, however H2001 implements it by fixing the values of variables (i.e. outcome for processes) on the other paths between the cause and the effect. In contrast, we retain the interaction structure as on the log, but not the actual values.

We demonstrate how our formalism and definition can be used for an example (*Example 5,*

[7]Doing so fixes the value of variables on other routes from X to Y. H2001 defines these as ENF counterfactuals [17] and discusses these in detail.

Backup), given in Figure 4.1, used by Hitchcock [17].

Review: Notation and encoding examples using process calculi. For every example, we first provide the description in text. Next, we provide the structural equations. Endogenous variables are marked with a subscript R. For instance, an event A being 0 or 1 will be represented as A_R to indicate that we are referring to the random variable for structural equations. The exogenous variables are denoted as subscripts of u (for instance u_A, u_B). In our process calculus framework, a process for an identifier A is denoted by A_P. Variables in our process calculus framework are denoted by lowercase alphabets $a, b, \ldots$. The process calculus encoding of the example contains four parts.

- Part **(a)** provides the model of programs corresponding to the example description.
- Part **(b)** specifies the log.
- Part **(c)** provides the causal sequence. The causal sequence is a subsequence of the log.
- Part **(d)** provides the dummified programs. These programs are obtained from the original programs given in part **(a)**, by dummifying the expressions omitted from the causal sequence in **(c)**.

The sequence given in **(c)** is a cause if all executions originating from the (dummified) programs in **(d)**, which contain **(c)** as a subsequence, contain a violation.

Example description. We encode this example using our process calculus framework in Figure 4.1. The four programs in column **(a)** in Figure 4.1 are written using the process calculus syntax.Here T denotes the process for the trainee's process, S denotes the process for the supervisor and V denotes the target's process. Since this is a case where the model allows preemption, therefore we model the example using the asymmetric disjunctive operator. T chooses to shoot and then sends the value to the supervisor and the program for the target. In response, the supervisor sends a value to the target process as well. The final result is evaluated via an asymmetric disjunction. As the log in column **(b)** shows, the trainee shoots and the supervisor does not (looking at the choice resolution in the log).

Example 5 is an instance of preemption (which we discuss in detail in Section 4.4.3) since the trainee's action preempts the supervisor's action. However in this case, the trainee's action could have triggered the supervisor's action as well. H2001 tests whether the target is hit when the trainee fires, keeping the supervisor's action fixed since it is dependent on trainee's action. Note that we will find the trainee's choice and actions as a cause of the violation. However, given the log in part **(b)**, we will not find the supervisor's actions as a cause. This happens since we use an asymmetric disjunction operator and dummifying $\langle V_P, 3, \mathtt{r}\rangle$, modifies $v = s \,[]\, t$ to $v = t$

as on the log.

Example 5 (Backup). *[17] A trainee (T_R) is required to shoot at a target. His supervisor (S_R) is also present- in case the trainee loses his nerve and does not shoot, then the supervisor will shoot. In the actual scenario, the trainee shoots and hits the target ($V_R = 1$).*

T_R, S_R, V_R are the random variables which correspond to the trainee's shooting, supervisor shooting and the target being hit respectively. The structural equations are given as:

$$\begin{aligned} V_R &= T_R \vee S_R \\ S_R &= \neg T_R \\ T_R &= 1 \end{aligned}$$

(a)

Original programs:

Program for T

$1 : t = 0 \oplus 1;$
$2 : \texttt{send}(S_P, t);$
$3 : \texttt{send}(V_P, t);$

Program for S

$1 : t = \texttt{recv}(T_P);$
$2 : s = \neg t;$
$3 : \texttt{send}(V_P, s);$

Program for V

$1 : t = \texttt{recv}(T_P);$
$2 : s = \texttt{recv}(S_P);$
$3 : v = s \; [] \; t;$

(b)

Log:

$\log(t)|_T$
$\langle T_P, 1, \mathtt{r}\rangle$
$\langle\langle T_P, 2\rangle, \langle S_P, 1\rangle\rangle$
$\langle\langle T_P, 3\rangle, \langle V_P, 1\rangle\rangle$

$\log(t)|_S$
$\langle\langle T_P, 2\rangle, \langle S_P, 1\rangle\rangle$
$\langle S_P, 2\rangle$
$\langle\langle S_P, 3\rangle, \langle V_P, 2\rangle\rangle$

$\log(t)|_V$
$\langle\langle T_P, 3\rangle, \langle V_P, 1\rangle\rangle$
$\langle\langle S_P, 3\rangle, \langle V_P, 2\rangle\rangle$
$\langle V_P, 3, \mathtt{r}\rangle$

(c)

Causal sequence:

$\vec{\alpha}|_T$
$\langle T_P, 1, \mathtt{r}\rangle,$
$\langle\langle T_P, 3\rangle, \langle V_P, 1\rangle\rangle$

$\vec{\alpha}|_S$

$\vec{\alpha}|_V$
$\langle\langle T_P, 3\rangle, \langle V_P, 1\rangle\rangle$
$\langle V_P, 3, \mathtt{r}\rangle$

(d)

Dummified programs:

Program for T

$1 : t = 1;$
$2 : no - op$
$3 : \texttt{send}(V_P, t);$

Program for S

$1 : t =$ 0 OR 1 ;
$2 : s =$ 0 OR 1 ;
$3 : no - op$

Program for V

$1 : t = \texttt{recv}(T_P);$
$2 : s =$ 0 OR 1 ;
$3 : v = t;$

Violation: All traces where $v = 1$.

Causal analysis: The causal sequence $\vec{\alpha}$: $\langle T_P, 1, \mathtt{r}\rangle, \langle\langle T_P, 3\rangle, \langle V_P, 1\rangle\rangle, \langle V_P, 3, \mathtt{r}\rangle$, i.e. the choice made by T, the interaction between T and V and then evaluation by V.

Figure 4.1: Example 5: Trainee supervisor example (preemption)

4.2.2 Hall 2007 (H-account)

Hall's theory (called the H-account in the sequel) is also based on path analysis where he defines $X = x$ to be a cause of $Y = y$ if there exists a path from X to Y such that, fixing the

values for off-path variables W to w' (possibly non-actual) does not affect the values for variables in $\vec{Z}$. That is, no variables in $\vec{W}$ can change the value of any of the variables in $\vec{Z}$ while holding $\vec{X}$ fixed at its actual value $\vec{x}$. Here is how the necessity and sufficiency condition can be formalized [16, 18]:

Let $\vec{z}$ represent the actual values for variables in $\vec{Z}$ i.e. $(M, \vec{u})| = [\vec{Z} = \vec{z}]$. There exist values $\vec{x'}, \vec{w'}$ and such that

- $(M, \vec{u})| = [\vec{X} = \vec{x'}, \vec{W} = \vec{w'}]\ \vec{Y} \neq \vec{y}$
- $(M, \vec{u})| = [\vec{W} = \vec{w'}]\vec{Z} = \vec{z}$

Relationship with our theory. Similar to H2001, H-account also finds a relevant path for making a causal determination. However, H-account sets the values of all off-path variables such that they do not affect any of the variable values between the cause and the violation. Since we focus on the existence of choices and actions and not just the outcomes of the processes, in our sufficiency condition we test whether the same sequence of actions is executed on the log. Note that this may lead to different values for the variables on the 'path' between the putative cause and the violation, but that does not affect our analysis.

We demonstrate how our formalism and definition can be used for another example used by Hall [16]. This example is based on late-preemption (*Example 9, Late preemption*) which we discuss in more detail in Section 4.4.3.

Reduction of the model, default and deviants. Hall also discusses the idea that the counterfactual dependence tests should be carried out in a reduction of the original world, i.e. where certain deviant events are replaced by their defaults. Hall also suggests that the alternate contingencies considered in the second condition above should be restricted to those where variables take their default values. Hall defines default values for variables as those which are taken when nothing acts on the system. We actualize a similar idea in our formalism. When we consider alternate contingencies for expressions not in the causal sequence, we remove the expressions from the program syntax and test for all possible returned values– this effectively corresponds to the removed expressions returning arbitrary values[8]. We can further constrain alternate contingencies in our system by allowing restricted choices for function f' which chooses alternate values for choice expressions, to only test for default values[9].

[8]Since removing expressions alone will halt the execution of the program.
[9]This is a subject for future research.

Generality of model. Hall briefly discusses the importance of designing a model which can exhibit a range of actual scenarios as opposed to being specific to only one context. Due to our syntactic constructs, we are able to express general models of interaction more concisely.

As demonstrated in prior work [18], Hall's definition does not work well with the Voting machine example given by Halpern and Pearl (*Example 4, Voting Machine*) in Figure 2.5. In this case, the definition above runs into issues because requiring that no off-path variables should affect any of the variables on the path from cause to the effect, can be too strong [14][10]. For instance, in Example 4, changing the value of either v_1 or v_2 will affect the value of m which is on the path to p.

When we encode Example 4 in our formalism, we find that both V_1 and V_2's choices and interactions can be found as independent causes of the violation $p = 1$ (See Figure 3.5). This difference stems from the fact we do not require the *return values for program expressions* to be the same as the log, rather we constrain the *same expressions* to execute, as in the causal sequence.

4.2.3 Halpern and Pearl (HP2001, HP2005)

Halpern-Pearl 2001 (HP2001). The definitions of Halpern and Pearl also incorporate elements of prior proposals to formally define actual cause in a computational setting. The definitions [14, 30] aimed to improve upon earlier formulations of causality to handle several examples that had proved problematic for earlier definitions and gave precise definitions for necessity and sufficiency. Since then, the structural model approach to causality has been used in various settings, including explaining counter examples generated in model checking [69] and in understanding the completeness of a system specification (in model checking) in terms of covering all system states [70].

The necessity and sufficiency conditions in Halpern and Pearl's initial definition [30] state that:

There exist $\vec{x'}$, $\vec{w'}$ s.t.:

- *Necessity*: When the putative cause $\vec{X}$ is changed to $\vec{x'}$, off-path variables $\vec{W}$ are also changed to $\vec{w'}$, $\vec{Y} \neq \vec{y}$, i.e. $(M, \vec{u})| = [\vec{X} = \vec{x'}, \vec{W} = \vec{w'}]\ \vec{Y} \neq \vec{y}$
- *Sufficiency*: When $\vec{X}$ is restored to its original value $\vec{x}$ and a subset of off-path variables $\vec{W}$ are restricted to values $\vec{w'}$, and the variables $\vec{Z}$ along the path from $\vec{X}$ to $\vec{Y}$ are restricted at actual values $\vec{z'}$, then $\vec{Y} = \vec{y}$ in all resulting cases, i.e. $(M, \vec{u})| = [\vec{X} = \vec{x}, \vec{W} = \vec{w'}, \vec{Z'} = \vec{z'}]\ \vec{Y} = \vec{y}$ for all subsets $\vec{Z'}$ of $\vec{Z}$.

[10]This was given as the reason for justifying subsets of Z requirement in HP2005 [14]

Relationship with our theory. HP2001 was found to be too permissive in the contingencies it allowed [14, 48]. For instance, for our running example (Figure 3.1) in Chapter 2, if E loaded the gun and A did not shoot, HP2001 definition will still find E's action as a cause. This occurs because HP2001 allows changes along all paths. If there exists some variable (in this case E's loading) that is not an actual cause but could turn into one if other variables took non-actual values (if A shot the bullet)- then this non-cause will be converted into a cause [56]. In our case, when we test for alternate contingencies and intervene on any action, then we will test *all possible values* which the variable being instantiated by that action would have taken. Therefore, if E had loaded the gun in the actual scenario, and we were testing whether A's action is relevant, then we will test both possibilities of what happens when A shoots and when it does not.

Halpern-Pearl 2005 (HP2005). Accordingly, the above definition was updated by Halpern and Pearl in 2005 in order to test for all subsets of $\vec{W}$ in the sufficiency condition as well. This addition was made to constrain the alternative values which can be considered for off-path variables. The updated necessity and sufficiency conditions now state: There exists $\vec{x'}$, $\vec{w'}$ s.t.:

- *Necessity*: $(M, \vec{u})| = [\vec{X} = \vec{x'}, \vec{W} = \vec{w'}]\ \vec{Y} \neq \vec{y}$
- *Sufficiency*: When $\vec{X}$ is restored to its original value $\vec{x}$, certain off-path variables W are restricted to values $\vec{w'}$, and the variables $\vec{Z}$ along the path from $\vec{X}$ to $\vec{Y}$ are restricted at actual values $\vec{z'}$, then $\vec{Y} = \vec{y}$ in all resulting cases, i.e. $(M, \vec{u})| = [\vec{X} = \vec{x}, \vec{W'} = \vec{w'}, \vec{Z'} = \vec{z'}]\ \vec{Y} = \vec{y}$ for all subsets $\vec{Z'}$ and $\vec{W'}$ of $\vec{Z}$ and $\vec{W}$, respectively.

Relationship with our theory. Similar to the prior three definitions which we have discussed, HP2005 also constrains the values of outcomes variables for processes. Due to this, HP2005 will find any variable along the causal sequence as a cause (as a single conjunct). Further, allowing a dependent variable value to be modified arbitrarily and testing with an existential quantifier, is problematic since it can find non-causes as causes. This issue is discussed in detail in Section 4.3. Our goal, in contrast, is to find the entire causal sequence.

HP2005 and the prior definitions we have discussed, cannot distinguish between conjunctive and disjunctive causal sequences. When two or more sequences conjunctively cause a violation, prior definitions can select outcome variables for processes on either of the sequences as causes. Accordingly, the intervention method for prior work involves setting any off-path variable (or set of off-path variables) to a single value, and testing the resulting instantiations. The choice of off-path variable values (i.e. values for $\vec{W}$) are not restricted, any value can be chosen and tested in sufficiency. HP2005 counters this over-permissiveness by testing for all subsets of $\vec{W}$.

However, there are still counterexamples since the definition is too permissive in generating counterfactual scenarios [29]. This can be seen in the Voting example (*Example 7, Voting scenario (Stone soup essay)*) described in the next section.

All the definition we have discussed so far propose a test for either subsets of $\vec{W}$ and $\vec{Z}$ in the sufficiency condition. In 2015, Halpern proposed a modification which removes the need to test for the subsets of these values.

4.2.4 Halpern 2015 (H2015)

Halpern's modified definition of causality does not allow values for any variable other than the putative cause to be modified, i.e. the definition freezes values for $\vec{W}$ at their actual values. This removes the need for testing subsets since $\vec{W}$ takes a value as in the original context. $\vec{X} = \vec{x}$ is a cause of $\vec{Y} = \vec{y}$ in context $\vec{u}$ of causal model M if conditions 1 (Occurrence) and 3 (Minimality) hold as in the previous definitions. The second condition is modified as follows:

- There exists a set $\vec{W}$ of variables disjoint from $\vec{X}$ such that $(M, \vec{u})| = [\vec{X} = \vec{x}, \vec{W} = \vec{w}]\ \vec{Y} \neq \vec{y}$, where $\vec{w}$ are the actual values of the variables in $\vec{W}$; i.e., $(M, \vec{u})| = \vec{W} = \vec{w}$.

Note that this definition differs from H2001 since H2015 freezes the values of all variables in $\vec{W}$ at their actual value, while H2001 considers a single causal path [18].

Relationship with our theory. This definition simplifies the checks required in earlier iterations of Halpern-Pearl's definitions and also handles several examples in a more intuitive manner than the previous iterations [18].

However, the causal determinations in case of disjunctive and conjunctive causes seem counterintuitive for our applications. For instance, consider *Example 3, Disjunctive causes* (in Chapter 2) which models a forest fire where either of the fires could have caused the violation. In this case, H2015 will find both the forest fires as a joint cause. On the other hand, in case both the fires were needed for the violation (*Example 2, Forest fire – conjunctive scenario*), then the definition will find these as disjoint independent causes. This differs from our analysis where we find as a joint cause, all the choices and expressions which are part of a single causal sequence. We are interested in finding distinct sequences of program expressions as independent causes.

4.2.5 Relationship with interventions and necessity clause in prior work

4.2.5.1 Intervening on dependent variables

Another point of difference from all the definitions we have discussed so far, is that our interventions on on intermediate 'variables' in the causal sequence, constrain all the constituent expressions in a causal sequence. Modifying any intermediate step in a causal sequence not only propagates the effect to subsequent expressions, but also constrains the values for prior expressions. For instance consider our running example in the previous section (*Example 1, Loader*) where we found A, E's choices and actions as well as B's choices and actions as independent causes of the target being hit (Figure 3.1). In contrast, HP definitions will find all the outcome variables for each of our actions, as causes. For instance, H2015 will find as joint causes, all sets of variables which suffice to set $h = 1$ i.e. H2015 will find $\{A_R, B_R\}, \{E_R, B_R\}$ as causes. Programs or actions for derived variables alone will not show up as causes for our definition. This holds since we don't fix the *values* as found on the log, rather we are interested in restoring *the action* as found on the log. We discuss more examples in Section 4.3.

4.2.5.2 Relationship with necessity clause in prior work

We use Example 1 to understand the necessity clause in prior work. To capture the counterfactual essence of necessity notion, we need to establish the condition that 'had expressions (modeling choices and actions) in $\vec{\alpha}$ been different from what were observed on the log, then the violation would not have occurred.' This implies that in our example above we need to consider alternative hypothetical scenarios in which E, B, A make different choices and execute possibly different actions such that $h = 0$ (the violation involves all traces where $h = 1$). If all of these hypothetical scenarios lead to an evaluation $h = 0$ then we can conclude that the execution of actions in $\vec{\alpha}$ was necessary for the violation to occur in the first place[11]. The question of which alternative programs to execute for E, A, B in the hypothetical scenarios is a challenging one.

Since considering arbitrary hypothetical scenarios can lead to counterintuitive cause determinations, we would like to restrict our attention to scenarios that are more "normal" [68, 71]. The concept of "normal", which has proved to be challenging to define in prior work, takes on a clear definition in a security setting. In the context of security protocols, "normal" could mean a setting which is conducive for the protocol specification to be met or 'benign' behavior of

[11]Note that this condition is trivially satisfied by the correctness proof in security protocols. In addition, unlike prior work, in security we have a well defined notion of norms- the programs which the agents should execute when prescribed by the protocol.

programs which causes no violation[12]. This distinction between default and deviant behavior has also been acknowledged in prior work on causation in philosophy [16, 17].

However, for the examples we consider here, it is not immediately clear what the default or norm value would be. Further, apart from choices, we are also interested in finding which of the other program expressions should be in the causal sequence. Information about default *program expressions* may not always be available. Therefore, we make a design choice of removing program expressions which are not a part of the causal sequence $\vec{\alpha}$, i.e. the hypothetical scenarios proceed as if the expression was not evaluated at al[13]. Note that this design choice implies that absence of an action can be found as a cause only if it is modeled as a choice over data variables, as discussed at the end of Chapter 2.

Distinctive features of our framework. In the next section, we discuss several such general features of our framework and relate these features to prior work. We illustrate these features with the help of examples. In particular, we demonstrate the following definitional features and modeling features.

1. Modeling interaction and choice
2. Combining process-oriented view and counterfactual-based view:
   (a) Finding causal sequences
   (b) Program expressions vs variable assignments as causes: Example 6.
   (c) Testing counterfactual scenarios: Example 7.
3. Consequence: Distinguishing between joint and independent causes: Examples 2, 3.
4. Using process calculus
5. Consequence: Expressing non-deterministic interacting systems concisely: Examples 1, 5.
6. Consequence: Handling preemption concisely: Examples 8, 9, 10.

The first three features are definitional while the last two features are due to the modeling choices.

[12] A natural notion of "normal" programs in the security setting are ones that are prescribed by the protocol for the regular participants and doing nothing for the adversary.

[13] If we are analyzing the violation of a safety property, as discussed in Chapter 5, the necessity condition is trivially satisfied since if no expression is evaluated, there will be no violation.

4.3 Definitional differences and consequences

The goal of our work differs significantly from prior work, hence there are differences in the approach and definitions. We focus on preserving program and interaction-based dependencies in our analysis, which differs from prior work. We do not assume that the processes generating outcomes are fixed or immutable, therefore our interventions require modifying the programs, in conjunction with the inputs. This also differs from prior work where the focus is on evaluating an outcome, given a set of structural equations. The interventions on structural equations fix the outcome, however these interventions do not test the relevance of the structural equations themselves – which is a focus of our work. The analogous notion given below further illustrates the point.

Analogous notion of a causal process using structural equations. A causal process for an event φ will contain all the relevant variables and the relevant structural equations. Informally, we choose a set of variables and their structural equations as part of the putative causal process α'. For the variables not included in α, we remove the structural equation and test for all possible values of the variable. For the variables included in α, the structural equations are not modified. In our definition, we further test whether parts of the structural equation are irrelevant for the causal analysis. Note that in our analysis we do not test variable assignments alone, which is a significant point of difference from prior work.

Combining process-oriented and counterfactual-based view. Sections 4.3.2, 4.3.3, 4.3.4 and 4.3.5 demonstrate how our approach blends the process and counterfactual-based approaches to actual causation and further illustrates the differences discussed above. We find a sequence as a cause, which contains relevant parts of the interacting processes. Additionally, the causal sequence consists of uninstantiated program expressions as opposed to the return values obtained from the evaluated expressions. Finally, we explain how we construct counterfactual scenarios in order to find a causal sequence. As a consequence, we can distinguish between joint and independent causes.

Next we discuss these differences in detail.

4.3.1 Modeling interaction and choice

Our framework treats interaction and choice primitives differently from the other constructs in the syntax. The process of constructing counterfactual scenarios is interaction-aware and

choice-aware as well. This allows us to focus on interactions and choices and use the causal analysis for blame assignment.

4.3.2 Finding causal sequences

Our focus is on finding all distinct causal sequences, which led to a violation. In contrast, prior work has focused on finding individual variable values (i.e. process outcomes) which are sufficient to sustain the effect, even in the presence of specific structural modifications. For instance, earlier definitions of Halpern and Pearl have been shown to find only single conjuncts as causes [72, 73]. In certain cases, it is useful to output the entire causal sequence, since we can provide a more detailed explanation for which choices or which parts of the programs were involved in a violation. This is especially significant in understanding which actions or choices should be held accountable. For instance, in our running example (Example 1, Figure 3.1), E_P loading the gun and A_P shooting, as a sequence is a cause. In contrast, prior work will individually output the events representing the outcome of these processes as causes, i.e. values for E_R, A_R, B_R.

Outputting the entire causal sequence of actions also helps to segregate which factors were relevant for which path. Such fine-grained information is relevant for debugging and protocol design. For instance, consider a modification of Example 1, E loaded A's gun and another agent F loaded B's gun. Prior work will find all events representing the outcomes of the relevant processes as independent causes. In particular, prior definitions will find the four *events* corresponding to E's loading, F's loading, A's shooting and B's shooting as causes. In contrast, we designate the two causal sequences: {E loading, A shooting} and {F loading, B shooting} as independent causes. This distinction allows our causal determinations to not be affected when more independent causes are added to the model, and to accurately pinpoint the specific factors which enabled the distinct causal sequences.

4.3.3 Program expressions vs variable assignments as causes

(Granularity of cause) *Example 6, Shock* [49, 56] demonstrates how finding a causal sequence including relevant parts of a program and not the entire program or the process outcomes, can aid accurate causal determination. In this example, two switches A and B are wired to an electrode, and their original values are the same. If the switch for A is flipped, then the switch for B is always flipped in response, else both the switches stay in their original position. Figure 4.2 shows the structural equations corresponding to the model. Prior definitions find A_R as a cause of the violation $C_R = 1$. This holds because in the counterfactual world where

B_R is fixed at its actual value, C_R will be 0 when A_R is 0. Researchers have been divided about whether or not such switches count as actual causes [27, 56, 74].

In our encoding (Figure 4.2), the process for A makes an internal choice and interacts with B and C to communicate the choice. The process for B sets the same value as a and sends it to C. A show is delivered if both the values received by C_P are 1. The log shown in column (b) indicates that $a = 1$ and hence $b = 1$. As shown in our causal sequence in part (c), we find that the choice made by A_P is not a part of the causal sequence, however other program expressions in A_P are still a part of the causal sequence. As long as A_P interacts with B_P and $b = a$, the interactions shown in $\vec{\alpha}$ will be considered as the minimal sequence of events which is a cause of $c = 1$ on all resulting traces. Our analysis points out that the value generated by A's process is not relevant, however the interaction with B_P is crucial for the violation. This example highlights the importance of finding causes at a fine grained level in order to pinpoint parts of the program which are relevant for the violation[14].

[14]Note that if *how* the shock is delivered is significant, then we can specify $c = (a \wedge b)\ [] \ (\neg a \wedge \neg b)$ instead of $c = (a == b)$ in C_P. With this formalization, our definition will find both A's choices and actions and B's actions as a joint cause.

Example 6 (Shock). *(Mcdermott 1995, [47, 49, 56]) This example is from Westlake [56]:'Two two-state switches are wired to an electrode. The switches are controlled by A and B respectively, and the electrode is attached to C. A has the first option to flip her switch (A = 1). B has the second option to flip her switch (B = 1). The electrode is activated and shocks C (C = 1) iff both switches are in the same position. B wants to shock C, and so flips her switch iff A does.' What is the cause of* $C = 1$*?*

$$\begin{aligned} C_R &\leftarrow (A_R == B_R) \\ B_R &\leftarrow A_R \\ A_R &\leftarrow 1 \end{aligned}$$

(a)

Original programs:

Program for A

$1 : a = 0 \oplus 1;$
$2 : \texttt{send}(B_P, a);$
$3 : \texttt{send}(C_P, a);$

Program for B

$1 : a = \texttt{recv}(A_P);$
$2 : b = a;$
$3 : \texttt{send}(C_P, b);$

Program for C

$1 : a = \texttt{recv}(A_P);$
$2 : b = \texttt{recv}(B_P);$
$3 : c = (a == b);$

(b)

Log:

$\log(t)|_A$
$\langle A_P, 1, \mathbf{r}\rangle$
$\langle\langle A_P, 2\rangle, \langle B_P, 1\rangle\rangle$
$\langle\langle A_P, 3\rangle, \langle C_P, 1\rangle\rangle$

$\log(t)|_B$
$\langle\langle A_P, 2\rangle, \langle B_P, 1\rangle\rangle$
$\langle B_P, 2\rangle$
$\langle\langle B_P, 3\rangle, \langle C_P, 2\rangle\rangle$

$\log(t)|_C$
$\langle\langle A_P, 3\rangle, \langle C_P, 1\rangle\rangle$
$\langle\langle B_P, 3\rangle, \langle C_P, 2\rangle\rangle$
$\langle C_P, 3\rangle$

(c)

Causal sequence:

$\vec{\alpha}|_A$
$\langle\langle A_P, 2\rangle, \langle B_P, 1\rangle\rangle$
$\langle\langle A_P, 3\rangle, \langle C_P, 1\rangle\rangle$

$\vec{\alpha}|_B$
$\langle\langle A_P, 2\rangle, \langle B_P, 1\rangle\rangle$
$\langle B_P, 2\rangle$
$\langle\langle B_P, 3\rangle, \langle C_P, 2\rangle\rangle$

$\vec{\alpha}|_C$
$\langle\langle A_P, 3\rangle, \langle C_P, 1\rangle\rangle$
$\langle\langle B_P, 3\rangle, \langle C_P, 2\rangle\rangle$
$\langle C_P, 3\rangle$

(d)

Dummified programs:

Program for A

$1 : a = 0$ OR $1;$
$2 : \texttt{send}(B_P, a);$
$3 : \texttt{send}(C_P, a);$

Program for B

$1 : a = \texttt{recv}(A_P);$
$2 : b = a;$
$3 : \texttt{send}(C_P, b);$

Program for C

$1 : a = \texttt{recv}(A_P);$
$2 : b = \texttt{recv}(B_P);$
$3 : c = (a == b);$

Violation: All traces where $c = 1$.

Causal analysis: If symmetric disjunction is used in C's program as shown above, our definition finds A's choice to be irrelevant for the violation. The causal sequence consists of A_P's interaction with B_P and the expression which denotes that B_P sets its value to the same as that chosen by A. The interactions with C_P, which are needed to compute the value of c, are also included in the causal sequence.

Figure 4.2: Example 6: Shock

The causal sequence for this example indicates that the evaluation of the expression $\langle B_P, 2\rangle$ is crucial for the violation. Let us consider an alternative interpretation where B also has a choice over whether to choose the same value as A or not. We model this variation in Figure 4.3.

In this case, both A and B's choices are considered relevant for the violation, since dummifying either of the choice expressions will lead to $c = 0$. Outputting the entire causal sequence with the uninstantiated program expressions and relevant choices provides an accurate explanation for the violation in both these variants.

Example 6, Shock

(a)

Original programs:

Program for A

$1 : a = 0 \oplus 1;$
$2 : \texttt{send}(B_P, a);$
$3 : \texttt{send}(C_P, a);$

Program for B

$1 : a = \texttt{recv}(A_P);$
$2 : b = 0 \oplus 1;$
$3 : b' = b \wedge a;$
$4 : \texttt{send}(C_P, s);$

Program for C

$1 : a = \texttt{recv}(A_P);$
$2 : b' = \texttt{recv}(B_P);$
$3 : c = (a == b');$

(b)

Log:

$\log(t)|_A$
$\langle A_P, 1, \mathtt{r}\rangle$
$\langle\langle A_P, 2\rangle, \langle B_P, 1\rangle\rangle$
$\langle\langle A_P, 3\rangle, \langle C_P, 1\rangle\rangle$

$\log(t)|_B$
$\langle\langle A_P, 2\rangle, \langle B_P, 1\rangle\rangle$
$\langle B_P, 2, \mathtt{r}\rangle$
$\langle B_P, 3\rangle,$
$\langle\langle B_P, 4\rangle, \langle C_P, 2\rangle\rangle$

$\log(t)|_C$
$\langle\langle A_P, 3\rangle, \langle C_P, 1\rangle\rangle$
$\langle\langle B_P, 4\rangle, \langle C_P, 2\rangle\rangle$
$\langle C_P, 3\rangle$

(c)

Causal sequence:

$\vec{\alpha}|_A$
$\langle A_P, 1, \mathtt{r}\rangle$
$\langle\langle A_P, 2\rangle, \langle B_P, 1\rangle\rangle$
$\langle\langle A_P, 3\rangle, \langle C_P, 1\rangle\rangle$

$\vec{\alpha}|_B$
$\langle\langle A_P, 2\rangle, \langle B_P, 1\rangle\rangle$
$\langle B_P, 2, \mathtt{r}\rangle$
$\langle B_P, 3\rangle$
$\langle\langle B_P, 4\rangle, \langle C_P, 2\rangle\rangle$

$\vec{\alpha}|_C$
$\langle\langle A_P, 3\rangle, \langle C_P, 1\rangle\rangle$
$\langle\langle B_P, 4\rangle, \langle C_P, 2\rangle\rangle$
$\langle C_P, 3\rangle$

(d)

Dummified programs:

Program for A

$1 : a = 1;$
$2 : \texttt{send}(B_P, a);$
$3 : \texttt{send}(C_P, a);$

Program for B

$1 : a = \texttt{recv}(A_P);$
$2 : b = 1;$
$3 : b' = b \wedge a;$
$4 : \texttt{send}(C_P, s);$

Program for C

$1 : a = \texttt{recv}(A_P);$
$2 : b' = \texttt{recv}(B_P);$
$3 : c = (a == b');$

Violation: All traces where $c = 1$.

Causal analysis: In this variation we find both A and B's choices to be relevant for the violation.

Figure 4.3: Example 6: Variation of the model

4.3.4 Testing counterfactual scenarios

In prior actual causation theories, counterfactual scenarios are constructed by modifying the output of the structural equations. The structural equations can be intervened on, in order to set the variable value, however the existence of structural equations is not questioned in prior work. For instance, for our running example in Figure 3.1, any of the structural equations can be intervened on, such that the value of the variables on the left hand side is instantiated to a different value, irrespective of the values of the variables on the right hand side of the equation. Since the goal of prior work is to find all variable assignments which could have lead to the

violation, therefore the modifications for structural equations can substitute a variable with any allowed value.

On the other hand, since we focus on identifying the entire causal sequence, our approach differs. In our work, we are also interested in finding which of the program expressions are relevant for the violation. Considering traces that are constructed from modified programs, rather than directly modifying the variable assignments for process outcomes on the log, allows us to prevent arbitrary contingencies. This two step process allows us to retain the interaction structure of the log and not remove individual events which could lead to spurious occurrence or removal of the violation[15].

Interventions on variable assignments vs program expressions. In prior work, the counterfactuals consider what value the variables could take, and then restore them to the actual value or constrain them at the fixed value chosen for necessity condition. There are significant differences between our approach to constructing counterfactual scenarios as opposed to prior work:

First, we define interventions such that they are aware of the interactions and do not break the dependence established by the program structure. For instance, when we consider alternate executions of the model as shown in $\vec{\alpha}'$, we test for the case when E loads the gun but A does not shoot; however we do not generate inconsistent contingencies for expressions included in the causal sequence. For instance, a case where both E and A's programs are in the causal sequence yet when E does not load the gun, the value sent out by A cannot be 1. Second, we treat the choice construct differently from actions which allows us to intervene in a different manner. For an action, we construct a counterfactual scenario based on: *whether the action will execute or not.* For an expression involving choice, we construct a counterfactual scenario based on: *could any other value be chosen.* Third, when we intervene on an action, we test the counterfactual condition *for all possible values* that the variable could have been instantiated to. This design choice allows us to test whether the violation depends on the value transmitted by the action or not. This also eliminates spurious causal determinations as testing only specific values could result in the violation (for instance, the violation could be due to other enabled conditions or dependence on a specific data value).

These differences in constructing counterfactual scenarios arise, since we test whether a specific action in the cause set should be executed (as on the log) or not executed at all. Not executing an action is captured by testing for all possible instantiations of the value it would have returned. We focus on *a sequence of program actions and choices* which are essential for

[15]We gave a formal definition of our dummifying transform in Definition 2.

the violation whereas prior work focuses on finding *a set of events* on the log which suffice to replicate the violation. This difference in the end goals motivates different approaches to constructing counterfactual scenarios.

Existential vs universal quantification. Prior work existentially quantifies over the values for off-path variables considered in necessity and the subsequent tests for sufficiency. In certain cases, this can lead to counter-intuitive causal determinations due to over-permissiveness in generating counterfactual scenarios [29, 61]. In contrast, when we intervene on an action, our definition tests for all possible values which the action could have returned. We discuss this difference with the help of an example based on a voting system.

Example 7. Example 7 (Figure 4.4) models a voting example [61]. A and B are two voters who can either vote for C (represented as 1), or vote for D (represented as 2) or abstain from voting (represented as 0). T represents the tabulator which outputs 1 if C wins, 2 if D wins and 0 in case no-one wins. A candidate can win if one of the two outcomes occur: A votes for the candidate or if A abstains and B votes for the candidate. On the log, both A and B vote for C. Note that in our encoding of this example (Figure 4.4), we show how to model a choice over multiple terms by nesting the choice operators. The label $\langle A_P, 1, (\texttt{l}, \texttt{r})\rangle$ over $a = ((0 \oplus 1) \oplus 2)$ denotes choosing the `left` branch between $(0 \oplus 1)$ and 2 and then choosing the `right` branch between 0 and 1. Therefore $a = 1$ in this case.

The example description demonstrates the preference given for A's vote, however prior definitions (H2001, HP2001, HP2005) find both $A_R = 1$ and $B_R = 1$ as independent causes of C's victory ($C_R = 1$) [61]. This is because the counterfactual scenarios considered in sufficiency condition are not restricted. H2015 finds $A_R = 1$ alone as a cause since it only considers contingencies where $B_R = 1$. Note that the following expression in T's program:
$3 : t = (a == 1)?\ 1 :\ ((a == 2)?\ 2 :\ ((b == 1)?\ 1 :\ (\ (b == 2)?\ 2 :\ 0)));$
combines the following expressions in A-normal form:
$3 : v1 = (b == 2)?\ 2 :\ 0;$
$4 : v2 = (b == 1)?\ 1 :\ v1;$
$5 : v3 = (a == 2)?\ 2 :\ v2;$
$6 : t = (a == 1)?\ 1 :\ v3;$

We use the conditional operator to model the priority given to A's vote[16]. When we intervene on any action, we ensure that it does not stop the *progress* of the rest of the sequence. If we are interested in finding whether A's choice and action alone are a cause, we *dummify*

[16]Note that if we only want to consider alternate scenarios with default settings, we can reduce choices for a, b. This will affect the causal outcome.

B's choice and remove its interaction with T. What this means is that we test for all possible choices which B is allowed to pick, and we test for all possible values for the input that T could receive from B[17].

Now for Example 7, B's actions and choices alone are not a cause since the value of t depends on A. When we dummify the choices and actions for A_P, then the value a received by the program for T could be either 0 or 1 or 2. Since $t \neq 1$ for all such logs, B's actions alone are not a cause.

[17]Note that the instantiation of these variables is done independently once we dummify the interaction, i.e. the value sent by B and the value received by T can be instantiated independently if the interaction has been dummified. We will quantify over all such instantiations.

Example 7 (Voting scenario (Stone soup essay)). *A and B have three mutually exclusive choices, to vote for C, or for D, or not to vote. An option wins if A votes for it, or if B votes for it and A does not vote. A and B both vote for C.*

The structural equations (C_R and D_R represent whether candidate C or D wins):

$$C_R \leftarrow (A_R == 1) \vee ((A_R == 0) \wedge (B_R == 1))$$
$$D_R \leftarrow (A_R == 2) \vee ((A_R == 0) \wedge (B_R == 2))$$

In the programs given below, a, b can be 0 (abstain), 1 (vote for C), 2 (vote for D). t gives the outcome.

(a)

Original programs:

Program for A

$1 : a = ((0 \oplus 1) \oplus 2);$
$2 : \texttt{send}(T, a);$

Program for B

$1 : b = ((0 \oplus 1) \oplus 2);$
$2 : \texttt{send}(T, b);$

Program for T

$1 : a = \texttt{recv}(A);$
$2 : b = \texttt{recv}(B);$
$3 : t = (a == 1)?\ 1 :$
$\quad ((a == 2)?\ 2 :$
$\quad ((b == 1)?\ 1 :$
$\quad (\ (b == 2)?\ 2 :\ 0)));$

(b)

Log:

$\log(t)|_A$
$\langle A, 1, (1, \mathtt{r})\rangle$
$\langle\langle A, 2\rangle, \langle T, 1\rangle\rangle$

$\log(t)|_B$
$\langle B, 1, (1, \mathtt{r})\rangle$
$\langle\langle B, 2\rangle, \langle T, 2\rangle\rangle$

$\log(t)|_T$
$\langle\langle A, 2\rangle, \langle T, 1\rangle\rangle$
$\langle\langle B, 2\rangle, \langle T, 2\rangle\rangle$
$\langle T, 3\rangle$

(c)

Causal sequence:

$\vec{\alpha}|_A$
$\langle A, 1, (1, \mathtt{r})\rangle$
$\langle\langle A, 2\rangle, \langle T, 1\rangle\rangle$

$\vec{\alpha}|_B$

$\vec{\alpha}|_T$
$\langle\langle A, 2\rangle, \langle T, 1\rangle\rangle$
$\langle T, 3\rangle$

(d)

Dummified programs:

Program for A

$1 : a = 1;$
$2 : \texttt{send}(T, a);$

Program for B

$1 : b =$ 0 OR 1 OR 2;
$2 : no - op;$

Program for T

$1 : a = \texttt{recv}(A);$
$2 : b =$ 0 OR 1 OR 2;
$3 : t = (a == 1)?\ 1 :$
$\quad ((a == 2)?\ 2 :$
$\quad ((b == 1)?\ 1 :$
$\quad (\ (b == 2)?\ 2 :\ 0)));$

Violation: All traces where $t = 1$.

Causal analysis: Our definition finds A_P's choice and interactions with T_P as a cause of the violation. As long as $a = 1$, t will be 1. However, when $b = 1$ and we dummify A's choice and actions, t will not be 1 on all traces.

Figure 4.4: Example 7: Voting Scenario (stone soup essay)

4.3.5 Distinguishing between joint and independent causes

While developing our definition, a criterion that we want our definition to satisfy is its ability to distinguish between independent and joint causes. Our ultimate goal is to use the notion of actual cause as a building block for accountability – the independent vs. joint cause distinction

is significant when making finer deliberations as to what agent is accountable in what proportion and punishing the violators. Joint causation is also significant in a legal perspective [75]. For instance, it is useful for holding liable a group of agents working together when none of them satisfy the cause criteria individually but together their actions are be found to be a cause.

Our definition can distinguish between joint from independent causes, due to the following features: (i) Focus on sequences of program expressions as opposed to individual variable assignments, (ii) an interaction-aware approach to constructing counterfactual scenarios, and considering all possible counterfactual scenarios (universal quantification) for expressions not in the putative causal sequences, and (iii) testing for minimality.

Prior definitions do not distinguish between joint and independent causes since the focus is on variable assignments and most of the definitions existentially choose an assignment to the variables not in the putative cause set (i.e. in $\vec{W}$)[18].

For instance, in Chapter 3, we model conjunctive scenarios (Example 2, Figure 3.4) and disjunctive scenarios (Example 3, Figure 3.3) for the same example based on a forest fire. If the forest fire depends on both the lightning and match being lit, then we refer to it as the conjunctive scenario. If the forest fire depends on either of the events happening, then we refer to it as a disjunctive scenario.

In case of conjunctive scenarios (Figure 3.4), our definition finds the choices and actions in the programs for both L and ML as a joint cause[19], i.e. our definition will find the sequence $\vec{\alpha}$ as a cause, where $\vec{\alpha}$ is :
$\langle L_P, 1, \mathtt{r}\rangle, \langle ML_P, 1, \mathtt{r}\rangle, \langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle, \langle\langle ML_P, 2\rangle, \langle F_P, 2\rangle\rangle, \langle F_P, 3\rangle$.

In contrast, in the disjunctive scenarios (Figure 3.3), our definition will find two different causal sequences, i.e. L_P's choice and interaction with F_P will form one causal sequence and ML_P's choice and interaction with F_P will be another distinct causal sequence. The two causal sequences are:

- $\langle L_P, 1, \mathtt{r}\rangle, \langle\langle L_P, 2\rangle, \langle F_P, 1\rangle\rangle, \langle F_P, 3\rangle$,
- $\langle ML_P, 1, \mathtt{r}\rangle, \langle\langle ML_P, 2\rangle, \langle F_P, 1\rangle\rangle, \langle F_P, 3\rangle$,

We contrast this analysis with prior work. For the conjunctive scenario HP2001, HP2005 will find both $ML_R = 1$ and $L_R = 1$ as distinct causes. When the same definitions are applied to the disjunctive case, again the output will be ML_R and L_R as distinct causes. H2015 will output $\{L_R, ML_R\}$ jointly, in case of *disjunctive scenario* (Figure 2.4) and will output L_R and ML_R as distinct causes in case of *conjunctive scenario* (Figure 3.4). For all these definitions, our

[18]The definition of strongly sufficient causation [14] considers universal quantification over variables in $\vec{W}$, however it does not capture the joint causal sequences since the other conditions are not satisfied.

[19]Note that the internal choices can be made in any order. This does not affect our causal inference.

causal sequence output differs significantly and is better suited for our purpose.

4.4 Modeling differences and consequences

4.4.1 Using process calculus

We discussed the advantages of using process calculus-based framework for modeling interacting systems in Chapter 2. Next, we discuss two specific consequences of using this framework.

4.4.2 Expressing concise general models of interaction

We focus on finding uninstantiated program expressions as part of a causal sequence. Therefore we require a general model which can express the effect of several interventions, that include changing the internal choices, and considering the absence of certain actions on the log.

As discussed in Section 2.4, process calculus naturally encodes general models of interaction. As a result, our model of programs is general and the log resolves the choices and records the synchronization structure. This is in contrast with prior work, where the structural equations have been used to model the specifics (*'model causal network according to context of interest'* [16]). For instance in our running example, *Example 1, Loader* (Figure 3.1), structural equations state that A will always fire if E loads. In our formalism, one can naturally express the fact if E loads, A can or cannot fire, but the conditions are set such that it can. All of these scenarios can be depicted with the same model.

4.4.3 Handling preemption concisely

Preemption occurs when two processes execute on a log such that each of these could independently be a cause of a violation, however one of the processes preempts the other and causes the effect while the other 'waits in reserve'[20]. Prior work [57] has distinguished between symmetric overdetermination (or independent causes) and preemption [47, 49]. In case of former, the two causes are indeed symmetric in every respect and both have an equal claim to be regarded as the cause of an effect. In contrast, preemption involves asymmetry in the two processes which could have caused the violation. Several types of preemption have been discussed in the philosophy literature. We consider some prominent examples and model these in our framework for the following types of preemption:

[20]Preemption refers to a class of examples which were suggested against Lewis's counterfactual theory [27, 47, 49]

- Trumping preemption
- Temporal preemption: Early preemption/ late preemption

Trumping preemption occurs when one cause trumps the other due to a pre-defined priority. *Early preemption* occurs if two different programs execute on the log, however the path from the preempted cause is cut off before it could cause the violation. On the other hand, *late preemption* occurs if two different actions execute on the log, but the path from the preempted cause is not cut off- the preempted cause would have caused the violation had the actual cause not done so (the effect would have shown). We can express both cases of early and late preemption using the same model. Further, we can deal with examples of trumping preemption by encoding the priority via the conditional operator, and using the asymmetric disjunctive operator.

Trumping preemption. Consider Example 8. Here, both a major and a sergeant can give an order to charge and a corporal obeys the order and shoots. Prior work [57, 76] claims that the major's order *trumps* the sergeant's order since the major is a higher ranking official. Therefore, if both the major and the sergeant were to give the command to charge, the major, and not the sergeant, would be the cause. We encode the priority for major's order; $c = 1$ if either the major's shouts charge ($m = 1$) *or* if the major does not say anything ($m = 0$) and sergeant shouts charge ($s = 1$). If the major shouts 'not charge' then the sergeant's order is irrelevant. Our model accurately captures this preference as shown in Figure 4.5.

Prior definitions (H2001, H-account, HP2001, HP2005) will find both major and sergeant's orders as causes unless an additional variable is used to distinguish the outcomes. Our definition will only find the major's order as a cause if both of them shouted 'charge' on the log. By encoding the priority and using the asymmetric disjunctive operator, we can accurately distinguish between the two paths.

Example 8 (Trumping preemption). *A major and sergeant both stand before a corporal and shout 'Charge!' 'Imagine that . . . the major and the sergeant stand before the corporal, both shout "Charge!" at the same time, and the corporal decides to charge.' [HP2005, Bas van Fraassen, Schaffer [57]]*

$$C \leftarrow M \vee S$$

(a) Original programs:	(b) Log:	(c) Causal sequence:	(d) Dummified programs:
Program for M	$\log(t)\vert_M$	$\vec{\alpha}\vert_M$	**Program for M**
$1 : m = (0 \oplus 1);$	$\langle M, 1, \mathtt{r}\rangle$	$\langle M, 1, \mathtt{r}\rangle$	$1 : m = 1;$
$2 : \mathtt{send}(C, m);$	$\langle\langle M, 2\rangle, \langle C, 1\rangle\rangle$	$\langle\langle M, 2\rangle, \langle C, 1\rangle\rangle$	$2 : \mathtt{send}(C, m);$
Program for S	$\log(t)\vert_S$	$\vec{\alpha}\vert_S$	**Program for S**
$1 : s = (0 \oplus 1);$	$\langle S, 1, \mathtt{r}\rangle$		$1 : s =$ 0 OR 1;
$2 : \mathtt{send}(C, s);$	$\langle\langle S, 2\rangle, \langle C, 2\rangle\rangle$		$2 : no - op$
Program for C	$\log(t)\vert_C$	$\vec{\alpha}\vert_C$	**Program for C**
$1 : m = \mathtt{recv}(M)$	$\langle\langle M, 1\rangle, \langle C, 1\rangle\rangle$	$\langle\langle M, 1\rangle, \langle C, 1\rangle\rangle$	$1 : m = \mathtt{recv}(M)$
$2 : s = \mathtt{recv}(S);$	$\langle\langle S, 2\rangle, \langle C, 2\rangle\rangle$		$2 : s =$ 0 OR 1;
$3 : c = m \,[]\, s;$	$\langle C, 3, 1\rangle$	$\langle C, 3, 1\rangle$	$3 : c = m;$

Violation: All traces where $c = 1$.

Causal analysis: Our definition finds only M's choice and actions (including interaction with C) as a cause for the given log.

Figure 4.5: Example 8: Trumping Preemption involving shooting (priority)

Early and late preemption. Prior definitions deal with preemption by using a variable to take on different values depending on the chosen path. We can deal with different types of preemption by incorporating priority rules (conditional statement in evaluating the expression) and by using asymmetric disjunction operator. We can incorporate choice in modeling the context (for instance, the programs can either choose a 0 or 1 for corresponding exogenous variables) and in modeling the preference given to one causal path over another. We demonstrate this with the help of an example of early and late preemption in Example 9 in Figure 4.6.

Note: In Figure 4.6, we depart from our convention of showing four columns with the original programs, the log, the causal sequence and the dummified programs. In this case we show the original programs and two logs: one denoting early preemption and late preemption.

We show the causal sequence and the dummified programs as part of the next example on temporal preemption (Example 10, Figure 4.7).

The structural equations and the encoding for Example 9 can be found in Figure 4.6. Part (a) gives the structural equations, part (b) gives the corresponding programs. Part (c) shows a log which will correspond to *early preemption* and part (d) shows a log for the same model which corresponds to *late preemption.* In prior work, a separate variable captures the effect of the actual cause and is used to model which of the two paths (i.e. path from the preempted cause and path from the the actual cause, to the violating event) was taken [14]. However if we model preemption based examples using an asymmetric disjunction operator, then only those counterfactuals will be considered on which the same path is chosen as on the log. For instance, in Example 9, in case of early preemption (Figure 4.6, part (c)), a victim drinks a poisoned cup and is shot later. We explicitly model the *time* t elapsed between the poisoning and the shooting of the canteen. Since this value will be fixed, therefore, it will be a part of the initial configuration. If the time t is less than the time taken for the poison to take effect (δ), then we will find shooting as a cause, else we will find poisoning as a cause. Note that in both cases, we do not need to add an extra variable which takes different value based on the effect, rather we choose to model time as an explicit factor. As a result we will also find the time factor in the causal sequence.

Example 9 (Late preemption). *Paula poisons a cup of tea which Victoria drinks. However before the poison can take effect, Sharon shoots her and Victoria dies. What is the cause of Victoria's death? [15, 42].*

$$D \leftarrow P \vee S$$

(a)
General model

Program for P
$1 : p = 0 \oplus 1;$
$2 : \texttt{send}(V, p);$

Program for S
$1 : s = 0 \oplus 1;$
$2 : \texttt{send}(V, s);$

Program for T
$1 : \texttt{send}(V, t);$

Program for V
$1 : p = \texttt{recv}(P);$
$2 : s = \texttt{recv}(S);$
$3 : t = \texttt{recv}(T);$
$4 : x1 = p \wedge (t \geq \delta);$
$5 : x2 = s \wedge (t < \delta);$
$6 : v = x1 \; [] \; x2;$

(b)
Log for early preemption
$t \geq \delta$ (initial config)

$\log(t)|_P$
$\langle P, 1, \mathbf{r} \rangle$
$\langle \langle P, 2 \rangle, \langle V, 1 \rangle \rangle$

$\log(t)|_S$
$\langle S, 1, \mathbf{r} \rangle$
$\langle \langle S, 2 \rangle, \langle V, 2 \rangle \rangle$

$\log(t)|_T$
$\langle \langle T, 1 \rangle, \langle V, 3 \rangle \rangle$

$\log(t)|_V$
$\langle \langle P, 2 \rangle, \langle V, 1 \rangle \rangle$
$\langle \langle S, 2 \rangle, \langle V, 2 \rangle \rangle$
$\langle \langle T, 1 \rangle, \langle V, 3 \rangle \rangle$
$\langle V, 4 \rangle$
$\langle V, 5 \rangle$
$\langle V, 6, \mathbf{l} \rangle$

(c)
Log for late preemption
$t < \delta$ (initial config)

$\log(t)|_P$
$\langle P, 1, \mathbf{r} \rangle$
$\langle \langle P, 2 \rangle, \langle V, 1 \rangle \rangle$

$\log(t)|_S$
$\langle S, 1, \mathbf{r} \rangle$
$\langle \langle S, 2 \rangle, \langle V, 2 \rangle \rangle$

$\log(t)|_T$
$\langle \langle T, 1 \rangle, \langle V, 3 \rangle \rangle$

$\log(t)|_V$
$\langle \langle P, 2 \rangle, \langle V, 1 \rangle \rangle$
$\langle \langle S, 2 \rangle, \langle V, 2 \rangle \rangle$
$\langle \langle T, 1 \rangle, \langle V, 3 \rangle \rangle$
$\langle V, 4 \rangle$
$\langle V, 5 \rangle$
$\langle V, 6, \mathbf{r} \rangle$

Violation: All traces where $v = 1$.

Modeling: Programs modeling time t elapsed between poisoning and shooting. δ captures the time it takes for poisoning to take effect. We can encode both early and late preemption as instances of same model.

Figure 4.6: Example 9: Poisoning (Late Preemption/Early preemption).

Next, we consider another example of overdetermination/preemption: *Example 10, Rock throwing example* in Figures 4.7 and 4.8. Billy and Suzy throw rocks at bottle and the bottle shatters. Similar to *Example 3, Disjunctive causes*, there are two ways in which we can model this scenario: either we allow for the possibility that both the throws can be independent causes

or we specify that only one of the rocks can hit. We demonstrate how an example encoding the structural equations for late preemption will be analyzed using our definitions.

If we model the initial set of structural equations, i.e. $BS_R = ST_R \vee BT_R$, then the model will be isomorphic to the disjunctive scenario in Example 3. In Figure 4.7, we show that we can encode the model without the need to introduce extra variables variables or by modeling a preference for Suzy's throw or Billy's throw. In this case, if additional evidence is not accessible, then the modeler can specify $bs = st \ \vee \ bt$ else, then we can capture preemption by modeling $bs = st \ [] \ bt$. In contrast, prior work has dealt with distinguishing the two preempting factors by adding extra variables to distinguish the relevant cause from the preempted cause, which we discuss next.

4.4.3.1 Modeling preemption with a symmetric disjunction

Consider a different model for Example 10. Prior work has dealt with distinguishing the two preempting factors by adding extra variables for Suzy's hit (SH_R) and Billy's hit (BH_R). Accordingly, the structural equations are modified, which we model in Figure 4.8 and Figure 4.9. Note that this encoding builds in the preference for Suzy's throw as we encode the modified structural equations. For $\vec{\alpha}$ shown in Figure 4.8, as long as $st = 1$ and the interaction structure between ST, SH and BS is preserved, $bs = 1$.

Example 10 (Rock throwing example). *(Hall [43]) Suzy and Billy both are expert rock throwers, i.e. if they throw a rock, they are always on target. Both of them throw a rock at a bottle. Suzy throws first however Billy's rock would have shattered the bottle had Suzy's throw not occurred. What is the cause of the bottle being shattered?*

$$BS_R \leftarrow ST_R \vee BT_R$$

(a)

Program for ST_P

$1 : st = 0 \oplus 1;$

$2 : \texttt{send}(BS, st);$

Program for BT_P

$1 : bt = 0 \oplus 1;$

$2 : \texttt{send}(BS, bt);$

Program for BS_P

$1 : st = \texttt{recv}(ST);$

$2 : bt = \texttt{recv}(BT);$

$3 : bs = st \; [] \; bt;$

(b)

$\log(t)|_{ST}$

$\langle ST, 1, \mathtt{r} \rangle$

$\langle \langle ST, 2 \rangle, \langle BS, 1 \rangle \rangle$

$\log(t)|_{BT}$

$\langle BT, 1, \mathtt{r} \rangle$

$\langle \langle BT, 2 \rangle, \langle BS, 2 \rangle \rangle$

$\log(t)|_{BS}$

$\langle \langle ST, 1 \rangle, \langle BS, 1 \rangle \rangle$

$\langle \langle BT, 2 \rangle, \langle BS, 2 \rangle \rangle$

$\langle BS, 3, \mathtt{1} \rangle$

(c)

$\vec{\alpha}|_{ST}$

$\langle ST, 1, \mathtt{r} \rangle$

$\langle \langle ST, 2 \rangle, \langle BS, 1 \rangle \rangle$

$\vec{\alpha}|_{BT}$

$\vec{\alpha}|_{BS}$

$\langle \langle ST, 2 \rangle, \langle BS, 1 \rangle \rangle$

$\langle BS, 3, \mathtt{1} \rangle$

(d)

Program for ST_P

$1 : st = 1;$

$2 : \texttt{send}(BS, st);$

Program for BT_P

$1 : bt =$ 0 OR 1 ;

$2 : no - op$

Program for BS_P

$1 : st = \texttt{recv}(ST);$

$2 : bt =$ 0 OR 1 ;

$3 : bs = st;$

Figure 4.7: Example 10: Late preemption without additional variables to distinguish outcome.

Example 10, Rock throwing example: Suzy and Billy both are expert rock throwers, i.e. if they throw a rock, they are always on target. Both of them throw a rock at a bottle. Suzy throws first however Billy's rock would have shattered the bottle had Suzy's throw not occurred. What is the cause of the bottle being shattered?

$$BS_R \leftarrow SH_R \vee BH_R$$
$$BH_R \leftarrow BT_R \wedge \neg SH_R$$
$$SH_R \leftarrow ST_R$$

(a) Original programs:

Program for ST_P
$1 : st = 0 \oplus 1;$
$2 : \texttt{send}(SH, st);$

Program for BT_P
$1 : bt = 0 \oplus 1;$
$2 : \texttt{send}(BH, bt);$

Program for SH_P
$1 : sh = \texttt{recv}(ST);$
$2 : \texttt{send}(BS, sh);$
$3 : \texttt{send}(BH, sh);$

Program for BH_P
$1 : bt = \texttt{recv}(BT);$
$2 : sh = \texttt{recv}(SH);$
$3 : bh = bt \wedge \neg sh;$
$4 : \texttt{send}(BS, bh);$

Program for BS_P
$1 : sh = \texttt{recv}(SH);$
$2 : bh = \texttt{recv}(BH);$
$3 : bs = sh \vee bh;$

(b) Log:

$\log(t)|_{ST}$
$\langle ST, 1, \mathbf{r}\rangle$
$\langle\langle ST, 2\rangle, \langle SH, 1\rangle\rangle$

$\log(t)|_{BT}$
$\langle BT, 1, \mathbf{r}\rangle$
$\langle\langle BT, 2\rangle, \langle BH, 1\rangle\rangle$

$\log(t)|_{SH}$
$\langle\langle ST, 2\rangle, \langle SH, 1\rangle\rangle$
$\langle\langle SH, 2\rangle, \langle BS, 1\rangle\rangle$
$\langle\langle SH, 3\rangle, \langle BH, 2\rangle\rangle$

$\log(t)|_{BH}$
$\langle\langle BT, 2\rangle, \langle BH, 1\rangle\rangle$
$\langle\langle SH, 3\rangle, \langle BH, 2\rangle\rangle$
$\langle BH, 3\rangle$
$\langle\langle BH, 4\rangle, \langle BS, 2\rangle\rangle$

$\log(t)|_{BS}$
$\langle\langle SH, 2\rangle, \langle BS, 1\rangle\rangle$
$\langle\langle BH, 4\rangle, \langle BS, 2\rangle\rangle$
$\langle BS, 3\rangle\rangle$

(c) Causal sequence:

$\vec{\alpha}|_{ST}$
$\langle ST, 1, \mathbf{r}\rangle$
$\langle\langle ST, 2\rangle, \langle SH, 1\rangle\rangle$

$\vec{\alpha}|_{BT}$

$\vec{\alpha}|_{SH}$
$\langle\langle ST, 2\rangle, \langle SH, 1\rangle\rangle$
$\langle\langle SH, 2\rangle, \langle BS, 1\rangle\rangle$

$\vec{\alpha}|_{BH}$

$\vec{\alpha}|_{BS}$
$\langle\langle SH, 2\rangle, \langle BS, 1\rangle\rangle$
$\langle BS, 3\rangle\rangle$

(d) Dummified programs:

Program for ST_P
$1 : st = 1;$
$2 : \texttt{send}(SH, st);$

Program for BT_P
$1 : bt =$ 0 OR 1 ;
$2 : no - op;$

Program for SH_P
$1 : sh = \texttt{recv}(ST);$
$2 : \texttt{send}(BS, sh);$
$3 : no - op$

Program for BH_P
$1 : bt =$ 0 OR 1 ;
$2 : sh =$ 0 OR 1 ;
$3 : bh =$ 0 OR 1 ;
$4 : no - op$

Program for BS_P
$1 : sh = \texttt{recv}(SH);$
$2 : bh =$ 0 OR 1 ;
$3 : bs = sh \vee bh;$

Figure 4.8: Example 10: Over-determination with five programs, as described in the original structural equations. This model captures the fact that if Suzy hits first, her throw gets preference.

If we consider another putative cause, $\vec{\alpha}'$ in the same model (Figure 4.8), which contains the choice made by BT and the interactions between BT, BH and BS, then we will find $\vec{\alpha}'$ as a

cause too. In this case, we find that the actual value sent by ST_P or received by SH_P does not matter. However, as long as the interaction structure is retained as on the log, we will obtain a violation. This structure is similar to Example 6 and occurs because our definitions will find all distinct causal sequences at the fine-grained level of program expressions.

Note that when the above example is modeled structural equations, even HP2005's necessity and sufficiency conditions will be satisfied by the values for the set $\{BT_R, SH_R\}$. However the reason this set is not a valid cause is because prior work allows values for intermediate variables along a causal sequence to be a cause and hence $SH_R = 1$ is a cause of $BS_R = 1$ independently. As a consequence, due to minimality, prior work does not find $BT_R = 1$ alone as a cause. For our process calculus framework, we cannot find the expression instantiating the value of sh in SH_P as a cause without finding ST_P as a cause since there is a dependency between the two expressions. In this case, modeling preemption with an asymmetric disjunction operator as shown in Figure 4.7 solves the issue.

Example 10 can also be used to highlight other points made in this section:

- Finding causal sequences: In the modified description, where extra variables are added (Figure 4.8), our definition will find the expressions in ST_P, SH_P and BS_P as a causal sequence, i.e. a sequence of actions and choices.
- Program expressions vs variable assignments as causes (Granularity of cause): In Figure 4.8, our definition points out parts of BS_P which are relevant for the causal sequence described in the previous point. Our definition pinpoints relevant expressions within a program as opposed to only testing the entire program or focusing on its outcome.
- Testing for counterfactual scenarios: In Figures 4.7 and 4.8, when we remove Billy's choice and actions from the causal sequence, we test for all allowed choices for bt in the program BT_P, and for all possible instantiations of bh, as received by the program BS_P. Testing for all instantiations in this manner allows us to test whether the causal sequence depends on the removed expressions merely for progress or if any of the removed expressions are relevant for the violation.
- Distinguishing between joint and independent causes: In Figure 4.8, if the symmetric disjunction is used, then our definition will find Suzy's throw and Billy's throw as independent causes. These can be distinguished from individual causal sequences.
- Expressing general models for interacting systems more concisely: As shown in Figure 4.7, can model the example such that we do not need to build in the preference for one throw over another. We can model multiple contexts and capture preemption without adding additional variables. Instead, we utilize the asymmetric disjunction operator

and the log.

Considering a different $\vec{\alpha}''$, *Example 10, Rock throwing example*

(a)

Original programs:

Program for ST_P
$1 : st = 0 \oplus 1;$
$2 : \texttt{send}(SH, st);$

Program for BT_P
$1 : bt = 0 \oplus 1;$
$2 : \texttt{send}(BH, bt);$

Program for SH_P
$1 : sh = \texttt{recv}(ST);$
$2 : \texttt{send}(BS, sh);$
$3 : \texttt{send}(BH, sh);$

Program for BH_P
$1 : bt = \texttt{recv}(BT);$
$2 : sh = \texttt{recv}(SH);$
$3 : bh = bt \wedge \neg sh;$
$4 : \texttt{send}(BS, bh);$

Program for BS_P
$1 : sh = \texttt{recv}(SH);$
$2 : bh = \texttt{recv}(BH);$
$3 : bs = sh \vee bh;$

(b)

Log:

$\log(t)|_{ST}$
$\langle ST, 1, \mathtt{r}\rangle$
$\langle\langle ST, 2\rangle, \langle SH, 1\rangle\rangle$

$\log(t)|_{BT}$
$\langle BT, 1, \mathtt{r}\rangle$
$\langle\langle BT, 2\rangle, \langle BH, 1\rangle\rangle$

$\log(t)|_{SH}$
$\langle\langle ST, 1\rangle, \langle SH, 1\rangle\rangle$
$\langle\langle SH, 2\rangle, \langle BS, 1\rangle\rangle$
$\langle\langle SH, 3\rangle, \langle BH, 2\rangle\rangle$

$\log(t)|_{BH}$
$\langle\langle BT, 2\rangle, \langle BH, 1\rangle\rangle$
$\langle\langle SH, 3\rangle, \langle BH, 2\rangle\rangle$
$\langle BH, 3\rangle$
$\langle\langle BH, 4\rangle, \langle BS, 2\rangle\rangle$

$\log(t)|_{BS}$
$\langle\langle SH, 2\rangle, \langle BS, 1\rangle\rangle$
$\langle\langle BH, 4\rangle, \langle BS, 2\rangle\rangle$
$\langle BS, 3\rangle\rangle$

(c)

Causal sequence:

$\vec{\alpha}''|_{ST}$

$\vec{\alpha}''|_{BT}$
$\langle BT, 1, \mathtt{r}\rangle$
$\langle\langle BT, 2\rangle, \langle BH, 1\rangle\rangle$

$\vec{\alpha}''|_{SH}$
$\langle\langle SH, 2\rangle, \langle BS, 1\rangle\rangle$
$\langle\langle SH, 3\rangle, \langle BH, 2\rangle\rangle$

$\vec{\alpha}''|_{BH}$
$\langle\langle BT, 2\rangle, \langle BH, 1\rangle\rangle$
$\langle\langle SH, 3\rangle, \langle BH, 2\rangle\rangle$
$\langle BH, 3\rangle$
$\langle\langle BH, 4\rangle, \langle BS, 2\rangle\rangle$

$\vec{\alpha}''|_{BS}$
$\langle\langle SH, 2\rangle, \langle BS, 1\rangle\rangle$
$\langle\langle BH, 4\rangle, \langle BS, 2\rangle\rangle$
$\langle BS, 3\rangle\rangle$

(d)

Dummified programs:

Program for ST_P
$1 : st =$ 0 OR 1 ;
$2 : no - op$

Program for BT_P
$1 : bt = 1;$
$2 : \texttt{send}(BH, bt);$

Program for SH_P
$1 : sh =$ 0 OR 1 ;
$2 : \texttt{send}(BS, sh);$
$3 : \texttt{send}(BH, sh);$

Program for BH_P
$1 : bt = \texttt{recv}(BT);$
$2 : sh = \texttt{recv}(SH);$
$3 : bh = bt \wedge \neg sh;$
$4 : \texttt{send}(BS, bh)$

Program for BS_P
$1 : sh = \texttt{recv}(SH);$
$2 : bh = \texttt{recv}(BH);$
$3 : bs = sh \vee bh;$

Violation: All traces where $bs = 1$.

Figure 4.9: Example 10: Late preemption as encoded in the structural equations with BT. $\vec{\alpha}''$ is found as a cause because a preemption-based example is modeled using a symmetric operator.

Part II

Application to Security Protocols

Chapter 5

Defining Program Actions as Actual Causes

Chapter goal. In this chapter we adapt the interaction-aware theory developed in Part 1 to security protocols. The chapter is organized as follows. Section 5.1 describes a representative example which we use throughout this chapter to explain important concepts. We motivate the clauses in the definition with the help of security-based example. We briefly revisit our model in Section 5.2.1, define auxiliary notions (like log consistency) in Section 5.2.2, and present the formal definition of programs as actual causes for security violations in Section 5.2.4. In Section 5.3, we discuss the relationship of the definition presented in this part with the definition proposed in Chapter 3. We apply the causal analysis to the running example in Section 5.4. The full proofs can be found in Appendix B.

Note: Since we do not discuss the structural equations for the case study, therefore, in this chapter we indicate the processes directly by their identifiers, and drop the subscript P.

Application to security protocols. We use a restricted syntax, and omit the internal choice operator and the asymmetric disjunction operator. We formalize the ideas in the previous part of the thesis, using a logic suitable for modeling security protocols. The central contribution of this chapter is a formal definition of *program actions as actual causes*. Specifically, we define what it means for a set of program actions to be an actual cause of a violation. The definition considers a set of interacting programs whose concurrent execution, as recorded in a log, violates a trace property. It identifies a subset of actions (program steps) of these programs as an actual cause of the violation. The definition applies in two phases. The first phase identifies what we call *Lamport causes*. A Lamport cause is a minimal prefix of the log of a violating trace that can

account for the violation. In the second phase, we refine the actions on this log by removing the actions which are merely *progress enablers* and obtain *actual action causes*. The former contribute only indirectly to the cause by enabling the actual action causes to make progress; the exact values returned by progress enabling actions are irrelevant.

We demonstrate the value of this formalism in two ways. First, we prove that violations of a precisely defined class of safety properties always have an actual cause. Thus, our definition applies to relevant security properties. Second, we provide a cause analysis of a representative protocol designed to address weaknesses in the current public key certification infrastructure. Moreover, our example illustrates that our definition cleanly handles the separation between joint and independent causes –a recognized challenge for actual cause definitions [14, 27, 30].

5.1 Motivating example

In this section we describe an example protocol designed to increase accountability in the current public key infrastructure. We use the protocol later to illustrate key concepts in defining causality.

Security protocol. Consider an authentication protocol in which a user (User1) authenticates to a server (Server1) using a pre-shared password over an adversarial network. User1 sends its user-id to Server1 and obtains a public key signed by Server1. However, User1 would need inputs from additional sources when Server1 sends its public key for the first time in a protocol session to verify that the key is indeed bound to Server1's identity. In particular, User1 can verify the key by contacting multiple notaries in the spirit of *Perspectives* [77]. For simplicity, we assume User1 verifies Server1's public key with three authorized notaries—Notary1, Notary2, Notary3—and accepts the key if and only if the majority of the notaries say that the key is legitimate. To illustrate some of our ideas, we also consider a parallel protocol where two parties (User2 and User3) communicate with each other.

We assume that the prescribed programs for Server1, User1, Notary1, Notary2, Notary3, User2 and User3 impose the following requirements on their behavior: (i) Server1 stores User1's password in a hashed form in a secure private memory location. (ii) User1 requests access to the account by sending an encryption of the password (along with its identity and a timestamp) to Server1 after verifying Server1's public key with a majority of the notaries. (iii) The notaries retrieve the key from their databases and attest the key correctly. (iv) Server1 decrypts and computes the hashed value of the password. (v) Server1 matches the computed hash value with the previously stored value in the memory location when the account was first created; if the

two hash values match, then Server1 grants access to the account to User1. (vi) In parallel, User2 generates and sends a nonce to User3. (vii) User3 generates a nonce and responds to User2.

Security property. The prescribed programs in our example aim to achieve the property that only the user who created the account and password (in this case, User1) gains access to the account.

Compromised Notaries Attack. We describe an attack scenario and use it to illustrate nuances in formalizing program actions as actual causes. User1 executes its prescribed program. User1 sends an access request to Server1. An Adversary intercepts the message and sends a public key to User1 pretending to be Server1. User1 checks with Notary1, Notary2 and Notary3 who falsely verify Adversary's public key to be Server1's key. Consequently, User1 sends the password to Adversary. Adversary then initiates a protocol with Server1 and gains access to User1's account. In parallel, User2 sends a request to Server1 and receives a response from Server1. Following this interaction, User2 forwards the message to User3. We assume that the actions of the parties are recorded on a *log*, say l. Note that this log contains a violation of the security property described above since Adversary gains access to an account owned by User1.

First, our definition finds *program actions as causes* of violations. At a high-level, as mentioned in the introduction, our definition applies in two phases. The first phase (Section 5.2, Definition 15) identifies a minimal prefix (Phase 1, *minimality*) of the log that can account for the violation i.e. we consider all scenarios where the sequence of actions execute in the same order as on the log, and test whether it suffices to recreate the violation in the absence of all other actions (Phase 1, *sufficiency*). In our example, this first phase will output a minimal prefix of log l above. In this case, the minimal prefix will not contain interactions between User2 and User3 after Server1 has granted access to the Adversary (the remaining prefix will still contain a violation).

Second, a nuance in defining the notion of *sufficiency* (Phase 1, Definition 15) is to constrain the interactions which are a part of the actual cause set in a manner that is consistent with the interaction recorded on the log. This constraint on interactions is quite subtle to define and depends on how strong a coupling we find appropriate between the log and possible counterfactual traces in sufficiency: if the constraint is too weak then the violation does not reappear in all sequences, thus missing certain causes; if it is too strong it leads to counter-intuitive cause determinations. For example, a weak notion of consistency is to require that each program locally execute the same prefix in sufficiency as it does on the log i.e. consistency w.r.t. program actions for individual programs. This notion does not work because for some violations to occur

the *order of interactions* on the log among programs is important. A notion that is too strong is to require matching of the total order of execution of all actions across all programs. We present a formal notion of *consistency* by comparing log projections (Section 5.2.2) that balance these competing concerns.

Third, note that while Phase 1 captures a minimal prefix of the log sufficient for the violation, it might be possible to remove actions from this prefix which are merely required for a program execution to progress. For instance note that while all three notaries' actions are required for User1 to progress (otherwise it would be stuck waiting to receive a message) and the violation to occur, the actual message sent by one of the notaries is irrelevant since it does not affect the majority decision in this example. Thus, separating out actions which are *progress enablers* from those which provide information that causes the violation is useful for fine-grained causal determination. This observation motivates the final piece (Phase 2) of our formal definition (Definition 17).

Finally, notice that in this example Adversary, Notary1, Notary2, Notary3, Server1 and User2 deviate from the protocol described above. However, the deviant programs are not sufficient for the violation to occur without the involvement of User1, which is also a part of the causal set. We thus seek a notion of sufficiency in defining a set of programs as a joint actual cause for the violation. Joint causation is also significant in legal contexts [75]. For instance, it is useful for holding liable a group of agents working together when none of them satisfy the cause criteria individually but together their actions are found to be a cause. The ability to distinguish between joint and independent (i.e., different sets of programs that independently caused the violation) causes is an important criterion that we want our definition to satisfy. In particular, Phase 2 of our definition helps identify independent causes. For instance, in our example, we get three different independent causes depending on which notary's action is treated as a progress enabler. Our ultimate goal is to use the notion of actual cause as a building block for accountability — the independent vs. joint cause distinction is significant when making deliberations about accountability and punishment for liable parties. We can use the result of our causal determinations to further remove deviants whose actions are required for the violation to occur but might not be blameworthy (Chapter 6).

Initial approach: Defining programs as actual causes. Considering programs instead of events as actual causes is appropriate in security settings because individual agents can exercise their choice to either execute the prescribed program or deviate from it. In Appendix C, we sketch an initial approach which finds programs, rather than program actions (the latter is more fine grained) as causes of violations. We then explain with the help of an example, why

we need a more fine grained analysis.

Next we turn our attention to defining program actions as actual causes.

5.2 Program actions as actual causes

We present our language model in Section 5.2.1, auxiliary notions in Section 5.2.2, properties of interest to our analysis in Section 5.2.3, and the formal definition of program actions as actual causes in Section 5.2.4.

5.2.1 Model

Note: The syntax presented here is an instantiation of the syntax presented in Chapter 2 with the following differences: we omit the choice operator and add an `assert` construct. We further model thread-local computations using ζ.

We model programs in a simple concurrent language, which we call L. The language contains sequential expressions, e, that execute concurrently in programs and communicate with each other through `send` and `recv` commands as discussed in this section. We also include very primitive condition checking in the form of $\texttt{assert}(v)$.

Our syntax is given using the A-normal form [58] where every term contains only one connective and all operands contain only variables. The syntax consists of values v for variables x, actions α and expressions e. Values v include boolean values, numerical values and all other return values (such as keys or cipher text). Variables, x, denote messages that may be passed through expressions or across programs.

An expression is a sequence of actions, α. An action may do one of the following: execute a primitive function ζ on values $v_1, v_2, \ldots$, or send or receive a message to another program; (written $\texttt{send}(v)$ and $\texttt{recv}()$, respectively).

$$
\begin{array}{lrcl}
\text{Values} & v & ::= & x \mid \texttt{true} \mid \texttt{false} \mid 1 \mid 2 \ldots \\
\text{Actions} & \alpha & ::= & v \mid \texttt{send}(v) \mid \texttt{recv}() \mid \zeta(v) \ldots \\
\text{Expressions} & e & ::= & v \mid (b : x = \alpha); e_1 \mid \texttt{assert}(v); e_1
\end{array}
$$

Following standard models of protocols, `send` and `recv` are untargeted in the operational semantics: A message sent by a thread may be received by any thread. Targeted communication may be layered on this basic semantics using cryptography. For readability in examples, we provide an additional first argument to `send` and `recv` that specifies the *intended* target (the operational semantics ignore this intended target). Action $\texttt{send}(v)$ always returns 0 to its

continuation.

Primitive functions ζ model thread-local computation like arithmetic and cryptographic operations. Primitive functions can also read and update a *thread-local state*, which may model local databases, permission matrices, session information, etc. If the term v in $\texttt{assert}(v)$ evaluates to a non-true value, then its containing thread gets stuck forever, else $\texttt{assert}(v)$ has no effect.

Operational Semantics. The language L's operational semantics define how a collection of *threads* execute concurrently. Each thread T contains a unique thread identifier i (drawn from a universal set of such identifiers), the executing expression e, and a local store. A *configuration* $\mathcal{C} = T_1, \ldots, T_n$ models the threads $T_1, \ldots, T_n$ executing concurrently. Our reduction relation is written $\mathcal{C} \to \mathcal{C}'$ and defined in the standard way by interleaving small steps of individual threads (the reduction relation is parametrized by a semantics of primitive functions ζ). Importantly, each reduction can either be internal to a single thread or a *synchronization* of a `send` in one thread with a `recv` in another thread.

We make the locus of a reduction explicit by annotating the reduction arrow with a *label* r. This is written $\mathcal{C} \xrightarrow{r} \mathcal{C}'$. A label is either the identifier of a thread i paired with a line number b, written $\langle i, b\rangle$ and representing an internal reduction of some $\zeta(t)$ in thread i at line number b, or a tuple $\langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle$, representing a synchronization between a `send` at line number b_s in thread i_s with a `recv` at line number b_r in thread i_r, or ϵ indicating an unobservable reduction (of t or $\texttt{assert}(t)$) in some thread. Labels $\langle i, b\rangle$ are called *local labels*, labels $\langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle$ are called *synchronization labels* and labels ϵ are called *silent labels*.

An *initial configuration* can be described by a triple $\langle I, \mathcal{A}, \Sigma\rangle$, where I is a finite set of thread identifiers, $\mathcal{A} : I \to$ Expressions and $\Sigma : I \to$ Stores. This defines an initial configuration of $|I|$ threads with identifiers in I, where thread i contains the expression $\mathcal{A}(i)$ and the store $\Sigma(i)$. In the sequel, we identify the triple $\langle I, \mathcal{A}, \Sigma\rangle$ with the configuration defined by it. We also use a configuration's identifiers to refer to its threads.

Definition 4 (Run). *Given an initial configuration $\mathcal{C}_0 = \langle I, \mathcal{A}, \Sigma\rangle$, a run is a finite sequence of labeled reductions $\mathcal{C}_0 \xrightarrow{r_1} \mathcal{C}_1 \ldots \xrightarrow{r_n} \mathcal{C}_n$.*

A pre-trace is obtained by projecting only the stores from each configuration in a run.

Definition 5 (Pre-trace). *Let $\mathcal{C}_0 \xrightarrow{r_1} \mathcal{C}_1 \ldots \xrightarrow{r_n} \mathcal{C}_n$ be a run and let Σ_i be the store in configuration $\mathcal{C}_i$. Then, the pre-trace of the run is the sequence $(\_, \Sigma_0), (r_1, \Sigma_1), \ldots, (r_n, \Sigma_n)$.*

If $r_i = \epsilon$, then the ith step is an unobservable reduction in some thread and, additionally, $\Sigma_{i-1} = \Sigma_i$. A trace is a pre-trace from which such ϵ steps have been dropped.

Definition 6 (Trace). *The trace of the pre-trace $(\_, \Sigma_0), (r_1, \Sigma_1), \ldots, (r_n, \Sigma_n)$ is the subsequence*

obtained by dropping all tuples of the form (ϵ, Σ_i). *Traces are denoted with the letter* t.

5.2.2 Logs and their projections

To define actual causation, we find it convenient to introduce the notion of a log and the log of a trace, which is just the sequence of non-silent labels on the trace. A log is a sequence of labels other than ϵ. The letter l denotes logs.

Definition 7 (Log). *Given a trace* $t = (\_, \Sigma_0), (r_1, \Sigma_1), \ldots, (r_n, \Sigma_n)$, *the log of the trace,* $log(t)$, *is the sequence of* $r_1, \ldots, r_m$. *(The trace* t *does not contain a label* r_i *that equals* ϵ, *so neither does* $log(t)$.*)*

We need a few more straightforward definitions on logs in order to define actual causation.

Definition 8 (Projection of a log). *Given a log* l *and a thread identifier* i, *the projection of* l *to* i, *written* $l|_i$ *is the subsequence of all labels in* l *that mention* i. *Formally,*

$$
\begin{aligned}
\bullet|_i &= \bullet \\
(\langle i, b\rangle :: l)|_i &= \langle i, b\rangle :: (l|_i) \\
(\langle j, b\rangle :: l)|_i &= l|_i \quad \textit{if } i \neq j \\
(\langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle :: l)|_i &= \langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle :: (l|_i) \\
&\quad \textit{if } i_s = i \textit{ or } i_r = i \\
(\langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle :: l)|_i &= l|_i \\
&\quad \textit{if } i_s \neq i \textit{ and } i_r \neq i
\end{aligned}
$$

Definition 9 (Projected prefix). *We call a log* l' *a* projected prefix *of the log* l, *written* $l' \leq_p l$, *if for every thread identifier* i, *the sequence* $l'|_i$ *is a prefix of the sequence* $l|_i$.

The definition of projected prefix allows the relative order of events in two different non-communicating threads to differ in l and l' but Lamport's happens-before order of actions [59] in l' must be preserved in l. Similar to projected prefix, we define projected sublog.

Definition 10 (Projected sublog). *We call a log* l' *a* projected sublog *of the log* l, *written* $l' \sqsubseteq_p l$, *if for every thread identifier* i, *the sequence* $l'|_i$ *is a subsequence of the sequence* $l|_i$ *(i.e., dropping some labels from* $l|_i$ *results in* $l'|_i$*).*

5.2.3 Properties of interest

A *property* is a set of (good) traces and violations are traces in the complement of the set. Our goal is to define the cause of a violation of a property. We are specifically interested in ascribing causes to violations of safety properties [78] because safety properties encompass

many relevant security requirements. We recapitulate the definition of a safety property below. Briefly, a property is safety if it is fully characterized by a set of finite violating prefixes of traces. Let U denote the universe of all possible traces.

Definition 11 (Safety property [79]). *A property P (a set of traces) is a safety property, written* $\mathsf{Safety}(P)$*, if* $\forall t \notin P.\ \exists t' \in U.\ (t'$ *is a prefix of* $t) \wedge (\forall t'' \in U.\ (t' \cdot t'' \notin P))$.

As we explain soon, our causal analysis ascribes thread actions (or threads) as causes. One important requirement for such analysis is that the property be closed under reordering of actions in different threads if those actions are not related by Lamport's happens-before relation [59]. For properties that are not closed in this sense, the *global order* between actions in a race condition may be a cause of a violation. Whereas causal analysis of race conditions may be practically relevant in some situation, we limit attention only to properties that are closed in the sense described here. We call such properties reordering-closed or RC.

Definition 12 (Reordering-equivalence). *Two traces t_1, t_2 starting from the same initial configuration are called reordering-equivalent, written $t_1 \sim t_2$ if for each thread identifier i, $log(t_1)|_i = log(t_2)|_i$. Note that $\sim$ is an equivalence relation on traces from a given initial configuration. Let $[t]_\sim$ denote the equivalence class of t.*

Definition 13 (Reordering-closed property). *A property P is called reordering-closed, written* $\mathsf{RC}(P)$*, if $t \in P$ implies $[t]_\sim \subseteq P$. Note that* $\mathsf{RC}(P)$ *iff* $\mathsf{RC}(\neg P)$.

5.2.4 Formal definition: Program actions as actual causes

In the sequel, let φ_V denote the *complement* of a reordering-closed safety property of interest. (The subscript V stands for "violations".) Consider a trace t starting from the initial configuration $\mathcal{C}_0 = \langle I, \mathcal{A}, \Sigma \rangle$. If $t \in \varphi_V$, then t violates the property $\neg\varphi_V$.

Definition 14 (Violation). *A violation of the property $\neg\varphi_V$ is a trace $t \in \varphi_V$.*

Our definition of actual causation identifies a subset of actions in $\{\mathcal{A}(i) \mid i \in I\}$ as the cause of a violation $t \in \varphi_V$. The definition applies in two phases. The first phase identifies what we call *Lamport causes*. A Lamport cause is a minimal projected prefix of the log of a violating trace that can account for the violation. In the second phase, we refine the log by removing actions that are merely *progress enablers*; the remaining actions on the log are the *actual action causes*. The former contribute only indirectly to the cause by enabling the actual action causes to make progress; the exact values returned by progress enabling actions are irrelevant.

The following definition, called Phase 1, determines Lamport causes. It works as follows. We first identify a projected prefix l of the log of a violating trace t as a potential candidate for a Lamport cause. We then check two conditions on l. The *sufficiency* condition tests that the

threads of the configuration, when executed at least up to the identified prefix, preserving all synchronizations in the prefix, suffice to recreate the violation. The *minimality* condition tests that the identified Lamport cause contains no redundant actions.

Definition 15 (Phase 1: Lamport Cause of Violation)**.** *Let $t \in \varphi_V$ be a trace starting from $\mathcal{C}_0 = \langle I, \mathcal{A}, \Sigma \rangle$ and l be a projected prefix of $log(t)$, i.e., $l \leq_p log(t)$. We say that l is the Lamport cause of the violation t of φ_V if the following hold:*

1. ***(Sufficiency)** Let T be the set of traces starting from $\mathcal{C}_0$ whose logs contain l as a projected prefix, i.e., $T = \{t' \mid t' \text{ is a trace starting from } \mathcal{C}_0 \text{ and } l \leq_p log(t')\}$. Then, every trace in T has the violation φ_V, i.e., $T \subseteq \varphi_V$. (Because $t \in T$, T is non-empty.)*
2. ***(Minimality)** No proper prefix of l satisfies condition 1.*

At the end of Phase 1, we obtain one or more minimal prefixes l which contain program actions that are sufficient for the violation. These prefixes represent independent Lamport causes of the violation. In the Phase 2 definition below, we further identify a sublog a_d of each l, such that the program actions in a_d are actual causes and the actions in $l \backslash a_d$ are progress enabling actions which only contribute towards the *progress* of actions in a_d that cause the violation. In other words, the actions not considered in a_d contain all labels whose actual returned values are irrelevant.

Briefly, here's how our Phase 2 definition works. We first pick a candidate projected sublog a_d of l, where log l is a Lamport cause identified in Phase 1. We consider counterfactual traces obtained from initial configurations in which program actions omitted from a_d are replaced by actions that do not have any effect other than enabling the program to progress (referred to as no-op). If a violation appears in all such counterfactual traces, then this sublog a_d is a good candidate. Of all such good candidates, we choose those that are minimal.

The key technical difficulty in writing this definition is replacing program actions omitted from a_d with no-ops. We cannot simply erase any such action because the action is expected to return a term which is bound to a variable used in the action's continuation. Hence, our approach is to substitute the variables binding the returns of no-op'ed actions with arbitrary (side-effect free) terms t. Formally, we assume a function $f : I \times \text{LineNumbers} \to \text{Terms}$ that for line number b in thread i suggests a suitable term $f(i, b)$ that must be returned if the action from line b in thread i is replaced with a no-op. In our cause definition we universally quantify over f, thus obtaining the effect of a no-op. For technical convenience, we define a syntactic transform called dummify() that takes an initial configuration, the chosen sublog a_d and the function f, and produces a new initial configuration obtained by erasing actions not in a_d by terms obtained through f.

Definition 16 (Dummifying transformation)**.** *Let $\langle I, \mathcal{A}, \Sigma \rangle$ be a configuration and let a_d be a*

log. Let $f : I \times \mathit{LineNumbers} \to \mathit{Terms}$. *The dummifying transform* $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ *is the initial configuration* $\langle I, \mathcal{D}, \Sigma \rangle$, *where for all* $i \in I$, $\mathcal{D}(i)$ *is* $\mathcal{A}(i)$ *modified as follows:*

- *If* $(b : x = \texttt{send}(t)); e$ *appears in* $\mathcal{A}(i)$ *but* $\langle i, b \rangle$ *does not appear in* a_d, *then replace* $(b : x = \texttt{send}(t)); e$ *with* $e[0/x]$ *in* $\mathcal{A}(i)$.
- *If* $(b : x = \alpha); e$ *appears in* $\mathcal{A}(i)$ *but* $\langle i, b \rangle$ *does not appear in* a_d *and* $\alpha \neq \texttt{send}(\_)$, *then replace* $(b : x = \alpha); e$ *with* $e[f(i, b)/x]$ *in* $\mathcal{A}(i)$.

We now present our main definition of actual causes.

Definition 17 (Phase 2: Actual Cause of Violation)**.** *Let* $t \in \varphi_V$ *be a trace from the initial configuration* $\langle I, \mathcal{A}, \Sigma \rangle$ *and let the log* $l \leq_p log(t)$ *be a Lamport cause of the violation determined by Definition 15. Let* a_d *be a projected sublog of* l, *i.e., let* $a_d \sqsubseteq_p l$. *We say that* a_d *is the actual cause of violation* t *of* φ_V *if the following hold:*

1. ***(Sufficiency')*** *Pick any* f. *Let* $\mathcal{C}_0' = \mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ *and let* T *be the set of traces starting from* $\mathcal{C}_0'$ *whose logs contain* a_d *as a projected sublog, i.e.,* $T = \{t' \mid t' \text{ is a trace starting from } \mathcal{C}_0' \text{ and } a_d \sqsubseteq_p log(t')\}$. *Then, for all* f, T *is non-empty and every trace in* T *has the violation* φ_V, *i.e,* $T \subseteq \varphi_V$.
2. ***(Minimality')*** *No proper sublog of* a_d *satisfies condition 1.*

At the end of Phase 2, we obtain one or more sets of actions a_d. These sets are deemed the independent actual causes of the violation t.

The following theorem states that for all safety properties that are re-ordering closed, the Phase 1 and Phase 2 definitions always identify at least one Lamport and at least one actual cause.

Theorem 1. *Suppose* φ_V *is reordering-closed and the complement of a safety property, i.e.,* $\mathsf{RC}(\varphi_V)$ *and* $\mathsf{safety}(\neg\varphi_V)$. *Then, for every* $t \in \varphi_V$: *(1) Our Phase 1 definition (Definition 15) finds a Lamport cause* l, *and (2) For every such Lamport cause* l, *the Phase 2 definition (Definition 17) finds an actual cause* a_d.

Proof. (1) Pick any $t \in \varphi_V$. We follow the Phase 1 definition. It suffices to prove that there is a log $l \leq_p log(t)$ that satisfies the sufficiency condition. Since $\mathsf{safety}(\neg\varphi_V)$, there is a prefix t_0 of t s.t. for all $t_1 \in U$, $t_0 \cdot t_1 \in \varphi_V$. Choose $l = log(t_0)$. Since t_0 is a prefix of t, $l = log(t_0) \leq_p log(t)$. To prove sufficiency, pick any trace t' s.t. $l \leq_p log(t')$. It suffices to prove $t' \in \varphi_V$. Since $l \leq_p log(t')$, for each i, $log(t')|_i = l|_i \cdot l_i'$ for some l_i'. Let t'' be the (unique) subsequence of t' containing all labels from the logs $\{l_i'\}$. Consider the trace $s = t_0 \cdot t''$. First, s extends t_0, so $s \in \varphi_V$. Second, $s \sim t'$ because $log(s)|_i = l|_i \cdot l_i' = log(t_0)|_i \cdot log(t'')|_i = log(t_0 \cdot t'')|_i = log(t')|_i$. Since $\mathsf{RC}(\varphi_V)$, $t' \in \varphi_V$.

(2) Pick any $t \in \varphi_V$ and let l be a Lamport cause of t as determined by the Phase 1 definition.

Following the Phase 2 definition, we only need to prove that there is at least one $a_d \sqsubseteq_p l$ that satisfies the sufficiency' condition. We choose $a_d = l$. To show sufficiency', pick any f. Because $a_d = l$, a_d specifies an initial prefix of every $\mathcal{A}(i)$ and the transform $\mathsf{dummify}()$ has no effect on this prefix. First, we need to show that at least one trace t' starting from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ satisfies $a_d \sqsubseteq_p log(t')$. For this, we can pick $t' = t$. Second, we need to prove that any trace t' starting from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ s.t. $a_d \sqsubseteq_p log(t')$ satisfies $t' \in \varphi_V$. Pick such a t'. Let t_0 be the prefix of t corresponding to l. Then, $log(t_0)|_i = l|_i$ for each i. It follows immediately that for each i, $t'|_i = t_0|_i \cdot t''_i$ for some t''_i. Let t'' be the unique subsequence of t' containing all labels from traces $\{t''_i\}$. Let $s = t_0 \cdot t''$. First, because for each i, $l|_i = log(t_0)|_i$, $l \leq_p log(t_0)$ trivially. Because l is a Lamport cause, it satisfies the sufficiency condition of Phase 1, so $t_0 \in \varphi_V$. Since $\mathsf{safety}(\neg\varphi_V)$, and s extends t_0, $s \in \varphi_V$. Second, $s \sim t'$ because $log(s)|_i = log(t_0)|_i \cdot log(t'')|_i = log(t')|_i$ and both s and t' are traces starting from the initial configuration $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$. Hence, by $\mathsf{RC}(\varphi_V)$, $t' \in \varphi_V$. ■

Our Phase 2 definition identifies a set of program actions as causes of a violation. However, in some applications it may be necessary to ascribe thread identifiers (or programs) as causes. This can be straightforwardly handled by lifting the Phase 2 definition: A thread i (or $\mathcal{A}(i)$) is a cause if one of its actions appears in a_d.

Definition 18 (Program Cause of Violation)**.** *Let a_d be an actual cause of violation φ_V on trace t starting from $\langle I, \mathcal{A}, \Sigma\rangle$. We say that the set $X \subseteq I$ of thread identifiers is a cause of the violation if $X = \{i \mid i$ appears in $a_d\}$.*

Remarks. We make a few technical observations about our definitions of cause. First, because Lamport causes (Definition 15) are projected *prefixes*, they contain all actions that occur before any action that actually contributes to the violation. Many of the actions in the Lamport cause may not contribute to the violation intuitively. Our actual cause definition filters out such "spurious" actions. As an example, suppose that a safety property requires that the value 1 never be sent on the network. The (only) trace of the program $x = 1; y = 2; z = 3; \mathtt{send}(x)$ violates this property. The Lamport cause of this violation contains all four actions of the program, but it is intuitively clear that the two actions $y = 2$ and $z = 3$ do not contribute to the violation. Indeed, the actual cause of the violation determined by Definition 17 does not contain these two actions; it contains only $x = 1$ and $\mathtt{send}(x)$, both of which obviously contribute to the violation.

Second, our definition of dummification is based on a program transformation that needs line numbers. One possibly unwanted consequence is that our traces have line numbers and, hence, we could, in principle, specify safety properties that are sensitive to line numbers. How-

ever, our definitions of cause are closed under bijective renaming of line numbers, so if a safety property is insensitive to line numbers, the actual causes can be quotiented under bijective renamings of line numbers.

Third, our definition of actual cause (Definition 17) separates actions whose return values are relevant to the violation from those whose return values are irrelevant for the violation. This is closely related to noninterference-like security definitions for information flow control, in particular, those that separate input presence from input content [80]. Lamport causes (Definition 15) have a trivial connection to information flow: If an action does not occur in any Lamport cause of a violation, then there cannot be an information flow from that action to the occurrence of the violation.

Fourth, the definition of actual cause (Definition 17) given in this chapter is closely related to the definition presented in Chapter 3 (Definition 3), which we discuss next.

5.3 Relationship with Definition in Part 1

The definitions given in Part 1 and Part 2 of the dissertation are closely related. As stated in the introduction, chronologically the definition in Part 2 of the dissertation preceded the definition in Part 1 of the dissertation and the latter is a generalization for the former. If we omit the choice operator and asymmetric disjunction from the syntax in Part 1 of the dissertation, then the dummifying transform in both parts of the dissertation (Definitions 16 and 2) coincide. Additionally, Phase 2 of the definition in Part 2 of the dissertation (Definition 17) coincides with the actual cause definition in Part 1 of the dissertation (Definition 3). We can show that Lamport cause (Definition 15) is not required for definitional purposes by proving the following theorem.

Theorem 2. *Given log l where the corresponding trace $t \in \varphi_V$, if applying Definitions 15 and 17 outputs a_d as a cause of violation t of φ_V, then Definition 3 also finds a_d as a cause of violation t of φ_V.*

Specifically, we show that any minimal cause a_d output by the definition of lamport cause along with Definition 17, will also be output by Definition 17 (Phase 2) alone since the latter coincides with the definition in Part 1 of the dissertation.

Let a_d be a sufficient cause of violation t of φ_V according to Phase 1 and Phase 2 (Definitions 15 and 17). Then it will also be a sufficient cause according to Definition 3, since the two sufficiency clauses coincide. Let a_d also be a minimal cause according to Definitions 15 and 17, but not according to Definition 3. This implies that Definition 3 must have found a subsequence of a_d as a cause of violation t of φ_V since Definition 3 outputs the minimal sublog of l that is

sufficient. However, this violates the minimality condition for Definition 17, since a_d must be the minimal of all sublogs, of a minimal prefix of log l (which is also a sublog of l), that satisfy the sufficiency clause in Phase 2. Therefore Definition 3 will also output a_d as a cause of violation t of φ_V.

However, the converse does not hold, i.e. if Definition 3 outputs a_d as a cause given log l, it is not always the case that Phase 1 and 2 (Definitions 15 and 17) will also find the same cause. This holds since Phase 1 definition (lamport cause) will find a minimal prefix sufficient for a violation and Definition 17 outputs a corresponding minimal sufficient sublog. Therefore if there exist independent causes of violation t of φ_V, Definition 17 will only find those causal sequences for which there exists a minimal sufficient projected prefix l' of the log l. In contrast, Definition 3 will find all independent causes for a violation.

5.4 Application: Causes of authentication failures

In this section, we model an instance of our running example based on passwords (Section 5.1) in order to demonstrate our actual cause definition. As explained in Section 5.1, we consider a protocol session where Server1, User1, User2, User3 and multiple notaries interact over an adversarial network to establish access over a password-protected account. We describe a formal model of the protocol in our language, examine the attack scenario from Section 5.1 and provide a cause analysis using the definitions from Section 5.2.

5.4.1 Protocol description

We consider our example protocol with eight threads named {Server1, User1, Adversary, Notary1, Notary2, Notary3, User2, User3}. In this section, we briefly describe the protocol and the programs specified by the protocol for each of these threads. For this purpose, we assume that we are provided a function $\mathcal{N} : I \rightarrow$ Expressions such that $\mathcal{N}(i)$ is the program that *ideally should have been* executing in the thread i. For each i, we call $\mathcal{N}(i)$ the *norm* for thread i. The violation is caused because some of the executing programs are different from the norms. These actual programs, called $\mathcal{A}$ as in Section 5.2, are shown later. The norms are shown here to help the reader understand what the ideal protocol is and also to facilitate some of the development in Chapter 6. The appendix describes an expansion of this example with more than the eight threads considered here to illustrate our definitions better. The proof included in the appendix deals with timestamps and signatures.

The norms in Figure 5.1 and the actuals in Figure 5.2 assume that User1's account (called $acct$ in Server1's program) has already been created and that User1's password, pwd is associated

with User1's user id, uid. This association (in hashed form) is stored in Server1's local state at pointer mem. The norm for Server1 is to wait for a request from an entity, respond with its (Server1's) public key, wait for a username-password pair encrypted with that public key and grant access to the requester if the password matches the previously stored value in Server1's memory at mem. To grant access, Server1 adds an entry into a private access matrix, called P. (A separate server thread, not shown here, allows User1 to access its account if this entry exists in P.)

The norm for User1 is to send an access request to Server1, wait for the server's public key, verify that key with three notaries and then send its password pwd to Server1, encrypted under Server1's public key. On receiving Server1's public key, User1 initiates a protocol with the three notaries and accepts or rejects the key based on the response of a majority of the notaries. For simplicity, we omit a detailed description of this protocol between User1 and the notaries that authenticates the notaries and ensures freshness of their responses. These details are included in our appendix. In parallel, the norm for User2 is to generate and send a nonce to User3. The norm for User3 is to receive a message from User2, generate a nonce and send it to User2.

Each notary has a private database of *(public_key, principal)* tuples. The notaries' norms assume that this database has already been created correctly. When User1 sends a request with a public key, the notary responds with the principal's identifier after retrieving the tuple corresponding to the key from its database.

Notation. The programs in this example use several primitive functions ζ. $\texttt{Enc}(k, m)$ and $\texttt{Dec}(k', m)$ denote encryption and decryption of message m with key k and k' respectively. $\texttt{Hash}(m)$ generates the hash of term m. $\texttt{Sig}(k, m)$ denotes message m signed with the key k, paired with m in the clear. pub_key_i and pvt_key_i denote the public and private keys of thread i, respectively. For readability, we include the intended recipient i and expected sender j of a message as the first argument of $\texttt{send}(i, m)$ and $\texttt{recv}(j)$ expressions. As explained earlier, i and j are ignored during execution and a network adversary, if present, may capture or inject any messages.

Security property. The security property of interest to us is that if at time u, a thread k is given access to account a, then k owns a. Specifically, in this example, we are interested in case $a = acct$ and $k =$ User1. This can be formalized by the following logical formula, $\neg\varphi_V$:

$$\forall u, k.\ (acct, k) \in P(u) \supset (k = \mathsf{User1}) \tag{5.1}$$

Here, $P(u)$ is the state of the access control matrix P for Server1 at time u.

5.4.2 Attack

As an illustration, we model the "Compromised Notaries" violation of Section 5.1. The programs executed by all threads are given in Figure 5.2. User1 sends an access request to Server1 which is intercepted by Adversary who sends its own key to User1 (pretending to be Server1). User1 checks with the three notaries who falsely verify Adversary's public key to be Server1's key. Consequently, User1 sends the password to Adversary. Adversary then initiates a protocol with Server1 and gains access to the User1's account. Note that the actual programs of the three notaries attest that the public key given to them belongs to Server1. In parallel, User2 sends a request to Server1 and receives a response from Server1. Following this interaction, User2 interacts with User3, as in their norms.

Figure 5.3 shows the expressions executed by each thread on the property-violating trace. For instance, the label $\langle\langle \mathsf{User1}, 1\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle$ indicates that both User1 and Adversary executed the expressions with the line number 1 in their actual programs, which resulted in a synchronous communication between them, while the label $\langle \mathsf{Adversary}, 4\rangle$ indicates the local execution of the expression at line 4 of Adversary's program. The initial configuration has the programs: $\{\mathcal{A}(\mathsf{User1}), \mathcal{A}(\mathsf{Server1}), \mathcal{A}(\mathsf{Adversary}), \mathcal{A}(\mathsf{Notary1}),$
$\mathcal{A}(\mathsf{Notary2}), \mathcal{A}(\mathsf{Notary3}), \mathcal{A}(\mathsf{User2}), \mathcal{A}(\mathsf{User3})\}$. For this attack scenario, the concrete trace t we consider is such that $\log(t)$ is any *arbitrary interleaving* of the actions for
$X = \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{User2}, \mathsf{User3}, \mathsf{Server1}, \mathsf{Notary1},$
$\mathsf{Notary2}, \mathsf{Notary3}\}$ shown in Figure 5.3(a). Any such interleaved log is denoted $\log(t)$ in the sequel. At the end of this log, $(acct, \mathsf{Adversary})$ occurs in the access control matrix P, but Adversary does not own $acct$. Hence, this log corresponds to a violation of our security property.

Note that if any two of the three notaries had attested the Adversary's key to belong to Server1, the violation would still have happened. Consequently, we may expect three independent program causes in this example: {Adversary, User1, Server1, Notary1, Notary2} with the action causes a_d as shown in Figure 5.3(c), {Adversary, User1, Server1, Notary1, Notary3} with the actions a'_d, and {Adversary, User1, Server1, Notary2, Notary3} with the actions a''_d where a'_d and a''_d can be obtained from a_d (Figure 5.3) by considering actions for {Notary1, Notary3} and {Notary2, Notary3} respectively, instead of actions for {Notary1, Notary2}. Our treatment of independent causes follows the tradition in the causality literature. The following theorem states that our definitions determine exactly these three independent causes – one notary is dropped from each of these sets, but no notary is discharged from all the sets. This determination reflects the intuition that only two dishonest notaries are sufficient to cause the violation. Additionally, while it is true that all parties who follow the protocol should not be *blamed* for a violation, an

honest party may be an *actual cause* of the violation (in both the common and the philosophical sense of the word), as demonstrated in this case study. This two-tiered view of accountability of an action by separately asserting cause and blame can also be found in prior work in law and philosophy [8, 36]. Determining actual cause is nontrivial and is the focus of this work.

Theorem 3. *Let* $I = \{\mathsf{User1}, \mathsf{Server1}, \mathsf{Adversary}, \mathsf{Notary1}, \mathsf{Notary2}, \mathsf{Notary3}, \mathsf{User2}, \mathsf{User3}\}$ *and* Σ *and* $\mathcal{A}$ *be as described above. Let* t *be a trace from* $\langle I, \mathcal{A}, \Sigma\rangle$ *such that* $log(t)|_i$ *for each* $i \in I$ *matches the corresponding log projection from Figure 5.3(a). Then, Definition 18 determines three possible values for the program cause* X *of violation* $t \in \varphi_V$*:* {Adversary, User*1,* Server*1,* Notary*1,* Notary*2}, {*Adversary, User*1,* Server*1,* Notary*1,* Notary*3}, and {*Adversary, User*1,* Server*1,* Notary*2,* Notary*3}* *where the corresponding actual causes are* a_d, a'_d *and* a''_d*, respectively.*

It is instructive to understand the proof of this theorem, as it illustrates our definitions of causation. We verify that our Phase 1, Phase 2 definitions (Definitions 15, 17, 18) yield exactly the three values for X mentioned in the theorem.

Lamport cause (Phase 1). We show that any l whose projections match those shown in Figure 5.3(b) satisfies sufficiency and minimality. From Figure 5.3(b), such an l has no actions for User3 and only those actions of User2 that are involved in synchronization with Server1. For all other threads, the log contains every action from t. The intuitive explanation for this l is straightforward: Since l must be a (projected) *prefix* of the trace, and the violation only happens because of `insert` in the last statement of Server1's program, every action of every program before that statement in Lamport's happens-before relation must be in l. This is exactly the l described in Figure 5.3(b).

Formally, following the statement of sufficiency, let T be the set of traces starting from $\mathcal{C}_0 = \langle I, \mathcal{A}, \Sigma\rangle$ (Figure 5.2) whose logs contain l as a projected prefix. Pick any $t' \in T$. We need to show $t' \in \varphi_V$. However, note that any t' containing all actions in l must also add $(acct, \mathsf{Adversary})$ to P, but $\mathsf{Adversary} \neq \mathsf{User1}$. Hence, $t' \in \varphi_V$. Further, l is minimal as described in the previous paragraph.

Actual cause (Phase 2). Phase 2 (Definitions 17, 18) determines three independent program causes for X: {Adversary, User1, Server1, Notary1, Notary2}, {Adversary, User1, Server1, Notary1, Notary3}, and {Adversary, User1, Server1, Notary2, Notary3} with the actual action causes given by a_d, a'_d and a''_d, respectively in Figure 5.3. These are symmetric, so we only explain why a_d satisfies Definition 17. (For this a_d, Definition 18 immediately forces $X = \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary2}\}$.) We show that (a) a_d satisfies sufficiency', and

(b) No proper sublog of a_d satisfies sufficiency' (minimality'). Note that a_d is obtained from l by dropping Notary3, User2 and User3, and all their interactions with other threads.

We start with (a). Let a_d be such that $a_d|_i$ matches Figure 5.3(c) for every i. Fix any dummifying function f. We must show that any trace originating from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, whose log contains a_d as a projected sublog, is in φ_V. Additionally we must show that there is such a trace. There are two potential issues in mimicking the execution in a_d starting from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ — first, with the interaction between User1 and Notary3 and, second, with the interaction between Server1 and User2. For the first interaction, on line 5, $\mathcal{A}(\mathsf{User1})$ (Figure 5.2) synchronizes with Notary3 according to l, but the synchronization label does not exist in a_d. However, in $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, the `recv()` on line 8 in $\mathcal{A}(\mathsf{User1})$ is replaced with a dummy value, so the execution from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ progresses. Subsequently, the majority check (assertion [B]) succeeds as in l, because two of the three notaries (Notary1 and Notary2) still attest the Adversary's key. A similar observation can be made about the interaction between Server1 and User2.

Next we prove that every trace starting from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, whose log contains a_d (Figure 5.3) as a projected sublog, is in φ_V. Fix a trace t' with log l'. Assume l' contains a_d. We show $t' \in \varphi_V$ as follows:

1. Since the synchronization labels in l' are a superset of those in a_d, Server1 must execute line 8 of its program $\mathcal{A}(\mathsf{Server1})$ in t'. After this line, the access control matrix P contains $(acct, J)$ for some J.
2. When $\mathcal{A}(\mathsf{Server1})$ writes (x, J) to P at line 8, then J is the third component of a tuple obtained by decrypting a message received on line 5.
3. Since the synchronization projections on l' are a superset of a_d, and on a_d $\langle \mathsf{Server1}, 5\rangle$ synchronizes with $\langle \mathsf{Adversary}, 8\rangle$, J must be the third component of an encrypted message sent on line 8 of $\mathcal{A}(\mathsf{Adversary})$.
4. The third component of the message sent on line 8 by Adversary is exactly the term "Adversary". (This is easy to see, as the term "Adversary" is hardcoded on line 7.) Hence, $J = \mathsf{Adversary}$.
5. This immediately implies that $t' \in \varphi_V$ since $(acct, \mathsf{Adversary}) \in P$, but $\mathsf{Adversary} \neq \mathsf{User1}$.

Last, we prove (b) — that no proper subsequence of a_d satisfies sufficiency'. Note that a_d (Figure 5.3(c)) contains exactly those actions from l (Figure 5.3) on whose returned values the last statement of Server1's program (Figure 5.2) is data or control dependent. Consequently, all of a_d as shown is necessary to obtain the violation.

(The astute reader may note that in Figure 5.2, there is no dependency between line 1 of Server1's program and the `insert` statement in Server1. Hence, line 1 should not be in a_d. While this is accurate, the program in Figure 5.2 is a slight simplification of the real protocol, which is shown in the appendix. In the real protocol, line 1 returns a received nonce, whose value does influence whether or not execution proceeds to the `insert` statement.)

Norm $\mathcal{N}$(Server1):

$1 : \_ = \texttt{recv}(j)$; //access req from thread j
$2 : \texttt{send}(j, pub\_key\_\text{Server1})$; //send public key to j
$3 : s = \texttt{recv}(j)$; //encrypted uid, pwd, thread id J
$4 : (uid, pwd, J) = \texttt{Dec}(pvt\_key\_\text{Server1}, s)$;
$5 : t = \texttt{hash}(uid, pwd)$;
$\texttt{assert}(mem = t)$ //compare hash with stored value
$6 : \texttt{insert}(P, (acct, J))$;

Norm $\mathcal{N}$(User1):

$1 : \texttt{send}(\text{Server1})$; //access request
$2 : pub\_key = \texttt{recv}(\text{Server1})$; //key from Server1
$3 : \texttt{send}(\text{Notary1}, pub\_key)$;
$4 : \texttt{send}(\text{Notary2}, pub\_key)$;
$5 : \texttt{send}(\text{Notary3}, pub\_key)$;
$6 : \texttt{Sig}(pub\_key, l1) = \texttt{recv}(\text{Notary1})$; //notary1 responds
$7 : \texttt{Sig}(pub\_key, l2) = \texttt{recv}(\text{Notary2})$; //notary2 responds
$8 : \texttt{Sig}(pub\_key, l3) = \texttt{recv}(\text{Notary3})$; //notary3 responds
$\texttt{assert}(\text{At least two of \{l1,l2,l3\} equal Server1})$
$9 : t = \texttt{Enc}(pub\_key, (uid, pwd, \text{User1}))$;
$10 : \texttt{send}(\text{Server1}, t)$; //send t to Server1

Norms $\mathcal{N}$(Notary1), $\mathcal{N}$(Notary2), $\mathcal{N}$(Notary3):

// o denotes Notary1, Notary2 or Notary3
$1 : pub\_key = \texttt{recv}(j)$;
$2 : pr = \texttt{KeyOwner}(pub\_key)$; //lookup key owner
$3 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_o, (pub\_key, pr)))$;

Norm $\mathcal{N}$(User2):

$1 : \texttt{send}(\text{User3})$;
$2 : \_ = \texttt{recv}(\text{User3})$;

Norm $\mathcal{N}$(User3):

$1 : \_ = \texttt{recv}(\text{User2})$;
$2 : \texttt{send}(\text{User3})$;

Figure 5.1: Norms for all threads. Adversary's norm is the trivial empty program.

Actual $\mathcal{A}$(Server1)**:**

$1 : \_ = \texttt{recv}(j)$; //access req from thread j
$2 : \texttt{send}(j, pub\_key\_\text{Server1})$; //send public key to j
$3 : \_ = \texttt{recv}(j)$; //receive nonce from thread User2
$4 : \texttt{send}(j)$; //send signed nonce
$5 : s = \texttt{recv}(j)$; //encrypted uid, pwd, thread id from j
$6 : (uid, pwd, J) = \texttt{Dec}(pvt\_key\_\text{Server1}, s)$;
$7 : t = \texttt{hash}(uid, pwd)$;
$\texttt{assert}(mem = t)$**[A]** //compare hash with stored value
$8 : \texttt{insert}(P, (acct, J))$;

Actual $\mathcal{A}$(User1)**:**

$1 : \texttt{send}(\text{Server1})$; //access request
$2 : pub\_key = \texttt{recv}(\text{Server1})$; //key from Server1
$3 : \texttt{send}(\text{Notary1}, pub\_key)$;
$4 : \texttt{send}(\text{Notary2}, pub\_key)$;
$5 : \texttt{send}(\text{Notary3}, pub\_key)$;
$6 : \texttt{Sig}(pub\_key, l1) = \texttt{recv}(\text{Notary1})$; //notary1 responds
$7 : \texttt{Sig}(pub\_key, l2) = \texttt{recv}(\text{Notary2})$; //notary2 responds
$8 : \texttt{Sig}(pub\_key, l3) = \texttt{recv}(\text{Notary3})$; //notary3 responds
$\texttt{assert}(\text{At least two of \{l1,l2,l3\} equal Server1})$**[B]**
$9 : t = \texttt{Enc}(pub\_key, (uid, pwd, \text{User1}))$;
$10 : \texttt{send}(\text{Server1}, t)$; //send t to Server1

Actual $\mathcal{A}$(Adversary)

$1 : \texttt{recv}(\text{User1})$; //intercept access req from User1
$2 : \texttt{send}(\text{User1}, pub\_key\_A)$; //send key to User
$3 : s = \texttt{recv}(\text{User1})$; //pwd from User1
$4 : (uid, pwd, \text{User1}) = \texttt{Dec}(pvt\_key\_A, s)$; //decrypt pwd
$5 : \texttt{send}(\text{Server1}, uid)$; //access request to Server1
$6 : pub\_key = \texttt{recv}(\text{Server1})$; //Receive Server1's public key
$7 : t = \texttt{Enc}(pub\_key, (uid, pwd, \text{Adversary}))$; //encrypt pwd
$8 : \texttt{send}(\text{Server1}, t)$; //pwd to Server1

Actuals $\mathcal{A}$(Notary1), $\mathcal{A}$(Notary2), $\mathcal{N}$(Notary3)**:**

// o denotes Notary1, Notary2 or Notary3
$1 : pub\_key = \texttt{recv}(j)$;
$2 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_o, (pub\_key, \text{Server1})))$;

Actual $\mathcal{A}$(User2)**:**

$1 : \texttt{send}(\text{Server1})$; //send nonce to Server1
$2 : \_ = \texttt{recv}(\text{Server1})$;
$3 : \texttt{send}(\text{User3})$; //forward nonce to User3
$4 : \_ = \texttt{recv}(\text{User3})$;

Actual $\mathcal{A}$(User3)**:**

$1 : \_ = \texttt{recv}(\text{User2})$;
$2 : \texttt{send}(\text{User2})$; //send nonce to User2

Figure 5.2: Actuals for all threads.

(a)

$\log(t)|_{\mathsf{Adversary}}$
$\langle\langle \mathsf{User1}, 1\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 2\rangle, \langle \mathsf{User1}, 2\rangle\rangle$
$\langle\langle \mathsf{User1}, 10\rangle, \langle \mathsf{Adversary}, 3\rangle\rangle,$
$\langle \mathsf{Adversary}, 4\rangle,$
$\langle\langle \mathsf{Adversary}, 5\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 2\rangle, \langle \mathsf{Adversary}, 6\rangle\rangle,$
$\langle \mathsf{Adversary}, 7\rangle,$
$\langle\langle \mathsf{Adversary}, 8\rangle, \langle \mathsf{Server1}, 5\rangle\rangle,$

$\log(t)|_{\mathsf{Server1}}$
$\langle\langle \mathsf{Adversary}, 5\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 2\rangle, \langle \mathsf{Adversary}, 6\rangle\rangle,$
$\langle\langle \mathsf{User2}, 1\rangle, \langle \mathsf{Server1}, 3\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 4\rangle, \langle \mathsf{User2}, 2\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 8\rangle, \langle \mathsf{Server1}, 5\rangle\rangle,$
$\langle \mathsf{Server1}, 6\rangle$
$\langle \mathsf{Server1}, 7\rangle$
$\langle \mathsf{Server1}, 8\rangle$

$\log(t)|_{\mathsf{User1}}$
$\langle\langle \mathsf{User1}, 1\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 2\rangle, \langle \mathsf{User1}, 2\rangle\rangle$
$\langle\langle \mathsf{User1}, 3\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 4\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 6\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 7\rangle\rangle,$
$\langle\langle \mathsf{Notary3}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$
$\langle \mathsf{User1}, 9\rangle$
$\langle\langle \mathsf{User1}, 10\rangle, \langle \mathsf{Adversary}, 3\rangle\rangle,$

$\log(t)|_{\mathsf{Notary1}}$
$\langle\langle \mathsf{User1}, 3\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 6\rangle\rangle,$

$\log(t)|_{\mathsf{Notary2}}$
$\langle\langle \mathsf{User1}, 4\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 7\rangle\rangle,$

$\log(t)|_{\mathsf{Notary3}}$
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary3}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$

$\log(t)|_{\mathsf{User2}}$
$\langle\langle \mathsf{User2}, 1\rangle, \langle \mathsf{Server1}, 3\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 4\rangle, \langle \mathsf{User2}, 2\rangle\rangle,$
$\langle\langle \mathsf{User2}, 3\rangle, \langle \mathsf{User3}, 1\rangle\rangle,$
$\langle\langle \mathsf{User2}, 4\rangle, \langle \mathsf{User3}, 2\rangle\rangle,$

$\log(t)|_{\mathsf{User3}}$
$\langle\langle \mathsf{User2}, 3\rangle, \langle \mathsf{User3}, 1\rangle\rangle,$
$\langle\langle \mathsf{User2}, 4\rangle, \langle \mathsf{User3}, 2\rangle\rangle,$

(b)

$l|_{\mathsf{Adversary}}$
$\langle\langle \mathsf{User1}, 1\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 2\rangle, \langle \mathsf{User1}, 2\rangle\rangle$
$\langle\langle \mathsf{User1}, 10\rangle, \langle \mathsf{Adversary}, 3\rangle\rangle,$
$\langle \mathsf{Adversary}, 4\rangle,$
$\langle\langle \mathsf{Adversary}, 5\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 2\rangle, \langle \mathsf{Adversary}, 6\rangle\rangle,$
$\langle \mathsf{Adversary}, 7\rangle,$
$\langle\langle \mathsf{Adversary}, 8\rangle, \langle \mathsf{Server1}, 5\rangle\rangle,$

$l|_{\mathsf{Server1}}$
$\langle\langle \mathsf{Adversary}, 5\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 2\rangle, \langle \mathsf{Adversary}, 6\rangle\rangle,$
$\langle\langle \mathsf{User2}, 1\rangle, \langle \mathsf{Server1}, 3\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 4\rangle, \langle \mathsf{User2}, 2\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 8\rangle, \langle \mathsf{Server1}, 5\rangle\rangle,$
$\langle \mathsf{Server1}, 6\rangle$
$\langle \mathsf{Server1}, 7\rangle$
$\langle \mathsf{Server1}, 8\rangle$

$l|_{\mathsf{User1}}$
$\langle\langle \mathsf{User1}, 1\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 2\rangle, \langle \mathsf{User1}, 2\rangle\rangle$
$\langle\langle \mathsf{User1}, 3\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 4\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 6\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 7\rangle\rangle,$
$\langle\langle \mathsf{Notary3}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$
$\langle \mathsf{User1}, 9\rangle$
$\langle\langle \mathsf{User1}, 10\rangle, \langle \mathsf{Adversary}, 3\rangle\rangle,$

$l|_{\mathsf{Notary1}}$
$\langle\langle \mathsf{User1}, 3\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 6\rangle\rangle,$

$l|_{\mathsf{Notary2}}$
$\langle\langle \mathsf{User1}, 4\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 7\rangle\rangle,$

$l|_{\mathsf{Notary3}}$
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary3}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$

$l|_{\mathsf{User2}}$
$\langle\langle \mathsf{User2}, 1\rangle, \langle \mathsf{Server1}, 3\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 4\rangle, \langle \mathsf{User2}, 2\rangle\rangle,$

(c)

$a_d|_{\mathsf{Adversary}}$
$\langle\langle \mathsf{User1}, 1\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 2\rangle, \langle \mathsf{User1}, 2\rangle\rangle$
$\langle\langle \mathsf{User1}, 10\rangle, \langle \mathsf{Adversary}, 3\rangle\rangle,$
$\langle \mathsf{Adversary}, 4\rangle,$
$\langle\langle \mathsf{Adversary}, 5\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 2\rangle, \langle \mathsf{Adversary}, 6\rangle\rangle,$
$\langle \mathsf{Adversary}, 7\rangle,$
$\langle\langle \mathsf{Adversary}, 8\rangle, \langle \mathsf{Server1}, 5\rangle\rangle,$

$a_d|_{\mathsf{Server1}}$
$\langle\langle \mathsf{Adversary}, 5\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 2\rangle, \langle \mathsf{Adversary}, 6\rangle\rangle,$

$\langle\langle \mathsf{Adversary}, 8\rangle, \langle \mathsf{Server1}, 5\rangle\rangle,$
$\langle \mathsf{Server1}, 6\rangle$
$\langle \mathsf{Server1}, 7\rangle$
$\langle \mathsf{Server1}, 8\rangle$

$a_d|_{\mathsf{User1}}$
$\langle\langle \mathsf{User1}, 1\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 2\rangle, \langle \mathsf{User1}, 2\rangle\rangle$
$\langle\langle \mathsf{User1}, 3\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 4\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$

$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 6\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 7\rangle\rangle,$

$\langle \mathsf{User1}, 9\rangle$
$\langle\langle \mathsf{User1}, 10\rangle, \langle \mathsf{Adversary}, 3\rangle\rangle,$

$a_d|_{\mathsf{Notary1}}$
$\langle\langle \mathsf{User1}, 3\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 6\rangle\rangle,$

$a_d|_{\mathsf{Notary2}}$
$\langle\langle \mathsf{User1}, 4\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 7\rangle\rangle,$

Figure 5.3: *Left to Right:* **(a):** $\log(t)|_i$ for $i \in I$. **(b):** Lamport cause l for Theorem 3. $l|_i = \emptyset$ for $i \in \{\mathsf{User3}\}$ as output by Definition 15. **(c):** Actual cause a_d for Theorem 3. $a_d|_i = \emptyset$ for $i \in \{\mathsf{Notary3}, \mathsf{User2}, \mathsf{User3}\}$. a_d is a projected *sublog* of Lamport cause l.

Chapter 6

Using Causation as a Building Block for Accountability

Chapter goal. In this chapter, we discuss how our theory of actual causation can be used for explanations (protocol debugging) and for assigning blame and blame exoneration. We conclude with a comparison with related work in accountability for security protocols and blame assignment.

6.1 Using causation for explanations (protocol debugging)

Generating explanations involves enhancing the epistemic state of an agent by providing information about the cause of an outcome [31]. Automating this process is useful for several tasks such as planning in AI-related applications and has also been of interest in the philosophy community [31, 63]. Causation has also been applied for explaining counter examples and providing explanations for errors in model checking [69, 70, 81, 82] where the abstract nature of the explanation provides insight about the model.

In prior work, Halpern and Pearl have defined explanation in terms of causality [31]. A fact, say E, constitutes an explanation for a previously established fact F in a given context, if had E been true then it would have been a sufficient cause of the established fact F. Moreover, having this information advances the prior epistemic state of the agent seeking the explanation, i.e. there exists a world (or a setting of the variables in Halpern and Pearl's model using structural equations) where F is not true but E is.

Our definition of cause (Chapter 3) could be used to explain violations arising from execution of programs in a given initial configuration. Given a log l, an initial configuration $\mathcal{C}_0$, and a

violation φ_V, our definition would pinpoint a sequence of program expressions, α, as an actual cause of the violation on the log. α would also be an explanation for the violation on l if having this causal information advances the epistemic knowledge of the agent. Note that there could be traces arising from the initial configuration where the behavior is inconsistent with the log. Knowing that α is consistent with the behavior on the log and that it is a cause of the violation would advance the agent's knowledge and provide an explanation for the violation.

Interventions in protocol design. Another aspect where our causal analysis might be useful is in designing better protocols. Prior work in social psychology and law [83] discusses that causal attributions can serve two main purposes: firstly, backward-looking functionality which enables one to assign blame or punishment for legal purposes that we discussed in the section above and, secondly, forward-looking functionality for causal judgments that aims at avoiding violating outcomes in the future. The second aspect can be exploited for protocol design.

For instance, if for an increasing number of violations, multiple norms are output as part of the causal sequence, this could suggest redesigning the protocol in order to minimize the norms which could be aiding the occurrence of the violation.

6.2 Using causation for blame attribution

Actual causation is an integral part of the prominent theories of blame in social psychology and legal settings [34, 35, 36, 37]. Most of these theories provide a comprehensive framework for blame which integrates causality, intentionality and foreseeability [34, 35, 83]. These theories recognize blame and cause as interrelated yet distinct concepts. Prior to attributing blame to an actor, a causal relation must be established between the actor's actions and the outcome. However, not all actions which are determined as a cause are blameworthy and an agent can be blamed for an outcome even if their actions were not a direct cause (for instance if an agent was responsible for another agent's actions). In our work we focus on the first aspect where we develop a theory for actual causation and provide a building block to find blameworthy programs from this set.

For instance, we can use the causal set output by the definitions in Section 5.2 and further narrow down the set to find blameworthy programs. Note that in order to use our definition as a building block for blame assignment, we require information about a) which of the executed programs deviate from the protocol, and b) which of these deviations are harmless. Some harmless deviants might be output as part of the causal set because their interaction is critical for the violation to occur. Definition 20 below provides one approach to removing such non-

blameworthy programs from the causal set. In addition we can filter the norms from the causal set.

For this purpose, we use the notion of protocol specified norms $\mathcal{N}$ introduced in Section 5.4. We impose an additional constraint on the norms, i.e., in the extreme counterfactual world where we execute norms only, there should be no possibility of violation. We call this condition *necessity*. Conceptually, necessity says that the reference standard (norms) we employ to assign blame is reasonable.

Definition 19 (Necessity condition for norms)**.** *Given $\langle I, \Sigma, \mathcal{N}, \varphi_V \rangle$, we say that $\mathcal{N}$ satisfies the necessity condition w.r.t. φ_V if for any trace t' starting from the initial configuration $\langle I, \mathcal{N}, \Sigma \rangle$, it is the case that $t' \notin \varphi_V$.*

We can use the norms $\mathcal{N}$ and the program cause X with its corresponding actual cause a_d from Phase 2 (Definitions 17, 18), in order to determine whether a program is a harmless deviant as follows. Definition 20 presents a sound (but not complete) approach for identifying harmless deviants.

Definition 20 (Harmless deviant)**.** *Let X be a program cause of violation V and a_d be the corresponding actual cause as determined by Definitions 17 and 18. We say that the program corresponding to index $i \in X$ is a harmless deviant w.r.t. trace t and violation φ_V if $\mathcal{A}(i)$ is deviant (i.e. $\mathcal{A}(i) \neq \mathcal{N}(i)$) and $a_d|_i$ is a prefix of $\mathcal{N}(i)$.*

For instance in our case study (Section 5.4), Theorem 3 outputs X and a_d (Figure 5.3) as a cause. X includes Server1. Considering Server1's norm (Figure 5.1), $\mathcal{A}\{\mathsf{Server}\}$ will be considered a deviant, but according to Definition 20, Server1 will be classified as a *harmless deviant* because $a_d|_{\mathsf{Server1}}$ is a prefix of $\mathcal{N}(\mathsf{Server1})$. Note that in order to capture blame attribution accurately, we will need a richer model which incorporates intentionality, epistemic knowledge and foreseeability, beyond causality.

Applications: comparing accountability solutions for PKI. The causal analysis techniques, proposed in this dissertation, can be used to perform a comparative analysis of prominent protocols which address weaknesses in the public key certificate infrastructure [50]. Our theory can be used to both provide blame assignment for well-known attacks and to provide a systematic basis for comparing different proposals. Specifically, we can analyze representative protocols based on mechanisms for key pinning [84, 85, 86] and multi-path probing [77, 87, 88] instead of relying on a single CA for verification (as we deal with in our example).

Our analysis in Section 5.4 demonstrates the feasibility of using such techniques to accurately pinpoint causes of security violations and for designing protocols. We can also use the cause definition to reason about which of the suggestions for enhancing accountability of pro-

tocols is more effective. In the absence of a general definition of accountability, one approach could be to look at the attack scenarios and determine whether an accountability solution ensures that if a violation is detected, the agents responsible for the violation can be pinpointed accurately. Such arguments could be used to inform the comparison between multiple approaches to enhance accountability.

6.3 Related work

Currently, there are multiple proposals for providing accountability in decentralized multi-agent systems [4, 6, 10, 12, 19, 20, 22, 23, 89]. The term accountability is used with varied interpretations. *Accountability* as a property provided by security protocols, in general, refers to the accuracy of pinpointing agents that are responsible for violating a protocol specification [4]. Here, the meaning of *responsibility* is not technically defined and is assumed to be clear in an intuitive sense. It is acknowledged that holding agents *accountable* for a violation in a protocol requires a justifiable basis. Many proposals in the literature regard deviation from protocol as a sufficient basis for holding an agent accountable while some others observe that deviation alone is not sufficient because some forms of deviance may be "irrelevant" to the violation [4, 19]. Feigenbaum et al. [19, 20] suggest the use of causation in context of accountability but use it to connect events to mediate punishment and do not focus on blame assignment.

Although the intrinsic relationship between causation and accountability is often acknowledged, the foundational studies of accountability do not explicitly incorporate the notion of cause in their formal definition or treat it as a blackbox concept without explicitly defining it. Our thesis is that accountability is not a trace property since evidence from the log alone does not provide a justifiable basis to blame agents. Inferring whether an agent's deviation on a log is blameworthy requires analyzing alternative settings where the agent might have behaved differently. We can use the absence or presence of violations in those settings to make a judgment about blame. Counterfactuals defined in analytical philosophy [27] provide a natural way to reason about alternative scenarios. As discussed in Chapter 4, prior work on actual causation in analytical philosophy and AI has considered counterfactual based causation in detail [14, 27, 30, 40, 41, 42]. These ideas have been applied for fault diagnosis where system components are analyzed, however these frameworks do not adequately capture all the elements crucial to model a security setting. Executions in security settings involve interactions among concurrently running programs in the presence of adversaries, and little can be assumed about the scheduling of events. We argue that ideas from actual causation can provide a justifiable basis for blame assignment in security settings if we use programs (or properties) as units

of blame rather than individual events.

We first discuss current approaches in defining accountability. We then discuss prior approaches that have used causality for blame assignment but not for security settings.

6.3.1 Accountability

Küsters et al [4] define a protocol P with associated accountability constraints that are rules of the form: if a particular property holds over runs of the protocol instances then particular agents may be blamed. Further, they define a judge J who gives a verdict over a run r of an instance π of a protocol P, where the verdict blames agents. In their work, Küsters et al assume that the accountability constraints for each protocol are given and complete. They state that the judge J should be designed so that J's verdict is fair and complete w.r.t. these accountability constraints. They design a judge separately for every protocol with a specific accountability property.

Küsters et al. [4] define accountability with respect to a set of constraints for each protocol and a judging procedure. The set of constraints can be thought of as augmenting the specification of the protocol to specify a priori what parties are to be blamed for a given violation. The judging procedure, which is also defined as a part of the protocol specification, provides a means to observe runs of the protocol and assign blame based on the specified accountability constraints. If a run of a protocol contains a violation, then the protocol is said to provide accountability if the judge's verdict is fair and complete: fairness states that no agent following the protocol (honest agent) is held accountable and completeness states that the judge blames at least one of the deviant (dishonest) agents mentioned in the accountability constraints. Küsters et al. [4] point to the significance of developing protocols that provide individual accountability (e.g. a judge can conclude exactly that parties A and B are blameworthy) rather than coarser-grained accountability (e.g. a judge can only conclude that at least one of the parties A and B is blameworthy but nothing more precise).

Küsters et al.'s definition of accountability has been successfully applied to substantial protocols such as voting, auctions, and contract signing. The authors have exposed previously undiscovered weaknesses in the protocols with respect to holding individual agents accountable when a violation is detected. Our work complements this line of work in that we aim to provide a semantic basis for arriving at such accountability constraints, thereby providing a justification for the blame assignment suggested by those constraints. Our actual cause definition can be viewed as a generic judging procedure that is defined independent of the violation and the protocol. We believe that using our cause definition as the basis for accountability constraints would also ensure the minimality of verdicts given by the judges.

Backes et al [6] define accountability as the ability to show evidence when an agent deviates. The authors analyze a contract signing protocol using protocol composition logic. In particular, the authors consider the case when the trusted third-party acts dishonestly and prove that the party can be held accountable by looking at a violating trace. This work can be viewed as a special case of the subsequent work of Küsters et al. [4] where the property associated with the violating trace is an example of an accountability constraint. In this line of work as well, the challenge lies in providing a method for designing constraints for a general class of protocols.

Feigenbaum et al [19, 20] also propose a definition of accountability that focuses on linking a violation to punishment. They focus on punishing agents who are accountable for a violation and provide two definitions of punishment. Their work uses Halpern and Pearl's definition [14, 30] of causality in order to define mediated punishment, where punishment is justified by the existence of a causal chain of events in addition to satisfaction of some utility conditions. We think that the use of causality in that setting is an apt choice. Causality is not used in case of automated punishment. However in case of mediated punishment, this work uses causality in order to remove events from a trace. Roughly, the trace containing a violation, is extended to a 'punishment' event. One of the clauses which tests whether punishment has been mediated, checks if the punishment event causally depends on the violating event. Feigenbaum et al provide their definitions using a trace-based framework similar to ours. The difference is that we focus on the interacting processes which executed on the trace. Instead of removing events from the trace, we remove the corresponding program expressions and then consider the entire trace set hence obtained. Feigenbaum et al directly manipulate traces by adding and removing events. In contrast, we modify the original programs (which generated the events) and then consider the traces.

The focus of their work differs from ours. The underlying ideas of our cause definition could be adapted to their framework to instantiate the causality notion that is currently used as a black box in their definition of mediated punishment. One key difference is that we focus on finding program actions that lead to the violation, which could explain why the violation happened while they focus on establishing a causal chain between violation and punishment events. While the event-based view is suitable for their work, we discuss in Chapter 4 why that view is not suitable for our purposes.

Jagadeesan et al [12] provide a definition of accountability in the context of authorization in distributed systems. Their analysis models an auditor as an honest agent of the protocol and uses game-based logics in order to study the tradeoffs between the requirements for honest agents and the audit protocols. The auditor considers the actions of agents in the protocol and can assign blame to the agents deviating from the protocol. In contrast, in our work, we

consider a trace-based model where a violation is detected on a finite trace and reason about the causal relationship between behaviors of interacting agents and the violation, not restricting our attention to deviant behavior only.

Haeberlen et al [10] provide a definition of accountability in distributed systems. Their system, PeerReview, maintains a record of all nodes' actions and provides non-repudiable evidence when a node deviates from its local protocol. The 'violation' occurs when a node does not respond to a message as expected by the protocol or sends a message not prescribed by the protocol. The motivation of this work differs from ours as we try to find causes of a global violation of a protocol specification. We differentiate between deviance, which occurs when an agent does not follow its locally specified program and a global violation, which occurs when a protocol property is not satisfied.

6.3.2 Causation for blame assignment

The work by Barth et al [89] provides a definition of accountability that uses the much coarser notion of Lamport causality, which is related to Phase 1 definition (Definition 15) in Part 2 of the dissertation. However, we use minimality checks and filter out *progress enablers* in Phase 2 (Definition 17) to obtain a finer determination of actual cause.

Gössler et al's work [22, 90] considers blame assignment for safety property violations where the violation of the global safety property implies that some components have violated their local specifications. They use a counterfactual notion of causality similar in spirit to ours to identify a subset of these faulty components as causes of the violation. The most recent work in this line applies the framework to real-time systems specified using timed automata [53].

A key technical difference between this line of work and ours is the way in which the contingencies to be considered in counterfactual reasoning are constructed. We have a program-based approach to leverage reasoning methods based on invariants and program logics. Gössler et al assume that a dependency relation that captures information flow between component actions are given and construct their contingencies using the traces of faulty components observed on the log as a basis. A set of faulty components is the necessary cause of the violation if the violation would disappear once the traces of these faulty components are modified to match the components' local specifications. They determine the longest prefixes of faulty components that satisfy the specification and replace the faulty suffixes with a correct one. Doing such a replacement without taking into account its impact on the behavior of other components that interact with the faulty components would not be satisfactory. Indeed, Wang et al [23] describe a counterexample to Gössler et al's work [22] where all causes are not found because of not being able to completely capture the effect of one component's behavior on another's. The most

recent definitions of Gössler et al [53, 90] address this issue by over approximating the parts of the log affected by the faulty components and replacing them with behavior that would have arisen had the faulty ones behaved correctly.

In constructing the contingencies to consider in counterfactual reasoning, we do not work with individual traces as Gössler et al. Instead, we work at the level of programs where "correcting" behavior is done by replacing program actions with those that do not have any effect on the violation other than enabling the programs to progress. The relevant contingencies follow directly from the execution of programs where such replacements have been done, without any need to develop additional machinery for reconstructing traces. Note also that we have a sufficiently fine-grained definition to pinpoint the minimal set of actions that make the component a part of the cause, where these actions may a part be of faulty or non-faulty programs. Moreover, we purposely separate cause determination and blame assignment because we believe that in the security setting, blame assignment is a problem that requires additional criteria to be considered such as the ability to make a choice, and intention. The work presented in this dissertation focuses on identifying cause as a *building block* for blame assignment.

"A thesis is not finished, it is abandoned."

—

Chapter 7

Conclusion and Future Work

7.1 Directions for future work

The directions for future work are divided into three main categories: The first direction involves developing definitions and analysis techniques for pinpointing properties of programs as actual causes. The second direction involves building a theory of intention and foreseeability and using these theories in combination with our theory of actual causation to provide a fine-grained definition of blame. The third direction involves an application of the proposed definitions to debugging programs in a sequential setting.

7.1.1 Properties as actual causes

This dissertation formalizes program expressions which model actions and choices, as actual causes and develops analysis techniques under the assumption that the complete event log is known. The definitions of actual cause presented in Chapters 3 and 5, assume that the entire log and the programs executed by every agent on the log are known, which may be hard to implement in practice. A challenge in security settings is that deviant programs executed by malicious agents may not be available for analysis; rather there will be evidence about certain actions committed by such agents. A generalized treatment accounting for such partial observability would be technically interesting and useful for other practical applications.

One interesting direction is to remove this assumption by formalizing *properties of programs* as actual causes. These definitions can then be used to analyze and compare the effectiveness of accountability solutions proposed to address weaknesses in web-based public key infrastructure, for web based pinning of certificates and multi-path probing techniques.

Defining actual cause in terms of properties of executed programs would require reasoning about *properties of programs* executed by agents, as opposed to reasoning about the exact program that was executed. This would allow us to designate properties as units of blame. One of the main challenges in extending the clauses to properties of programs lies in defining minimality of 'properties' (or invariants) of the purported cause X.

We present a brief definition outline. Here we directly focus on properties of programs, rather than specific actions within the program. Therefore, we lift the cause definition to program identifiers. Let l be a log of agents' actions and choices, where the log may not be complete. Let X be the set of identifiers for the programs which are a part of the purported cause set and Y denote the remaining programs. The evidence $\mathcal{E}(X)$ denotes a set of properties about the programs in X and $\mathcal{E}(Y)$ about the programs in Y. The basic idea of our definition is to use evidence from the log to partition the properties of concurrently executing programs into two sets X and Y such that we can establish properties of programs in X to be an actual cause of the violation φ_V. X is an actual cause of φ_V on a log l if:

- *Occurrence*: The violation has occurred on the log l. That is, the combination of evidence $\mathcal{E}(X)$ and $\mathcal{E}(Y)$ implies φ_V.
- *Necessity*: When all agents (apart from adversary) execute their prescribed programs X' and Y', then no resulting trace satisfies φ_V. We intend to prove that any set of invariants of the norms, X' and Y' implies the specification in order to be able to re-use the correctness proofs.
- *Sufficiency*: The combination of evidence $\mathcal{E}(X)$ from the log and the properties of programs in Y', $\mathcal{E}(Y')$, implies φ_V.
- *Non-redundancy*: No proper subset of X satisfies all three conditions above.
- *Minimality*: There does not exist a weaker property $\mathcal{E}'(X)$ which implies $\mathcal{E}(X)$.

Actual cause fingerprint. Another interesting direction would be to focus on a class of protocols for which the set of violating traces can be characterized as a finite number of disjunctive sets. The aim of this analysis would be to derive causal relations such that given a violating trace, there exists a unique set of agents which can be blamed.

Blaming properties of programs as opposed to programs could also be useful for developing this unique 'causal fingerprint' which indicates the exact set of agents to be blamed when a particular type of violation of the protocol specification is detected. The aim is to develop accountability constraints [4] for a special class of protocols, where the set of violations of the protocol specification can be classified as a finite number of disjunctive sets of attack traces.

Küsters et al [4] have discussed accountability constraints however they do not provide any semantic basis for such constraints. It would be interesting to develop constraints using causal analysis of each class of violating traces for a specific protocol. Developing an actual cause fingerprint in this manner can aid automation of blame assignment.

Other security applications. We have defined what it means for a sequence of program actions to be an actual cause of a violation of a security property. This question is motivated by security applications where agents can exercise their choice to either execute a prescribed program or deviate from it. While we demonstrate the value of this definition by analyzing a set of authentication failures as well as demonstrating its applicability to examples from AI/ philosophy, it would be interesting to explore applications to other protocols in which accountability concerns are central, in particular, protocols for electronic voting and secure multiparty computation in the semi-honest model.

7.1.2 Towards a theory of blame: intention, foreseeability

A problem with simply blaming all of the deviants whose expressions are a part of the causal sequence for the violation, is that agents that deviate may have actually been forced into the deviation by other agents. In that case, it appears that blaming the agent that forced the deviation is more appropriate than blaming the agent who had no choice but to deviate. Additionally, there have been recent studies which indicate that causation might depend on how people 's moral judgment and how they perceive blame, i.e. people might find as cause an agent who they would intuitively blame [91]. An interesting extension of this dissertation will be to understand these connections by defining intention, foreseeability and the connection with blame formally. This direction will also involve defining accountability to be sensitive to forced deviations.

7.1.3 Actual causation in sequential setting

When adapted for the sequential setting where execution differences of a single program are analyzed, our approach could open up the possibility of an automated causal analysis. One scenario where such automation is particularly useful is software debugging. Often, in software debugging, a lot of manual effort goes into identifying the cause of software bugs. Prior work [82] on developing causal analysis techniques for detecting errors via counterexamples output all lines in the coding base which differ from intended execution order. Using the causal techniques can help pinpoint the source of the software bug or narrow down the search space significantly.

7.2 Concluding remarks

In this dissertation, we have proposed a definition of cause that can be used to identify a sequence of program expressions and choices which are an actual cause of a violation in a system of interacting processes. Our definition is inspired by prior work on counterfactual-based and process-based actual causation [14, 16, 17, 44, 45, 46, 68]. Our design considerations are motivated by the security domain where researchers are interested in finding who to hold accountable for security violations as well as to fix faulty protocol design. To the best of our knowledge, this is the first instance of an interaction-aware approach to actual causation being proposed for blame assignment in such protocol based settings, i.e. to connect actions to occurrence of violation in interacting systems. Our approach blends ideas from both process-based and counterfactual-based approaches to actual causation in AI and analytical philosophy.

This work demonstrates the importance of interaction and choice in order to formulate actual causation as a building block for accountability in interacting systems. Our definition formalizes program expressions as causes, which is a suitable abstraction level and is a useful *building block* for several such applications, in particular for providing explanations, assigning blame and providing accountability guarantees for security protocols. The applicability of ideas in this dissertation extends beyond protocol settings and these concepts can be applied in different (non-deterministic) settings where causal inferences and interactions are significant.

Appendices

Appendix A

Operational Semantics

Syntax. The syntax given here includes the internal choice operator, the asymmetric disjunction as well as the `assert` statement introduced in Chapter 5. Our syntax is given using the A-normal form [58] where every term contains only one connective and all operands contain only variables. The syntax consists of values v for variables x, actions α and expressions e. Values v include boolean values, numerical values and all other return values (such as keys or cipher text). ζ denotes primitive functions for thread local computations, discussed in Chapter 5.

$$
\begin{array}{lrcl}
\text{Values} & v & ::= & x \mid \texttt{true} \mid \texttt{false} \mid 1 \mid 2 \ldots \\
\text{Actions} & \alpha & ::= & v \mid \texttt{send}(v) \mid \texttt{recv}() \mid \zeta(v) \mid v_1 \wedge v_2 \mid v_1 \vee v_2 \mid \\
 & & & \neg v \mid v_1 ?\, v_2 :\, v_3 \mid v_1 \oplus\, v_2 \mid v_1 [] \, v_2 \mid \ldots \\
\text{Expressions} & e & ::= & v \mid (b : x = \alpha); e_1 \mid \texttt{assert}(v); e_1
\end{array}
$$

Operational Semantics. Selected rules of the operational semantics of the programming language described above are shown in Figures A.1 and A.2. The symmetric disjunction evaluates to true as long as one of the terms evaluates to true. For the asymmetric disjunction, if the left disjunct evaluates to true, the label record `l`. If the right disjunct evaluates to true, the label record `r`. If both the disjuncts are true, then either one of the rules can fire non-deterministically. If both the disjuncts are false, the label records `n`. Note that we can represent an internal choice over more terms by nesting the choice operator as shown in Chapter 4.4.3.1. In the rules give below, an operator marked with $\dot{}$ indicates the standard semantic interpretation of the operator. Note that σ does not change in these rules but is included because some applications may require actions that affect state.

$\boxed{T \hookrightarrow T'}$

$$\frac{}{I\,,(b : x = v; e)\,,\sigma \overset{\langle I,b\rangle}{\hookrightarrow} I\,,e\{v/x\}\,,\sigma}\text{red-var}$$

$$\frac{}{I\,,((b : x = \zeta(v)); e)\,,\sigma \overset{\langle I,b\rangle}{\hookrightarrow} I\,,e\{\dot{\zeta}(v)/x\}\,,\sigma}\text{red-act}$$

$$\frac{}{I\,,(b : x = v_1 \ \vee\ v_2); e\,,\sigma \overset{\langle I,b\rangle}{\hookrightarrow} I\,,e\{v_1 \dot{\vee} v_2/x\}\,,\sigma}\text{red-symmetric-disj}$$

$$\frac{}{I\,,(b : x = v_1 \oplus\ v_2); e\,,\sigma \overset{\langle I,b,1\rangle}{\hookrightarrow} I\,,e\{v_1/x\}\,,\sigma}\text{red-int-choice1}$$

$$\frac{}{I\,,(b : x = v_1 \oplus\ v_2); e\,,\sigma \overset{\langle I,b,\mathrm{r}\rangle}{\hookrightarrow} I\,,e\{v_2/x\}\,,\sigma}\text{red-int-choice2}$$

$$\frac{}{I\,,(b : x = \texttt{true}\ [\!]\ v_2); e\,,\sigma \overset{\langle I,b,1\rangle}{\hookrightarrow} I\,,e\{\texttt{true}/x\}\,,\sigma}\text{red-asymmetric-disj1}$$

$$\frac{}{I\,,(b : x = v_1\ [\!]\ \texttt{true}); e\,,\sigma \overset{\langle I,b,\mathrm{r}\rangle}{\hookrightarrow} I\,,e\{\texttt{true}/x\}\,,\sigma}\text{red-asymmetric-disj2}$$

$$\frac{}{I\,,(b : x = \texttt{false}\ [\!]\ \texttt{false}); e\,,\sigma \overset{\langle I,b,\mathrm{n}\rangle}{\hookrightarrow} I\,,e\{\texttt{false}/x\}\,,\sigma}\text{red-asymmetric-disj3}$$

$$\frac{}{I\,,(b : x = v_1 \ \wedge\ v_2); e\,,\sigma \overset{\langle I,b\rangle}{\hookrightarrow} I\,,e\{v_1 \dot{\wedge} v_2/x\}\,,\sigma}\text{red-conj}$$

$$\frac{}{I\,,(b : x = \neg v_1); e\,,\sigma \overset{\langle I,b\rangle}{\hookrightarrow} I\,,e\{\dot{\neg} v_1/x\}\,,\sigma}\text{red-neg}$$

$$\frac{}{I\,,(b : x = \texttt{true}\ ?\ v_2\ :\ v_3); e\,,\sigma \overset{\langle I,b\rangle}{\hookrightarrow} I\,,v_2;\ e\,,\sigma}\text{red-cond1}$$

$$\frac{}{I\,,(b : x = \texttt{false}\ ?\ v_2\ :\ v_3); e\,,\sigma \overset{\langle I,b\rangle}{\hookrightarrow} I\,,v_3;\ e\,,\sigma}\text{red-cond2}$$

$$\frac{}{I\,,(\texttt{assert}(\texttt{true}); e)\,,\sigma \overset{\epsilon}{\hookrightarrow} I\,,e\,,\sigma}\text{red-assert}$$

Figure A.1: Operational semantics. An operator marked with $\dot{}$ indicates the standard semantic interpretation of the operator.

$\boxed{\mathcal{C} \longrightarrow \mathcal{C}'}$

Internal reduction

$$\frac{T_i \overset{r}{\hookrightarrow} T_i'}{\ldots, T_i, \ldots \overset{r}{\longrightarrow} \ldots, T_i', \ldots}\text{red-config}$$

Communication actions

$$\frac{}{\ldots, \langle I_s\,, (b_s : x = \mathtt{send}(v); e_s)\,, \sigma_s\rangle, \langle I_r\,, ((b_r : y = \mathtt{recv}(); e_r\,, \sigma_r\rangle, \ldots \overset{\langle\langle I_s, b_s\rangle, \langle I_r, b_r\rangle\rangle}{\longrightarrow} \ldots, \langle I_s\,, e_s[0/x]\,, \sigma_s\rangle, \langle I_r\,, e_r[v/y]\,, \sigma_r\rangle, \ldots}\text{red-comm}$$

Figure A.2: Operational semantics (contd.)

Appendix B

Proof for Case Study: Program Actions as Causes

We model an instance of our running example based on passwords in order to demonstrate our actual cause definition. As explained in Section 5.1, we consider a protocol session where Server1, User1, User2, User3 and multiple notaries interact over an adversarial network to establish access over a password-protected account. In parallel for this scenario, we assume the log also contains interactions of a second server (Server2), one notary (Notary4, not contacted by User1, User2 or User3) and another user (User4) who follow their norms for account access. These threads do not interact with threads {User1, Server1, Notary1, Notary2, Notary3, Adversary, User2, User3}. The protocol has been described in detail below.

B.A Protocol description

We consider our example protocol with eleven threads named {Server1, User1, User2, User3, Adversary, Notary1, Notary2, Notary3, Notary4, Server2, User4}. The *norms* for all these threads, except Adversary are shown in Figure B.1. The actual violation is caused because some of the executing programs are different from the norms. These actual programs, called $\mathcal{A}$ as in Section 5.2, are shown later. The norms are shown here to help the reader understand what the ideal protocol is.

In this case study, we have two servers (Server1, Server2) running the protocol with two different users (User1, User4) and each server allocates account access separately. The norms in Figure B.1 assume that User1's and User4's accounts (called $acct_1$ and $acct_2$ in Server1's and Server2's norm respectively) have been created already. User1's password, pwd_1 is associated

with User1's user id $uid1$. Similarly User4's password pwd_2 is associated with its user id $uid2$. This association (in hashed form) is stored in Server1's local state at pointer mem_1 (and at mem_2 for Server2). The norm for Server1 is to wait for a request from an entity, respond with its public key, then wait for a password encrypted with that public key and grant access to the requester if the password matches the previously stored value in Server1's memory at mem_1. To grant access, Server1 adds an entry into a private access matrix, called P_1. (A separate server thread, not shown here, allows User1 to access its resource if this entry exists in P_1.)

The norm for User1 is to send an access request to Server1, wait for the server's public key, verify that key with three notaries and then send its password pwd_1 to Server1, encrypted under Server1's public key. On receiving Server1's public key, User1 initiates a protocol with the three notaries and accepts or rejects the key based on the response of a majority of the notaries.

The norm for User4 is the same as that for User1 except that it interacts with Server2. Note that User4 only verifies the public key with one notary, Notary4. The norm for Server2 is the same as that for Server1 except that it interacts with User4.

In parallel, the norm for User2 is to generate and send a nonce to User3. The norm for User3 is to receive a message from User2, generate a nonce and send it to User2.

Each notary has a private database of *(public_key, principal)* tuples. The norms here assume that this database has already been created correctly. When User1 or User4 send a request with a public key, the notary responds with the principal's identifier after retrieving the tuple corresponding to the key in its database. (Note that, in this simple example, we identify threads with principals, so the notaries just store an association between public keys and their threads.)

B.B Preliminaries

Notation. The programs in this example use several primitive functions. $\texttt{Enc}(k, m)$ and $\texttt{Dec}(k', m)$ denote encryption and decryption of message m with key k and k' respectively. $\texttt{Hash}(m)$ generates the hash of term m. $\texttt{Sig}(k, m)$ denotes message m signed with the key k, paired with m in the clear. pub_key_i and pvt_key_i denote the public and private keys of thread i, respectively. For readability, we include the intended recipient i and expected sender j of a message as the first argument of $\texttt{send}(i, m)$ and $\texttt{recv}(j)$ expressions. As explained earlier, i and j are ignored during execution and a network adversary, if present, may capture or inject any messages. $\texttt{Send}(i, j, m)$ @ u holds if thread i sends message m to thread j at time u and $\texttt{Recv}(i, j, m)$ @ u hold if thread i receives message m from thread j at time u. $P_1(u)$ and $P_2(u)$ denotes the tuples in the permission matrices at time u. Initially P_1 and P_2 do not contain any access permissions.

Norm $\mathcal{N}$(Server1):

$1 : (uid1, n1) = \texttt{recv}(j)$; //access req from thread j
$2 : n2 = \text{new}$;
$3 : \texttt{send}(j, (pub\_key\_\text{Server1}, n2, n1))$; //sign and send public key
$4 : s1 = \texttt{recv}(j)$; //encrypted $uid1, pwd1$ from j, alongwith its thread id J
$5 : (n3, uid1, pwd1, J) = \texttt{Dec}(pvt\_key\_\text{Server1}, s1)$;
$6 : t = \texttt{Hash}(uid1, pwd1)$;
$\texttt{assert}(mem_1 = t)$ //compare hash with stored hash value for same uid
$7 : \texttt{insert}(P_1, (acct_1, J))$;

Norm $\mathcal{N}$(User1):

$1 : n1 = \text{new}$;
$2 : \texttt{send}(\text{Server1}, (uid1, n1))$; //access request
$3 : (pub\_key1, n2, n1) = \texttt{recv}(j)$; //key from j
$4 : n3, n4, n5 = \text{new}$;
$5 : \texttt{send}(\text{Notary1}, pub\_key1, n3)$;
$6 : \texttt{send}(\text{Notary2}, pub\_key1, n4)$;
$7 : \texttt{send}(\text{Notary3}, pub\_key1, n5)$;
$8 : \texttt{Sig}(pvt\_key\_\text{Notary1}, (pub\_key1, l1, n3)) = \texttt{recv}(\text{Notary1})$; //notary1 responds
$9 : \texttt{Sig}(pvt\_key\_\text{Notary2}, (pub\_key1, l2, n4)) = \texttt{recv}(\text{Notary2})$; //notary2 responds
$10 : \texttt{Sig}(pvt\_key\_\text{Notary3}, (pub\_key1, l3, n5)) = \texttt{recv}(\text{Notary3})$; //notary3 responds
$\texttt{assert}(\text{At least two of \{l1,l2,l3\} equal Server1})$
$11 : t = \texttt{Enc}(pub\_key1, n2, (uid1, pwd1, \text{User1}))$;
$12 : \texttt{send}(\text{Server1}, t)$; //send t to Server1;

Norms $\mathcal{N}$(Notary1), $\mathcal{N}$(Notary2), $\mathcal{N}$(Notary3), $\mathcal{N}$(Notary4):

// o denotes Notary1, Notary2, Notary3 or Notary4
$1 : (pub\_key, n1) = \texttt{recv}(j)$;
$2 : pr = \texttt{KeyOwner}(pub\_key)$; //lookup key owner
$3 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_o, (pub\_key, pr, n1)))$; //signed certificate;

Norm $\mathcal{N}$(Server2):

$1 : (uid2, n1) = \texttt{recv}(j)$; //access req from thread j
$2 : n2 = \text{new}$;
$3 : \texttt{send}(j, (pub\_key\_\text{Server2}, n2, n1))$;
$4 : s1 = \texttt{recv}(j)$; //encrypted $uid2, pwd2$ from j, alongwith its thread id J
$5 : (n2, uid2, pwd2, J) = \texttt{Dec}(pvt\_key\_\text{Server2}, s1)$;
$6 : t = \texttt{Hash}(uid2, pwd2)$;
$\texttt{assert}(mem_2 = t)$ //compare hash with stored hash value for same uid
$7 : \texttt{insert}(P_2, (acct_2, J))$;

Norm $\mathcal{N}$(User4):

$1 : n1 = \text{new}$;
$2 : \texttt{send}(\text{Server2}, (uid2, n1))$; //access request
$3 : pub\_key, n2, n1 = \texttt{recv}(j)$; //key from j
$4 : n3 = \text{new}$;
$5 : \texttt{send}(\text{Notary4}, pub\_key, n3)$;
$6 : \texttt{Sig}(pvt\_key\_\text{Notary4}, (pub\_key, l1, n3)) = \texttt{recv}(\text{Notary4})$; //notary4 responds
$\texttt{assert}(\{\text{l1}\} \text{ equals Server2})$
$7 : t = \texttt{Enc}(pub\_key, n2, (uid2, pwd2, \text{User4}))$;
$8 : \texttt{send}(\text{Server2}, t)$; //send t to Server2;

Norm $\mathcal{N}$(User2):

$1 : n1 = \text{new}$;
$2 : \texttt{send}(\text{User3}, (n1))$;
$3 : \texttt{Sig}(pvt\_key\_j, (n2, n1)) = \texttt{recv}(\text{User3})$; 4 :

Norm $\mathcal{N}$(User3):

$1 : n1 = \texttt{recv}(\text{User2})$;
$2 : n2 = \text{new}$;
$3 : \texttt{send}(\text{User3}, \texttt{Sig}(pvt\_key\_\text{User3}, (n2, n1)))$;

Figure B.1: Norms for Server1, User1, Server2, User4, User2, User3 and the notaries. Adversary's norm is the trivial empty program.

Assumptions. (A1)

$$\texttt{HonestThread}(\mathsf{Server1}, \mathcal{A}(\mathsf{Server1}))$$

We are interested in security guarantees about users who create accounts by interacting with the server and who do not share the generated password or user-id with any other principal except for sending it according to the roles specified in the program given below.

(A2)

$$\texttt{HonestThread}(\mathsf{User1}, \mathcal{A}(\mathsf{User1}))$$

(A3)

$$\texttt{HonestThread}(\mathsf{Adversary}, \mathcal{A}(\mathsf{Adversary}))$$

(A4)

$$\texttt{HonestThread}(\mathsf{Notary1}, \mathcal{A}(\mathsf{Notary1}))$$

(A5)

$$\texttt{HonestThread}(\mathsf{Notary2}, \mathcal{A}(\mathsf{Notary2}))$$

(A6)

$$\texttt{HonestThread}(\mathsf{Notary3}, \mathcal{A}(\mathsf{Notary3}))$$

(A7)

$$\texttt{HonestThread}(\mathsf{Notary4}, \mathcal{A}(\mathsf{Notary4}))$$

(A8)

$$\texttt{HonestThread}(\mathsf{Server2}, \mathcal{A}(\mathsf{Server2}))$$

(A9)

$$\texttt{HonestThread}(\mathsf{User4}, \mathcal{A}(\mathsf{User4}))$$

(A10)

$$\texttt{HonestThread}(\mathsf{User2}, \mathcal{A}(\mathsf{User2}))$$

(A11)

$$\texttt{HonestThread}(\mathsf{User3}, \mathcal{A}(\mathsf{User3}))$$

A principal following the protocol never shares its keys with any other entity. We also assume that the encryption scheme in semantically secure and non-malleable. Since we identify threads with principals therefore each of the threads are owned by principals with the same identifier, for instance Server1 owns the thread that executes the program A(Server1).

(Start1)

$$Start(i) \ @ \ -\infty$$

where i refers to all the threads in the set described above.

Security property. The security property of interest to us is that if at time u, a thread k is given access to account a, then k owns a. Specifically, in this example, we are interested in the $a = acct_1$ and $k = \mathsf{User1}$. This can be formalized by the following logical formula, $\neg\varphi_V$:

$$\forall u, k.\ (acct_1, k) \in P_1(u) \supset (k = \mathsf{User1}) \tag{B.1}$$

Here, $P_1(u)$ is the state of the access control matrix P_1 for Server1 at time u.

The actuals for all threads are shown in Figure B.2 and B.3.

B.C Attack

As an illustration, we model the "Compromised Notaries" violation of Section 5.1. The programs executed by all threads are given in Figures B.2 and B.3. User1 sends an access request to Server1 which is intercepted by Adversary who sends its own key to User1 (pretending to be Server1). User1 checks with the three notaries who falsely verify Adversary's public key to be Server1's key. Consequently, User1 sends the password to Adversary. Adversary then initiates a protocol with Server1 and gains access to the User1's account. Note that the actual programs of the three notaries attest that the public key given to them belongs to Server1. In parallel, User2 sends a request to Server1 and receives a response from Server1. Following this interaction, User2 interacts with User3, as in their norms. User4, Server2 and Notary4 execute their actuals in order to access the account $acct_2$ as well.

Figure B.4 shows the expressions executed by each thread on the property-violating trace. For instance, the label $\langle\langle \mathsf{User1}, 1\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle$ indicates that both User1 and Adversary executed the expressions with the line number 1 in their actual programs, which resulted in a synchronous communication between them, while the label $\langle \mathsf{Adversary}, 4\rangle$ indicates the local execution of the expression at line 4 of Adversary's program. The initial configuration has the programs:
$\{\mathcal{A}(\mathsf{User1}), \mathcal{A}(\mathsf{Server1}), \mathcal{A}(\mathsf{Adversary}), \mathcal{A}(\mathsf{Notary1}), \mathcal{A}(\mathsf{Notary2}), \mathcal{A}(\mathsf{Notary3}), \mathcal{A}(\mathsf{User2}),$
$\mathcal{A}(\mathsf{User3}), \mathcal{A}(\mathsf{User4}), \mathcal{A}(\mathsf{Server2}),$
$\mathcal{A}(\mathsf{Notary4})\}$. For this attack scenario, the concrete trace t we consider is such that $\log(t)$ is any *arbitrary interleaving* of the actions for $X_1 =$

Actual $\mathcal{A}$(Adversary)

$1 : (uid1, n1) = \texttt{recv}(j)$; //intercept req from User1
$2 : n2 = \texttt{new}$;
$3 : \texttt{send}(\text{User1}, (pub\_key\_\text{Adversary}1, n2, n1))$; //send key to User1
$4 : s = \texttt{recv}(\text{User1})$; //pwd from User
$5 : n2, uid1, pwd1, \text{User1} = \texttt{Dec}(pvt\_key\_\text{Adversary}, s)$; //decrypt pwd;
$6 : n3 = \texttt{new}$;
$7 : \texttt{send}(\text{Server1}, (uid1, n3))$; //access request to Server
$8 : pub\_key, n4, n3 = \texttt{recv}(\text{Server1})$;
$9 : t = \texttt{Enc}(pub\_key, (n4, uid1, pwd1, \text{Adversary}))$; //encrypt pwd
$10 : \texttt{send}(\text{Server1}, t)$; //pwd to Server1

Actuals $\mathcal{A}$(Notary1), $\mathcal{A}$(Notary2), $\mathcal{A}$(Notary3)**:**

// o denotes Notary1, Notary2 or Notary3
$1 : (pub\_key\_\text{Adversary}, n1) = \texttt{recv}(j)$;
$2 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_o, (pub\_key\_\text{Adversary}, \text{Server1}, n1))$; //signed certificate to j;

Actual $\mathcal{A}$(Server1)**:**

$1 : (uid1, n1) = \texttt{recv}(j)$; //access req from thread j
$2 : n2 = \texttt{new}$;
$3 : \texttt{send}(j, (pub\_key\_\text{Server}1, n2, n1))$;
$4 : n4 = \texttt{recv}(j)$; //receive nonce from thread User2
$5 : n5 = \texttt{new}$;
$6 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_\text{Server}1, (n5, n4)))$;
$7 : s1 = \texttt{recv}(j)$; //encrypted $uid1, pwd1$ from j, alongwith its thread id J
$8 : (n3, uid1, pwd1, J) = \texttt{Dec}(pvt\_key\_\text{Server}1, s1)$;
$9 : t = \texttt{Hash}(uid1, pwd1)$;
$\texttt{assert}(mem_1 = t)$**[A]** //compare hash with stored hash value for same uid
$10 : \texttt{insert}(P_1, (acct_1, J))$;

Actual $\mathcal{A}$(User1)**:**

$1 : n1 = \texttt{new}$;
$2 : \texttt{send}(\text{Server1}, (uid1, n1))$; //access request
$3 : (pub\_key, n2, n1) = \texttt{recv}(j)$; //key from j
$4 : n3, n4, n5 = \texttt{new}$;
$5 : \texttt{send}(\text{Notary1}, pub\_key, n3)$;
$6 : \texttt{send}(\text{Notary2}, pub\_key, n4)$;
$7 : \texttt{send}(\text{Notary3}, pub\_key, n5)$;
$8 : \texttt{Sig}(pvt\_key\_\text{Notary}1, (pub\_key, l1, n3)) = \texttt{recv}(\text{Notary1})$; //notary1 responds
$9 : \texttt{Sig}(pvt\_key\_\text{Notary}2, (pub\_key, l2, n4)) = \texttt{recv}(\text{Notary2})$; //notary2 responds
$10 : \texttt{Sig}(pvt\_key\_\text{Notary}3, (pub\_key, l3, n5)) = \texttt{recv}(\text{Notary3})$; //notary3 responds
$\texttt{assert}(\text{At least two of}\{l1, l2, l3\}\text{equal Server1})$; **[B]** //
$11 : t = \texttt{Enc}(pub\_key, n2, (uid1, pwd1, \text{User1}))$;
$12 : \texttt{send}(\text{Server1}, t)$; //send t to Server1;

Actual $\mathcal{A}$(User2)**:**

$1 : n1 = \texttt{new}$;
$2 : \texttt{send}(\text{Server1}, (n1))$;
$3 : \texttt{Sig}(pvt\_key, (n2, n1)) = \texttt{recv}(\text{Server1})$;
$4 : \texttt{send}(\text{User3}, (n2))$;
$5 : \texttt{Sig}(pub\_key, n3, n2) = \texttt{recv}(\text{User3})$;

Actual $\mathcal{A}$(User3)**:**

$1 : n1 = \texttt{recv}(\text{User2})$;
$2 : n2 = \texttt{new}$;
$3 : \texttt{send}(\text{User3}, \texttt{Sig}(pvt\_key\_\text{User}3, n2, n1))$;

Figure B.2: Actuals for Adversary, Notary1, Notary2, Notary3, Server1, User1, User2, User3

Actual $\mathcal{A}$(Server2):

$1 : (uid2, n1) = \texttt{recv}(j);$ //access req from thread j
$2 : n2 = \texttt{new};$
$3 : \texttt{send}(j, (pub\_key\_\text{Server2}, n2, n1));$
$4 : s1 = \texttt{recv}(j);$ //encrypted $uid2, pwd2$ from j, alongwith its thread id J
$5 : (n2, uid2, pwd2, J) = \texttt{Dec}(pvt\_key\_\text{Server2}, s1);$
$6 : t = \texttt{Hash}(uid2, pwd2);$
$\texttt{assert}(mem_2 = t)$ //(C)compare hash with stored hash value for same uid
$7 : \texttt{insert}(P_2, (acct_2, J));$

Actual $\mathcal{A}$(User4):

$1 : n1 = \texttt{new};$
$2 : \texttt{send}(\text{Server2}, (uid2, n1));$ //access request
$3 : \texttt{Sig}(pub\_key, n2, n1) = \texttt{recv}(j);$ //key from j
$4 : n3 = \texttt{new};$
$5 : \texttt{send}(\text{Notary4}, pub\_key, n3);$
$6 : \texttt{Sig}(pvt\_key\_\text{Notary4}, (pub\_key, l1, n3)) = \texttt{recv}(\text{Notary4});$ //notary4 responds
$\texttt{assert}(\{\text{l1}\} \text{ equals Server2})(D)$
$7 : t = \texttt{Enc}(pub\_key, n2, (uid2, pwd2, \text{User4}));$
$8 : \texttt{send}(\text{Server2}, t);$ //send t to Server2;

Actual $\mathcal{A}$(Notary4):

// o denotes Notary1, Notary2, Notary3 or Notary4
$1 : (pub\_key, n1) = \texttt{recv}(j);$
$2 : pr = \texttt{KeyOwner}(pub\_key);$ //lookup key owner
$3 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_o, (pub\_key, pr, n1)));$ //signed certificate;

Figure B.3: Actuals for Server2, User4, Notary4

$\{\mathsf{Adversary}, \mathsf{User}1, \mathsf{Server}1, \mathsf{Notary}1, \mathsf{Notary}2, \mathsf{Notary}3, \mathsf{User}2, \mathsf{User}3\}$ and $X_2 = \{\mathsf{Server}2, \mathsf{User}4, \mathsf{Notary}4\}$ shown in Figure B.4(a) and Figure B.5. Any such interleaved log is denoted $\log(t)$ in the sequel. At the end of this log, $(acct_1, \mathsf{Adversary})$ occurs in the access control matrix P_1, but Adversary does not own $acct_1$. Hence, this log corresponds to a violation of our security property.

Note that, if any two of the three notaries had attested the Adversary's key to belong to Server1, the violation would have still happened. Consequently, we may expect three independent program causes in this example: {Adversary, User1, Server1, Notary1, Notary2} with the action causes a_d as shown in Figure B.4(c), {Adversary, User1, Server1, Notary1, Notary3} with the actions a'_d, and {Adversary, User1, Server1, Notary2, Notary3} with the actions a''_d where a'_d and a''_d can be obtained from a_d (Figure B.4(c)) by considering actions for {Notary1, Notary3} and {Notary2, Notary3} respectively, instead of actions for {Notary1, Notary2}. The following theorem states that our definitions determine exactly these three independent causes.

Theorem 4. *Let* $I = \{\mathsf{User}1, \mathsf{Server}1, \mathsf{Adversary}, \mathsf{Notary}1, \mathsf{Notary}2, \mathsf{Notary}3, \mathsf{Notary}4, \mathsf{Server}2, \mathsf{User}4, \mathsf{User}2, \mathsf{User}3\}$, *and* Σ *and* $\mathcal{A}$ *be as described above. Let* t *be a trace from* $\langle I, \mathcal{A}, \Sigma\rangle$ *such that* $log(t)|_i$ *for each* $i \in I$ *matches the corresponding log projection from Figures B.4(a) and B.5. Then, Definition 18 determines three possible values for the program cause* X *of violation* $t \in \varphi_V$*: {*Adversary*,* User*1,* Server*1,* Notary*1,* Notary*2}, {*Adversary*,* User*1,* Server*1,* Notary*1,* Notary*3}, and {*Adversary*,* User*1,* Server*1,* Notary*2,* Notary*3} where the corresponding actual causes are* a_d, a'_d *and* a''_d *respectively.*

It is instructive to understand the proof of this theorem, as it illustrates our definitions of causation. We verify that our Phase 1 and Phase 2 definitions (Definitions 15, 17, 18) yield exactly the three values for X mentioned in the theorem.

Lamport cause (Phase 1). We show that any l whose projections match those shown in Figure B.4(b) satisfies sufficiency and minimality. From Figure B.4(b), such an l has no actions for User3, User4, Notary4, Server2 and only those actions of User2 that are involved in synchronization with Server1. For all other threads, the log contains every action from t. The intuitive explanation for this l is straightforward: Since l must be a (projected) *prefix* of the trace, and the violation only happens because of `insert` in the last statement of Server1's program, every action of every program before that statement in Lamport's happens-before relation must be in l. This is exactly the l described in Figure B.4(b).

Formally, following the statement of sufficiency, let T be the set of traces starting from $\mathcal{C}_0 = \langle I, \mathcal{A}, \Sigma\rangle$ (Figure B.2) whose logs contain l as a projected prefix. Pick any $t' \in T$. We need to show $t' \in \varphi_V$. However, note that any t' containing all actions in l must also add

(a)

$\log(t)|_{\mathsf{Adversary}}$
$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle \mathsf{Adversary}, 2\rangle,$
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle,$
$\langle\langle \mathsf{User1}, 12\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle,$
$\langle \mathsf{Adversary}, 5\rangle,$
$\langle \mathsf{Adversary}, 6\rangle,$
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle,$
$\langle \mathsf{Adversary}, 9\rangle,$
$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 7\rangle\rangle,$

$\log(t)|_{\mathsf{User1}}$
$\langle \mathsf{User1}, 1\rangle,$
$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle,$
$\langle \mathsf{User1}, 4\rangle,$
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 7\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 9\rangle\rangle,$
$\langle\langle \mathsf{Notary3}, 2\rangle, \langle \mathsf{User1}, 10\rangle\rangle,$
$\langle \mathsf{User1}, 11\rangle,$
$\langle\langle \mathsf{User1}, 12\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle,$

$\log(t)|_{\mathsf{Server1}}$:
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle \mathsf{Server1}, 2\rangle,$
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle,$
$\langle\langle \mathsf{User2}, 2\rangle, \langle \mathsf{Server1}, 4\rangle\rangle,$
$\langle \mathsf{Server1}, 5\rangle,$
$\langle\langle \mathsf{Server1}, 6\rangle, \langle \mathsf{User2}, 3\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 7\rangle\rangle,$
$\langle \mathsf{Server1}, 8\rangle,$
$\langle \mathsf{Server1}, 9\rangle,$
$\langle \mathsf{Server1}, 10\rangle,$

$\log(t)|_{\mathsf{Notary1}}$:
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$

$\log(t)|_{\mathsf{Notary2}}$:
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 9\rangle\rangle,$

$\log(t)|_{\mathsf{Notary3}}$:
$\langle\langle \mathsf{User1}, 7\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary3}, 2\rangle, \langle \mathsf{User1}, 10\rangle\rangle,$

$\log(t)|_{\mathsf{User2}}$:
$\langle \mathsf{User2}, 1\rangle,$
$\langle\langle \mathsf{User2}, 2\rangle, \langle \mathsf{Server1}, 4\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 6\rangle, \langle \mathsf{User2}, 3\rangle\rangle,$
$\langle\langle \mathsf{User2}, 4\rangle, \langle \mathsf{User3}, 1\rangle\rangle,$
$\langle\langle \mathsf{User3}, 3\rangle, \langle \mathsf{User2}, 5\rangle\rangle,$

$\log(t)|_{\mathsf{User3}}$:
$\langle\langle \mathsf{User2}, 4\rangle, \langle \mathsf{User3}, 1\rangle\rangle,$
$\langle \mathsf{User3}, 2\rangle,$
$\langle\langle \mathsf{User3}, 3\rangle, \langle \mathsf{User2}, 5\rangle\rangle,$

(b)

$l|_{\mathsf{Adversary}}$
$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle \mathsf{Adversary}, 2\rangle,$
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle,$
$\langle\langle \mathsf{User1}, 12\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle,$
$\langle \mathsf{Adversary}, 5\rangle,$
$\langle \mathsf{Adversary}, 6\rangle,$
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle,$
$\langle \mathsf{Adversary}, 9\rangle,$
$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 7\rangle\rangle,$

$l|_{\mathsf{User1}}$
$\langle \mathsf{User1}, 1\rangle$
$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle,$
$\langle \mathsf{User1}, 4\rangle,$
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 7\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 9\rangle\rangle,$
$\langle\langle \mathsf{Notary3}, 2\rangle, \langle \mathsf{User1}, 10\rangle\rangle,$
$\langle \mathsf{User1}, 11\rangle,$
$\langle\langle \mathsf{User1}, 12\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle,$

$l|_{\mathsf{Server1}}$:
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle \mathsf{Server1}, 2\rangle,$
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle,$
$\langle\langle \mathsf{User2}, 2\rangle, \langle \mathsf{Server1}, 4\rangle\rangle,$
$\langle \mathsf{Server1}, 5\rangle,$
$\langle\langle \mathsf{Server1}, 6\rangle, \langle \mathsf{User2}, 3\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 7\rangle\rangle,$
$\langle \mathsf{Server1}, 8\rangle,$
$\langle \mathsf{Server1}, 9\rangle,$
$\langle \mathsf{Server1}, 10\rangle,$

$l|_{\mathsf{Notary1}}$:
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$

$l|_{\mathsf{Notary2}}$:
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 9\rangle\rangle,$

$l|_{\mathsf{Notary3}}$:
$\langle\langle \mathsf{User1}, 7\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary3}, 2\rangle, \langle \mathsf{User1}, 10\rangle\rangle,$

$l|_{\mathsf{User2}}$:
$\langle \mathsf{User2}, 1\rangle,$
$\langle\langle \mathsf{User2}, 2\rangle, \langle \mathsf{Server1}, 4\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 6\rangle, \langle \mathsf{User2}, 3\rangle\rangle,$

(c)

$a_d|_{\mathsf{Adversary}}$
$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle \mathsf{Adversary}, 2\rangle,$
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle,$
$\langle\langle \mathsf{User1}, 12\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle,$
$\langle \mathsf{Adversary}, 5\rangle,$
$\langle \mathsf{Adversary}, 6\rangle,$
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle,$
$\langle \mathsf{Adversary}, 9\rangle,$
$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 7\rangle\rangle,$

$a_d|_{\mathsf{User1}}$
$\langle \mathsf{User1}, 1\rangle$
$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle,$
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle$
$\langle \mathsf{User1}, 4\rangle,$
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$

$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 9\rangle\rangle,$

$\langle \mathsf{User1}, 11\rangle,$
$\langle\langle \mathsf{User1}, 12\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle,$

$a_d|_{\mathsf{Server1}}$:
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle,$
$\langle \mathsf{Server1}, 2\rangle$
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle,$

$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 7\rangle\rangle,$
$\langle \mathsf{Server1}, 8\rangle,$
$\langle \mathsf{Server1}, 9\rangle,$
$\langle \mathsf{Server1}, 10\rangle,$

$a_d|_{\mathsf{Notary1}}$:
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary1}, 2\rangle, \langle \mathsf{User1}, 8\rangle\rangle,$

$a_d|_{\mathsf{Notary2}}$:
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle,$
$\langle\langle \mathsf{Notary2}, 2\rangle, \langle \mathsf{User1}, 9\rangle\rangle,$

Left to Right: **(a):** $\log(t)|_i$ for $i \in \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary2}, \mathsf{Notary3}, \mathsf{User2}, \mathsf{User3}\}$. **(b):** Lamport cause l for Theorem 4. $l|_i = \emptyset$ for $i \in \{\mathsf{Notary4}, \mathsf{Server2}, \mathsf{User4}, \mathsf{User3}\}$ as output by Definition 15. **(c):** Actual cause a_d for Theorem 4. $a_d|_i = \emptyset$ for $i \in \{\mathsf{Notary3}, \mathsf{Notary4}, \mathsf{Server2}, \mathsf{User4}, \mathsf{User2}, \mathsf{User3}\}$. a_d is a projected *sublog* of Lamport cause l.

Figure B.4: $\log(t)|_i$ and a_d

$\log(t)|_{\mathsf{Server2}}$:
$\langle\langle \mathsf{User}4, 2\rangle, \langle \mathsf{Server}2, 1\rangle\rangle$,
$\langle \mathsf{Server}2, 2\rangle$,
$\langle\langle \mathsf{Server}2, 3\rangle, \langle \mathsf{User}4, 3\rangle\rangle$,
$\langle\langle \mathsf{User}4, 8\rangle, \langle \mathsf{Server}2, 4\rangle\rangle$,
$\langle \mathsf{Server}2, 5\rangle$,
$\langle \mathsf{Server}2, 6\rangle$,
$\langle \mathsf{Server}2, 7\rangle$,

$\log(t)|_{\mathsf{User1}}$
$\langle \mathsf{User}4, 1\rangle$,
$\langle\langle \mathsf{User}4, 2\rangle, \langle \mathsf{Server}2, 1\rangle\rangle$,
$\langle\langle \mathsf{Server}2, 3\rangle, \langle \mathsf{User}4, 3\rangle\rangle$,
$\langle \mathsf{User}4, 4\rangle$,
$\langle\langle \mathsf{User}4, 5\rangle, \langle \mathsf{Notary}4, 1\rangle\rangle$,
$\langle\langle \mathsf{Notary}4, 3\rangle, \langle \mathsf{User}4, 6\rangle\rangle$,
$\langle \mathsf{User}4, 7\rangle$,
$\langle\langle \mathsf{User}4, 8\rangle, \langle \mathsf{Server}2, 4\rangle\rangle$,

$\log(t)|_{\mathsf{Notary}4}$:
$\langle\langle \mathsf{User}4, 5\rangle, \langle \mathsf{Notary}4, 1\rangle\rangle$,
$\langle \mathsf{Notary}4, 2\rangle$,
$\langle\langle \mathsf{Notary}4, 3\rangle, \langle \mathsf{User}4, 6\rangle\rangle$,

Figure B.5: $\log(t)|_i$ where $i \in \{\mathsf{User}4, \mathsf{Server}2, \mathsf{Notary}4\}$

$(acct_1, \mathsf{Adversary})$ to P_1, but $\mathsf{Adversary} \neq \mathsf{User1}$. Hence, $t' \in \varphi_V$. Further, l is minimal as described in the previous paragraph.

Actual cause (Phase 2). Phase 2 (Definitions 17, 18) determines three independent program causes for X: {Adversary, User1, Server1, Notary1, Notary2}, {Adversary, User1, Server1, Notary1, Notary3}, and {Adversary, User1, Server1, Notary2, Notary3} with the actual action causes given by a_d, a'_d and a''_d, respectively in Figure B.4(c). These are symmetric, so we only explain why a_d satisfies Definition 17. (For this a_d, Definition 18 immediately forces $X = \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary2}\}$.) We show that (a) a_d satisfies sufficiency', and (b) No proper *sublog* of a_d satisfies sufficiency' (minimality'). Note that a_d is obtained from l by dropping Notary3, User2 and User3, and all their interactions with other threads.

We start with (a). Let a_d be such that $a_d|_i$ matches Figure B.4(c) for every i. Fix any dummifying function f. We must show that any trace originating from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, whose log contains a_d as a projected sublog, is in φ_V. Additionally we must show that there is such a trace. There are two potential issues in mimicking the execution in a_d starting from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ — first, with the interaction between User1 and Notary3 and, second, with the interaction between Server1 and User2. For the first interaction, on line 7, $\mathcal{A}(\mathsf{User1})$ (Figure B.2) synchronizes with Notary3 according to l, but the synchronization label does not exist in a_d. However, in $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, the `recv()` on line 10 in $\mathcal{A}(\mathsf{User1})$ is replaced with a dummy value, so the execution from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ progresses. Subsequently, the majority check (assertion [B]) succeeds as in l, because two of the three notaries (Notary1 and Notary2) still attest the Adversary's key.

A similar observation can be made about the interaction between Server1 and User2. Line 4, $\mathcal{A}(\mathsf{Server1})$ (from Figure B.4(b)) synchronizes with User2 according to l, but this synchronization label does not exist in a_d. However, in $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, the `recv()` on line 4 in $\mathcal{A}(\mathsf{Server1})$ is replaced with a dummy value, so the execution from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ progresses. Subsequently, Server1 still adds permission for the Adversary.

Next we prove that every trace starting from $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$, whose log contains a_d (Figure B.4(c)) as a projected *sublog*, is in φ_V. Fix a trace t' with log l'. Assume l' coincides with a_d. We show $t' \in \varphi_V$ as follows:

1. Since the synchronization labels in l' are a superset of those in a_d, Server1 must execute line 10 of its program $\mathcal{A}(\mathsf{Server1})$ in t'. After this line, the access control matrix P_1 contains $(acct_1, J)$ for some J.
2. When $\mathcal{A}(\mathsf{Server1})$ writes (x, J) to P_1 at line 10, then J is the third component of a tuple obtained by decrypting a message received on line 7.

3. Since the synchronization projections on l' are a superset of a_d, and on a_d $\langle \mathsf{Server1}, 7\rangle$ synchronizes with $\langle \mathsf{Adversary}, 10\rangle$, J must be the third component of an encrypted message sent on line 10 of $\mathcal{A}(\mathsf{Adversary})$.
4. The third component of the message sent on line 10 by Adversary is exactly the term "Adversary". (This is easy to see, as the term "Adversary" is hardcoded on line 9.) Hence, $J = \mathsf{Adversary}$.
5. This immediately implies that $t' \in \varphi_V$ since $(acct_1, \mathsf{Adversary}) \in P_1$, but $\mathsf{Adversary} \neq \mathsf{User1}$.

Last, we prove (b) — that no proper subsequence of a_d satisfies sufficiency'. Note that a_d (Figure B.4(c)) contains exactly those actions from l (Figure B.4) on whose returned values the last statement of Server1's program (Figure B.2) is data or control dependent. Consequently, all of a_d as shown is necessary to obtain the violation.

In particular, observe that if labels for Server1 ($a_d|_{\mathsf{Server1}}$) are not a part of a'_d, then Server1's labels are not in $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ and, hence, on any counterfactual trace Server1 cannot write to P_1, thus precluding a violation. Therefore, the sequence of labels in $a_d|_{\mathsf{Server1}}$ are required in the actual cause.

By sufficiency', for any f, the log of trace t' of $\mathsf{dummify}(I, \mathcal{A}, \Sigma, a_d, f)$ must contain a_d as a projected *sublog*. This means that in t', the assertion [A] of $\mathcal{A}(\mathsf{Server1})$ must succeed and, hence, on line 7, the correct password pwd_1 must be received by Server1, independent of f. This immediately implies that Adversary's action of sending that password must be in a_d, else some dummified executions will have the wrong password sent to Server1 and the assertion [A] will fail.

Extending this logic further, we now observe that because Adversary forwards a password received from User1 (line 4 of $\mathcal{A}(\mathsf{Adversary})$) to Server1, the send action of User1 will be in a_d (otherwise, some dummifications of line 4 of $\mathcal{A}(\mathsf{Adversary})$ will result in the wrong password being sent to Server1, a contradiction). Since User1's action is in a_d and l' must contain a_d as a *sublog*, the majority check of $\mathcal{A}(\mathsf{User1})$ must also succeed. This means that at least two of $\{\mathsf{Notary1}, \mathsf{Notary2}, \mathsf{Notary3}\}$ must send the confirmation to User1, else the dummification of lines 8 – 10 of $\mathcal{N}(\mathsf{User1})$ will cause the assertion [B] to fail for some f. Since we are looking for a minimal *sublog* therefore we only consider the send actions from two threads i.e. $\{\mathsf{Notary1}, \mathsf{Notary2}\}$. At this point we have established that each of the labels as shown in Figure B.4(c) are required in a_d. Hence, $a'_d = a_d$.

Appendix C

Defining Programs as Actual Causes

Note: This is a preliminary interaction-aware theory of actual causation which finds programs as causes of violations. This theory explores ideas such as occurrence, necessity, sufficiency and minimality similar to prior work. In Chapter 5, we explain why we require a more fine grained approach of program actions for our purpose of blame assignment. Nevertheless, the ideas used in the formalization of program actions as actual causes in Chapter 5 are inspired from this approach and an interested reader can trace the connections as well as the recurrent theme of interaction-aware approach to actual causation.

The central contributions of this section are the following: We initiate a formal study of *programs as actual causes*. Specifically, we define what it means for a set of programs to be an actual cause of a violation when they run concurrently with a set of other programs. We also present a sound technique for establishing programs as actual causes building on prior work on proving security properties of protocols [92, 93]. We demonstrate the value of this approach by providing a cause analysis of a representative protocol designed to address weaknesses in the current public key certification infrastructure.

C.A Programs as actual causes

The formalization of programs as actual causes proceeds in three stages. The first stage captures two conditions. The *Occurrence* condition ensures that a safety property was violated on the log. The log records what programs actually executed and how they interacted. This model of a log is appropriate in settings where executed programs are available for analysis during forensics. The *Necessity* condition ensures that in a counterfactual scenario where all actually executed programs are replaced by the corresponding norms (i.e., the protocol specified ideal

programs), the violation goes away on all resulting traces.

The second stage identifies a set X of the actual programs as the suspected cause by requiring three conditions. The *Closure* condition ensures that every program that interacted with programs in X on the log is also in X. The *Sufficiency* condition requires that all traces resulting from the execution of X in conjunction with the norms for the other programs restores the violation so long as the programs in X interact in a way consistent with the log. Finally, the *Minimality* condition requires that no subset of X satisfies *Closure* and *Sufficiency*.

The third stage further separates the set X into X_d (the actual causes) and X_p (the progress enablers). Informally, the actual values sent by the programs in X_p are irrelevant to the violation—they merely serve to enable the progress of other programs that is essential for the violation. Roughly, this idea is formalized by replacing messages received by X_d from X_p by dummy values. An additional *Sufficiency'* condition requires that the violation is restored under this transformation so long as the programs in X_d interact in a way consistent with the log. A *Minimality'* condition requires that no subset of X_d satisfies *Sufficiency'*.

Our nearest neighbor is a treatment of actual causation by Halpern and Pearl [14, 15]. There are analogies between their conditions of *Occurrence*, *Necessity*, *Sufficiency*, and *Minimality* and our corresponding conditions in the first two stages. However, there are some important points of difference. First, in our non-deterministic, interactive program setting, it is critical to ensure that all interacting agents are identified in the *Closure* condition and the actual interactions of the suspected causes are held fixed as we move from *Occurrence* to the *Sufficiency* condition. There is no analogous concept in the work of Halpern and Pearl. Second, we exploit special characteristics of our security settings to simplify the definition. Specifically, in the *Necessity* condition when considering counterfactuals (alternative hypothetical scenarios in which the violation does not occur), it is essential to restrict the set of hypothetical scenarios to avoid counter-intuitive cause determinations. Halpern [15] recognizes this problem and considers only “more normal worlds" in his definition. While it may be challenging to figure out how to rank worlds according to normality in certain application domains, our insight is that the protocol-specified prescribed programs provide a natural basis for constructing such a ranking function. Finally, our third stage enables us to separate out the programs in X_d whose information flows are causally related to the violation from the programs in X_p who are just progress enablers but do not contribute relevant information flows. This distinction is important in security settings but is not considered by Halpern and Pearl.

C.A.1 Problematic example

The above definition sketch may deem norms as part of the causal set X. For instance, consider the "Compromising Notaries" example presents in Section 5.1. Server is part of the causal set in both the attacks even though it follows its norm and we do not wish to hold it accountable for the violation. In order to do so, a natural approach would be to filter the set X by removing all of the parties that follow their norms and hold all of the remaining parties accountable.

One problem with this approach is that it might not provide adequate basis for blame assignment in the presence of independent causes. Consider an attack where Server does not store passwords securely and gets hacked by Adversary2. In this case Server is clearly deviant. In parallel, consider the occurrence of *Compromising Notaries*. The above discussed cause definition will output two independent causes, {Server, Adversary2} and {User, Adversary1, Server, Notary1, Notary2}. In this case since Server is deviant, it would not be removed from the second set, even though it did not act in a deviant manner when interacting with User and Adversary1. Developing a definition based on actions of programs eliminates this problem since in the first set, the 'deviant actions' of Server's program will be a part of the causal set and in the second set, the norm of the Server would be part of the causal set. Similarly for the example of Server sending a harmless 'hello' message to User might deem the Server as deviant and prevent a causal filtering operation based on norms to exonerate Server. However analyzing program actions will also allow us to remove harmless deviants from the causal set.

Another issue with the above formalization concerns the Closure condition. If there exist two independent causes on the log and both of these causes interact with the final program, for instance two adversaries might interact independently with a server and cause an attack. The restriction of finding closed set over programs would make it problematic to find these adversaries as independent causes since both will be connected with Server's program via the same interaction structure. Working with program actions instead, helps segregate different parts of programs which are affected by each of the causes. As discussed in Section 5.3, Definition 3 provides the most general form of our cause definition, and adequately deals with these issues.

Next, we describe the formalization.

C.A.2 Formal definitions

C.A.2.1 Auxiliary definitions

To define actual causation, we find it convenient to introduce the notion of the log of a trace, which is just the sequence of labels on the trace.

Definition 21 (Log). *Given a trace $t = \mathcal{C}_0 \xrightarrow{r_0} \mathcal{C}_1 \ldots \xrightarrow{r_{m-1}} \mathcal{C}_m$, the log of the trace t, $log(t)$, is the sequence $r_0, \ldots, r_{m-1}$.*

The letter l denotes logs. We need a few more straightforward definitions on logs in order to define actual causation. In the sequel, X, Y, Z denote sets of thread identifiers. A sublog of $l = r_0, \ldots, r_m$ is a subsequence of $r_0, \ldots, r_m$.

Definition 22 (Closed set of thread identifiers). *A set of thread identifiers X is said to be closed on log l if for every synchronization label $\langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle \in l$, $i_s \in X$ if and only if $i_r \in X$.*

Intuitively, X is closed on l if threads in X synchronize only amongst themselves according to l.

Definition 23 (Local projection of a log). *The local projection of a log l with respect to a thread identifier i, written $l \downarrow_i$, is the sublog of l containing all local labels whose thread identifier is i. Formally,*

$$
\begin{array}{lcll}
\bullet \downarrow_i & = & \bullet & \\
(\langle i, b\rangle :: l) \downarrow_i & = & \langle i, b\rangle :: (l \downarrow_i) & \\
(\langle j, b\rangle :: l) \downarrow_i & = & l \downarrow_i & \text{if } i \neq j \\
(\langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle :: l) \downarrow_i & = & l \downarrow_i &
\end{array}
$$

Definition 24 (Synchronization projection of a log). *The synchronization projection of a log l with respect to a set of thread identifiers X, written $l|_X$, is the sublog of l containing synchronization labels from l both of whose thread identifiers are in X. Formally,*

$$
\begin{array}{lcll}
\bullet|_X & = & \bullet & \\
(\langle j, b\rangle :: l)|_X & = & l|_X & \\
(\langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle :: l)|_X & = & \langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle :: (l|_X) & \text{if } i_s, i_r \in X \\
(\langle\langle i_s, b_s\rangle, \langle i_r, b_r\rangle\rangle :: l)|_X & = & l|_X & \text{if } i_s \notin X \vee i_r \notin X
\end{array}
$$

A log l' is called consistent with a log l relative to a set of thread identifiers X if X is closed on l' and the local and synchronization labels of threads in X are identical (and identically ordered) in l and l'. This is formalized below.

Definition 25 (Log consistency). *A log l' is said to be consistent with a log l relative to a set of thread identifiers X if the following hold:*

1. *X is closed on l'*
2. *$\forall i \in X.\ l' \downarrow_i = l \downarrow_i$*
3. *$l'|_X = l|_X$*

C.A.2.2 Actual cause definition

Our goal is to define actual causes of the violation of an expected security property. By property, we mean any safety property of labelled traces that is closed under permutation of reductions that are unrelated to each other in Lamport's "happens-before" relation [59]. We use φ_V to denote the complement of such a property (i.e., φ_V is the set of property-violating traces).

Consider a trace t starting from the initial configuration $\mathcal{C}_0 = \langle I, \mathcal{A}, \Sigma \rangle$. Suppose $t \in \varphi_V$, so this trace violates the safety property $\neg\varphi_V$. Our definition of actual causation identifies a subset of the threads I as the cause of the violation. To do this, we employ a set of hypothetical counterfactual scenarios, in which subsets of the programs $\mathcal{A}(i)$, $i \in I$ are replaced by *norms* or the correct, prescribed programs. Consequently, we assume that we are provided a second function $\mathcal{N} : I \rightarrow$ Expressions such that $\mathcal{N}(i)$ is the program that *ideally should have been* executing in the thread i. Note that for some i, $\mathcal{A}(i)$ and $\mathcal{N}(i)$ may be equal, but $\mathcal{A}(i)$ may still have contributed to the violation in collaboration with other threads j for which $\mathcal{A}(j) \neq \mathcal{N}(j)$. For each i, we call $\mathcal{N}(i)$ the norm for thread i and $\mathcal{A}(i)$ the "actual" for thread i.

We formalize a violation and the corresponding norms in a *violation structure*, to which our definition of actual causation applies. We impose two conditions on a violation structure. First, there must actually be a violation, else looking for causes is meaningless. We call this condition *occurrence*. Second, in the extreme counterfactual world where we execute norms only, there should be no possibility of violation. We call this condition *necessity*. Conceptually, necessity says that the reference standard (norms) we employ to define causes is reasonable.

Definition 26 (Violation structure)**.** *A violation structure is a tuple $V = \langle I, \mathcal{A}, \Sigma, \mathcal{N}, \varphi_V, t \rangle$ such that t is a trace starting from $\langle I, \mathcal{A}, \Sigma \rangle$ and the following two conditions hold:*

1. *(**Occurrence**) $t \in \varphi_V$.*
2. *(**Necessity**) For any trace t' starting from the initial configuration $\langle I, \mathcal{N}, \Sigma \rangle$, it is the case that $t' \notin \varphi_V$.*

When considering counterfactuals, we often replace the actuals in a subset Y of the threads I with their norms. The following definition captures the resulting initial configuration.

Definition 27 (Normification)**.** *Given a violation structure $V = \langle I, \mathcal{A}, \Sigma, \mathcal{N}, \varphi_V, t \rangle$ and a partition (X, Y) of I, we define the normified initial configuration* $\mathsf{norm}(V, X, Y) = \langle I, \mathcal{E}, \Sigma \rangle$*, where*

$$\mathcal{E}(i) \;=\; \begin{cases} \mathcal{N}(i) \text{ if } i \in Y \\ \mathcal{A}(i) \text{ if } i \in X \end{cases}$$

Armed with these definitions, we are now in a position to formally define programs as actual causes. Our definition applies in two phases. The first phase identifies *suspected causes*.

Technically, a suspected cause is a minimal, closed set of threads that can account for the violation, even if all other threads are replaced by norms. In the second phase, we refine this set into *actual causes* and *progress enablers*. The latter contribute only indirectly to the cause by enabling the actual causes to make progress; the exact values transmitted by progress enablers are irrelevant.

Our Phase 1 definition below determines suspected causes. It contains two conditions. The *sufficiency* condition tests that the suspected causes, when combined with norms for the remaining threads, suffice to recreate the violation. Technically, we consider all traces from a normified counterfactual, in which the candidate suspected causes (called X in the definition) follow the same sequence of reductions as in the original trace. A key criteria is that X be closed on the violating trace's log. This is important, because any thread that communicates with a suspected cause may have at the least enabled progress of the latter and, hence, contributed to the violation. The *minimality* condition tests that the identified suspected causes contain no redundant threads.

Definition 28 (Suspected Cause of Violation: Phase 1)**.** *Let $V = \langle I, \mathcal{A}, \Sigma, \mathcal{N}, \varphi_V, t\rangle$ be a violation structure and $l = log(t)$. We say that $X \subseteq I$ is a suspected cause of the violation V if the following hold:*

1. ***(Closure)*** *X is closed on l.*
2. ***(Sufficiency)*** *Let $Y = I \backslash X$ and $\mathcal{C}_0' = \mathsf{norm}(V, X, Y)$. Let T be the set of traces starting from $\mathcal{C}_0'$ that are log-consistent with l relative to X. Then, T is non-empty and $T \subseteq \varphi_V$.*
3. ***(Minimality)*** *No proper subset X' of X satisfies conditions 1 and 2.*

The Phase 1 definition above identifies a minimal set X of threads, which is sufficient to cause the violation and does not interact with other threads (called Y). In the Phase 2 definition below, we further partition X into X_d (actual cause) and X_p (progress enablers) such that the threads in X_p contribute only towards the *progress* of other threads that cause the violation. In other words, the set X_p contains all threads whose actual transmitted values are irrelevant.

Briefly, here's how our Phase 2 definition works. We first pick a candidate set $X_d \subseteq X$ (where X is the suspected cause set identified in Phase 1) and define $X_p = X \backslash X_d$. We consider counterfactual traces obtained from initial configurations in which threads from X_p (the hypothesized progress enablers) are completely dropped and, instead, any inputs that threads of X_d received from X_p are replaced by arbitrary dummy values and, additionally, the synchronizations within X_d are the same as in the original violating trace. If a violation appears in all such counterfactual traces, then the partition of X into X_d and X_p is a good candidate. Of all such good candidates, we choose those with minimal X_d (or, equivalently, maximal X_p).

The key technical difficulty in writing this definition is replacing values communicated from

X_p to X_d with arbitrary dummy values. While there are many ways to do this, we choose a simple method: We syntactically transform initial expressions of threads in X_d, replacing every `recv`(), which synchronized with an expression in X_p, with a dummy value. Since our communication model is synchronous, we must also erase all `send`() expressions from threads in X_d, if the recipient was in X_p. The following definition formalizes this idea. It defines a new initial configuration obtained by replacement with dummy values in X_d, removal of X_p and normification of threads outside X_d and X_p. The function f supplies dummy values for use in replacement. In the Phase 2 definition, we quantify universally over this function.

Definition 29 (Dummifying transformation)**.** *Let $V = \langle I, \mathcal{A}, \Sigma, \mathcal{N}, \varphi_V, t\rangle$ be a violation structure, X_d, X_p, Y be disjoint subsets of I, $l = log(t)$ and $f : I \times$ LineNumbers $\to Terms$. The dummifying transform* dummify(V, X_d, X_p, f) *is the initial configuration $\langle I \backslash X_p, \mathcal{E}, \Sigma\rangle$, where $\mathcal{E}$ is defined as follows:*

- *For $i \in (I \backslash X_p) \backslash X_d$, $\mathcal{E}(i) = \mathcal{N}(i)$.*
- *For $i \in X_d$, $\mathcal{E}(i)$ is $\mathcal{A}(i)$ modified as follows:*
  - *If $\langle\langle j, b_s\rangle, \langle i, b_r\rangle\rangle \in l$ and $j \in X_p$, then replace b_r : `recv`() in $\mathcal{A}(i)$ with $b_r : f(i, b_r)$.*
  - *If $\langle\langle i, b_s\rangle, \langle j, b_r\rangle\rangle \in l$ and $j \in X_p$, then replace b_s : `send`(t) in $\mathcal{A}(i)$ with $b_s : 0$.*

Definition 30 (Actual Cause of Violation: Phase 2)**.** *Let $V = \langle I, \mathcal{A}, \Sigma, \mathcal{N}, \varphi_V, t\rangle$ be a violation structure and $l = log(t)$. Let X be a suspected cause of the violation V determined by Definition 28. We say that $X_d \subseteq X$ is an actual cause of the violation V if the following hold:*

1. ***(Sufficiency')** Let $X_p = X \backslash X_d$. For every f, if $\mathcal{C}_0' =$ dummify(V, X_d, X_p, f) and T is the set of traces starting from $\mathcal{C}_0'$ that are log-consistent with l relative to X_d, then T is non-empty and $T \subseteq \varphi_V$.*
2. ***(Minimality')** No proper subset X_d' of X_d satisfies condition 1.*

More than one minimal set X_d may satisfy the above Phase 2 definition for a given violation V. Every such X_d is deemed an *independent* actual cause of the violation.

C.B Case study

We model an instance of our running example based on passwords in order to demonstrate our actual cause definition. As explained in Section 5.1, we consider a protocol session where User1, Server1 and multiple notaries interact over an adversarial network to establish access over a password-protected account. In parallel for this scenario, we assume the log also contains interactions of a second server (Server2), one notary (Notary4, not contacted by User1) and a second user (User2) who follow their norms for account creation. These threads do not interact

with threads {User1, Server1, Notary1, Notary2, Notary3, Adversary}. The protocol has been described in detail below.

C.B.1 Protocol description

We consider our example protocol with nine threads named {Server1, User1, Adversary, Notary1, Notary2, Notary3, Notary4, Server2, User2}. The norms for all these threads, except Adversary are shown in Figure C.1 and C.2. The Adversary's norm is empty, because in an ideal world, the Adversary should not participate. In this case study, we have two servers (Server1, Server2) running the protocol with two different users (User1, User2) and each server allocates account access separately.

The norms in Figures C.1, C.2 assume that User1's and User2's accounts (called $acct_1$ and $acct_2$ in Server1's and Server2's norm respectively) have been created already. User1's password, pwd_1 is associated with User1's user id $uid1$. Similarly User2's password pwd_2 is associated with its user id $uid2$. User1 generated pwd_1 associated with $acct_1$ and User2 created pwd_2 associated with $acct_2$. This association (in hashed form) is stored in Server1's local state at pointer mem_1 (and at mem_2 for Server2). The norm for Server1 is to wait for a request from an entity, respond with its public key, then wait for a password encrypted with that public key and grant access to the requester if the password matches the previously stored value in Server1's memory at mem_1. To grant access, Server1 adds an entry into a private access matrix, called P_1. (A separate server thread, not shown here, allows User1 to access its resource if this entry exists in P_1.)

The norm for User1 is to send the password $pwd1$ to Server1, encrypted under Server1's public key and not share the password with any other agent. On receiving Server1's public key, User1 initiates a protocol with the three notaries and accepts or rejects the key based on the response of a majority of the notaries.

The norm for User2 is the same as that for User1 except that it interacts with Server2. Note that User2 only verifies the public key with one notary, Notary4.

The norm for Server2 is the same as that for Server1 except that it interacts with User2.

Each notary has a private database of *(public_key, principal)* tuples. The norms here assume that this database has already been created correctly. When User1 or User2 send a request with a public key, the notary responds with the principal's identifier after retrieving the tuple corresponding to the key in its database. (Note that, in this simple example, we identify threads with principals, so the notaries just store an association between public keys and their threads.)

C.B.1.1 Preliminaries

Notation. The norms in Figures C.1, C.2 use several primitive functions. pub_key_i and pvt_key_i denote the public and private keys of thread i, respectively. For a given (public or private) key k, Inv(k) denotes the corresponding private or public key for k. For readability, we include the intended recipient i and expected sender j of a message as the first argument of $\texttt{send}(i, m)$ and $\texttt{recv}(j)$ expressions. As explained earlier, i and j are ignored during execution and the adversary, if present, may capture or inject messages. $P_1(u)$ and $P_2(u)$ denotes the tuples in the permission matrices at time u. Initially P_1 and P_2 do not contain any access permissions.

Action Predicates and terms.

- $\texttt{Send}(i, j, m)$ @ u holds if thread i sends message m to thread j at time u.
- $\texttt{Recv}(i, j, m)$ @ u hold if thread i receives message m from thread j at time u.
- $\texttt{Enc}(k, m)$ and $\texttt{Dec}(k', m)$ denote encryption and decryption of message m with key k and k' respectively.
- The action new generates a new value, the predicate $\texttt{New}(i, n)$ @ u holds when thread i generates nonce n at time u.
- The action $\texttt{sig}(l, v)$ signs v with key l and returns $\texttt{Sig}(l, v)$ while $\texttt{verify}(l', v)$ verifies v of the form $\texttt{Sig}(l, v')$ and returns $true$ if the signature is valid.
- The action hash(m,m1) generates the hash of terms $(m, m1)$ and returns $\texttt{Hash}(m, m1)$.
- The action $\texttt{read}(r)$ reads a location r and the corresponding predicate $\texttt{Read}(i, r)$ @ u holds if thread i reads resource r at time u. The action $\texttt{write}(r, d)$ @ u writes value d to a resource r and the corresponding predicate $\texttt{Write}(i, r, d)$ @ u holds if thread i writes value d at location r at time u. For instance in our example, Server1 writes to mem_1.
- $\texttt{Mem}_1(t)$ @ u holds if mem_1 contains t at time u. $\texttt{Mem}_2(t)$ @ u holds if mem_2 contains t at time u.

Assumptions. (A1)

$$\texttt{HonestThread}(\mathsf{Server1}, \mathcal{N}(\mathsf{Server1}))$$

We are interested in security guarantees about honest users who create accounts by interacting with the server and who do not share the generated password or user-id with any other principal except for sending it according to the roles specified in the norms.

(A2)

$$\texttt{HonestThread}(\mathsf{User1}, \mathcal{N}(\mathsf{User1}))$$

(A3)

$$\texttt{HonestThread}(\mathsf{Adversary}, \mathcal{A}(\mathsf{Adversary}))$$

(A4)

$$\texttt{HonestThread}(\mathsf{Notary1}, \mathcal{A}(\mathsf{Notary1}))$$

(A5)

$$\texttt{HonestThread}(\mathsf{Notary2}, \mathcal{A}(\mathsf{Notary2}))$$

(A6)

$$\texttt{HonestThread}(\mathsf{Notary3}, \mathcal{A}(\mathsf{Notary3}))$$

(A7)

$$\texttt{HonestThread}(\mathsf{Notary4}, \mathcal{N}(\mathsf{Notary4}))$$

(A8)

$$\texttt{HonestThread}(\mathsf{Server2}, \mathcal{N}(\mathsf{Server2}))$$

(A9)

$$\texttt{HonestThread}(\mathsf{User2}, \mathcal{N}(\mathsf{User2}))$$

A principal following the protocol never shares its keys with any other entity. We also assume that the encryption scheme in semantically secure and non-malleable. Since we identify threads with principals therefore each of the threads are owned by principals with the same identifier, for instance Server1 owns the thread that executes the program N(Server1).

(Start1)

$$Start(i) \ @ \ -\infty$$

where i refers to all the threads in the set described above.

Security property. The security property of interest to us is that if at time u, a thread k is given access to account a, then k owns a. Specifically, in this example, we are interested in the $a = acct_1$ and $k = \mathsf{User1}$. This can be formalized by the following logical formula, $\neg\varphi_V$:

$$\forall u, k.\ (acct_1, k) \in P_1(u) \supset (k = \mathsf{User1}) \tag{C.1}$$

Here, $P_1(u)$ is the state of the access control matrix P_1 for Server1 at time u. (We use this logical formalization of the property in establishing the actual causes using our definition. Specifically,

Norm $\mathcal{N}$(Server1):

$1 : (uid1, n1) = \texttt{recv}(j)$; //access req from thread j
$2 : n2 = \texttt{new}$;
$3 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_\text{Server1}, (pub\_key\_\text{Server1}, n2, n1)))$;
$4 : s1 = \texttt{recv}(j)$; //encrypted $uid1, pwd1$ from j, alongwith its thread id J
$5 : (n3, uid1, pwd1, J) = \texttt{Dec}(pvt\_key\_\text{Server1}, s1)$;
$6 : t = \texttt{Hash}(uid1, pwd1)$;
$7 : \texttt{assert}(mem_1 = t)$ //compare hash with stored hash value for same uid
$8 : \texttt{insert}(P_1, (acct_1, J))$;

Norm $\mathcal{N}$(User1):

$1 : n1 = \texttt{new}$;
$2 : \texttt{send}(\text{Server1}, (uid1, n1))$; //access request
$3 : \texttt{Sig}(pub\_key1, n2, n1) = \texttt{recv}(j)$; //key from j
$4 : n3, n4, n5 = \texttt{new}$;
$5 : \texttt{send}(\text{Notary1}, pub\_key, n3)$;
$6 : \texttt{send}(\text{Notary2}, pub\_key, n4)$;
$7 : \texttt{send}(\text{Notary3}, pub\_key, n5)$;
$8 : \texttt{Sig}(pvt\_key\_\text{Notary1}, (pub\_key, l1, n3)) = \texttt{recv}(\text{Notary1})$; //notary1 responds
$9 : \texttt{Sig}(pvt\_key\_\text{Notary2}, (pub\_key, l2, n4)) = \texttt{recv}(\text{Notary2})$; //notary2 responds
$10 : \texttt{Sig}(pvt\_key\_\text{Notary3}, (pub\_key, l3, n5)) = \texttt{recv}(\text{Notary3})$; //notary3 responds
$11 : \texttt{assert}(\text{At least two of \{l1,l2,l3\} equal Server1})$
$12 : t = \texttt{Enc}(pub\_key, n2, (uid1, pwd1, \text{User1}))$;
$13 : \texttt{send}(\text{Server1}, t)$; //send t to Server1;

Norms $\mathcal{N}$(Notary1), $\mathcal{N}$(Notary2), $\mathcal{N}$(Notary3), $\mathcal{N}$(Notary4):

// o denotes Notary1, Notary2, Notary3 or Notary4
$1 : (pub\_key, n1) = \texttt{recv}(j)$;
$2 : pr = \texttt{KeyOwner}(pub\_key)$; //lookup key owner
$3 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_o, (pub\_key, pr, n1)))$; //signed certificate;

Figure C.1: Norms for Server1, User1, notaries. Adversary's norm is the trivial empty program.

we use a program logic to establish the sufficiency' and necessity conditions.)

C.B.2 Causal analysis of attack scenario

As an illustration, we model the violation in the "Compromising Notaries" attack of Section 5.1. In this attack scenario, User1 and Server1 execute norms. User1 sends an access request to Server1 which is intercepted by Adversary who sends its own key to User1 (pretending to be Server1). User1 checks with the three notaries who falsely verify Adversary's public key to be Server1's key. Consequently, User1 sends the password to Adversary. Adversary then initiates a protocol with Server1 and gains access to the User1's account. Note that the actual programs of the three notaries attest that the public key given to them belongs to Adversary. User2, Server2

Norm $\mathcal{N}$(Server2)**:**

$1 : (uid2, n1) = \texttt{recv}(j)$; //access req from thread j
$2 : n2 = \texttt{new}$;
$3 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_\text{Server2}, (pub\_key\_\text{Server2}, n2, n1)))$;
$4 : s1 = \texttt{recv}(j)$; //encrypted $uid2, pwd2$ from j, alongwith its thread id J
$5 : (n2, uid2, pwd2, J) = \texttt{Dec}(pvt\_key\_\text{Server2}, s1)$;
$6 : t = \texttt{Hash}(uid2, pwd2)$;
$7 : \texttt{assert}(mem_2 = t)$ //compare hash with stored hash value for same uid
$8 : \texttt{insert}(P_2, (acct_2, J))$;

Norm $\mathcal{N}$(User2)**:**

$1 : n1 = \texttt{new}$;
$2 : \texttt{send}(\text{Server2}, (uid2, n1))$; //access request
$3 : \texttt{Sig}(pub\_key, n2, n1) = \texttt{recv}(j)$; //key from j
$4 : n3 = \texttt{new}$;
$5 : \texttt{send}(\text{Notary4}, pub\_key, n3)$;
$6 : \texttt{Sig}(pvt\_key\_\text{Notary4}, (pub\_key, l1, n3)) = \texttt{recv}(\text{Notary4})$; //notary4 responds
$7 : \texttt{assert}(\{\text{l1}\} \text{ equals Server1})$
$8 : t = \texttt{Enc}(pub\_key, n2, (uid2, pwd2, \text{User2}))$;
$9 : \texttt{send}(\text{Server2}, t)$; //send t to Server2;

Figure C.2: Norms for Server2, User2

Actual program $\mathcal{A}$(Adversary)

$1 : (uid1, n1) = \texttt{recv}(j)$; //intercept req from User1
$2 : n2 = \texttt{new}$;
$3 : \texttt{send}(\text{User1}, pub\_key\_\text{Adversary})$; //send key to User1
$4 : s = \texttt{recv}(\text{User1})$; //pwd from User
$5 : n2, uid1, pwd1, \text{User1} = \texttt{Dec}(pvt\_key\_\text{Adversary}, s)$; //decrypt pwd;
$6 : n3 = \texttt{new}$;
$7 : \texttt{send}(\text{Server1}, (uid1, n3))$; //access request to Server
$8 : pub\_key, n4, n3 = \texttt{recv}(\text{Server1})$;
$9 : t = \texttt{Enc}(pub\_key, (n4, uid1, pwd1, \text{Adversary}))$; //encrypt pwd
$10 : \texttt{send}(\text{Server1}, t)$; //pwd to Server1

Actual programs $\mathcal{A}$(Notary1), $\mathcal{A}$(Notary2), $\mathcal{N}$(Notary3)**:**

// o denotes Notary1, Notary2 or Notary3
$1 : (pub\_key\_\text{Adversary}, n1) = \texttt{recv}(j)$;
$2 : \texttt{send}(j, \texttt{Sig}(pvt\_key\_o, (pub\_key\_\text{Adversary}, \text{Server1}, n1))$; //signed certificate to j;

Figure C.3: Deviants for Adversary and Notary1, Notary2, Notary3

Synchronization projection $l|_{\{\mathsf{Adversary},\mathsf{User1},\mathsf{Server1},\mathsf{Notary1},\mathsf{Notary2},\mathsf{Notary3}\}}$

$\langle\langle \mathsf{User1}, 2\rangle, \langle \mathsf{Adversary}, 1\rangle\rangle$,
$\langle\langle \mathsf{Adversary}, 3\rangle, \langle \mathsf{User1}, 3\rangle\rangle$
$\langle\langle \mathsf{User1}, 5\rangle, \langle \mathsf{Notary1}, 1\rangle\rangle$,
$\langle\langle \mathsf{User1}, 6\rangle, \langle \mathsf{Notary2}, 1\rangle\rangle$,
$\langle\langle \mathsf{User1}, 7\rangle, \langle \mathsf{Notary3}, 1\rangle\rangle$,
$\langle\langle \mathsf{Notary1}, 3\rangle, \langle \mathsf{User1}, 8\rangle\rangle$,
$\langle\langle \mathsf{Notary2}, 3\rangle, \langle \mathsf{User1}, 9\rangle\rangle$,
$\langle\langle \mathsf{Notary3}, 3\rangle, \langle \mathsf{User1}, 10\rangle\rangle$,
$\langle\langle \mathsf{User1}, 13\rangle, \langle \mathsf{Adversary}, 4\rangle\rangle$,
$\langle\langle \mathsf{Adversary}, 7\rangle, \langle \mathsf{Server1}, 1\rangle\rangle$,
$\langle\langle \mathsf{Server1}, 3\rangle, \langle \mathsf{Adversary}, 8\rangle\rangle$,
$\langle\langle \mathsf{Adversary}, 10\rangle, \langle \mathsf{Server1}, 4\rangle\rangle$

Synchronization projection $l|_{\{\mathsf{Server2},\mathsf{User2},\mathsf{Notary4}\}}$

$\langle\langle \mathsf{User2}, 1\rangle, \langle \mathsf{Server2}, 1\rangle\rangle$,
$\langle\langle \mathsf{Server2}, 3\rangle, \langle \mathsf{User2}, 3\rangle\rangle$,
$\langle\langle \mathsf{User2}, 5\rangle, \langle \mathsf{Notary4}, 1\rangle\rangle$,
$\langle\langle \mathsf{Notary4}, 3\rangle, \langle \mathsf{User2}, 6\rangle\rangle$,
$\langle\langle \mathsf{User2}, 9\rangle, \langle \mathsf{Server2}, 4\rangle\rangle$,

Figure C.4: Synchronization projections

and Notary4 execute their norms in order to access the account $acct_2$ as well.

In the property-violating trace, User1, Server1, User2, Server2 and Notary4 execute their norms and the expressions executed by Adversary and the three deviant notaries are shown in Figure C.3. Thus, the initial configuration has the programs:
$\{\mathcal{N}(\mathsf{User1}), \mathcal{N}(\mathsf{Server1}), \mathcal{A}(\mathsf{Adversary}), \mathcal{A}(\mathsf{Notary1}), \mathcal{A}(\mathsf{Notary2}), \mathcal{A}(\mathsf{Notary3}),$
$\mathcal{N}(\mathsf{User2}), \mathcal{N}(\mathsf{Server2}), \mathcal{N}(\mathsf{Notary4})\}$.

For this attack scenario, the concrete trace we consider is any *arbitrary interleaving* of the logs for $X = \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary2}, \mathsf{Notary3}\}$ and,
$Y = \{\mathsf{Server2}, \mathsf{User2}, \mathsf{Notary4}\}$ shown in Figure C.4. Any such interleaved log is denoted l in the sequel.

At the end of l, $(acct_1, \mathsf{Adversary})$ occurs in the access control matrix P_1, but Adversary does not own $acct_1$. Hence, this log corresponds to a violation of our security property. Importantly, even though only Adversary and three notaries deviate from their norms, this violation cannot happen without participation from User1 and Server1. Moreover, if any two of the three notaries had deviated from their norm, the violation would have still happened. Consequently, we may expect three independent causes in this example: {Adversary, User1, Server1, Notary1, Notary2}, {Adversary, User1, Server1, Notary1, Notary3}, and {Adversary, User1, Server1, Notary2, Notary3}. The following theorem states that our definitions determine exactly these three independent

causes.

Theorem 5. *Let* $I = \{\mathsf{User1}, \mathsf{Server1}, \mathsf{Adversary}, \mathsf{Notary1}, \mathsf{Notary2}, \mathsf{Notary3}, \mathsf{Notary4}, \mathsf{Server2}, \mathsf{User2}\}$*,* l *be a log defined above,* t *be a trace with* $log(t) = l$ *and* $\mathcal{A}$*,* $\mathcal{N}$ *and* Σ *be as described above. Let* $V = \langle I, \mathcal{A}, \Sigma, \mathcal{N}, \varphi_V, t\rangle$*. Then, Definition 30 determines three possible values for the actual cause* X_d *of violation:* $\{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary2}\}$*,* $\{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary3}\}$*, and* $\{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary2}, \mathsf{Notary3}\}$.

It is instructive to understand the proof of this theorem, as it illustrates our definitions of causation. We verify that V is a violation structure and that our Phase 1 and Phase 2 definitions yield exactly the three values for X_d mentioned in the theorem. To check that V is a violation structure, we must verify the occurrence and necessity conditions from Definition 26. For occurrence, we must show that $t \in \varphi_V$. This is clear because at the end of the trace, $(acct_1, \mathsf{Adversary}) \in P_1$, but $\mathsf{Adversary} \neq \mathsf{User1}$. For necessity, we show that any trace starting from $\langle I, \mathcal{N}, \Sigma\rangle$ cannot be in φ_V, i.e., any such trace has the invariant (C.1) [Security property]. To prove this, we can use any sound program logic. In particular, we use the logic of Garg *et al.* [93]. Our proof, in fact, shows the stronger property that (C.1) is an invariant even when the nine norms execute concurrently with any number of standard Dolev-Yao adversaries (we assume that encryption is non-malleable). This is just a standard proof of protocol correctness with a program logic.

Phase 1. The closure condition of Phase 1 (Definition 28) can be satisfied by the following four values of X: $X_1 = I$, $X_2 = \{\}$, $X_3 = \{\mathsf{Notary4}, \mathsf{Server2}, \mathsf{User2}\}$ and $X_4 = \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary2}, \mathsf{Notary3}\}$. When $X = X_2$ or $X = X_3$, $\mathsf{norm}(V, X, Y)$ contains *only* norms for all nine threads. We have already established in necessity that starting from all nine norms, we cannot obtain a violation, so sufficiency cannot be satisfied for these two values of X. That leaves only $X = X_1$ and $X = X_4$. We now show that $X = X_4$ satisfies sufficiency. (Since $X_4 \subseteq X_1$, minimality then implies that the X determined by Definition 28 must be X_4.) Following the statement of sufficiency, let T be the set of traces starting from $\mathsf{norm}(V, X_4, I \backslash X_4)$ that are log-consistent with l relative to X_4. Note that $\mathsf{norm}(V, X_4, I \backslash X_4)$ is exactly the same as the original initial configuration since threads in $I \backslash X_4 = \{\mathsf{Notary4}, \mathsf{Server2}, \mathsf{User2}\}$ were already executing norms. This implies that $t \in T$, so T is non-empty. If we pick any other $t' \in T$, then because the log of t' is consistent with l relative to X_4, that log must be exactly $l|_{\{\mathsf{Adversary},\mathsf{User1},\mathsf{Server1},\mathsf{Notary1},\mathsf{Notary2},\mathsf{Notary3}\}}$ from Figure C.4. This trivially implies that $t' \in \varphi_V$. Hence, Phase 1 determines $X = X_4 = \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary2}, \mathsf{Notary3}\}$.

Phase 2. Phase 2 (Definition 30) determines three independent actual causes for X_d: {Adversary, User1, Server1, Notary1, Notary2}, {Adversary, User1, Server1, Notary1, Notary3}, and {Adversary, User1, Server1, Notary2, Notary3}. These are symmetric, so we only explain why $X_d = \{\mathsf{Adversary}, \mathsf{User1}, \mathsf{Server1}, \mathsf{Notary1}, \mathsf{Notary2}\}$ satisfies Definition 30. We show that (a) This set X_d satisfies sufficiency', and (b) No proper subset of X_d satisfies sufficiency' (minimality').

We start with (a). By definition, $X_p = X \backslash X_d = \{\mathsf{Notary3}\}$. Fix any dummifying function f. We must show two things. First, there is a trace from $\mathsf{dummify}(V, X_d, X_p, f)$, whose log is consistent with l relative to X_d. Second, any such trace is in φ_V. The first statement is easily established by mimicking the execution in l starting from $\mathsf{dummify}(V, X_d, X_p, f)$. The only potential issue is that on line 7, $\mathcal{A}(\mathsf{User1})$ (which equals $\mathcal{N}(\mathsf{User1})$ from Figure C.1) synchronizes with Notary3 according to l, but Notary3 does not exist in $\mathsf{dummify}(V, X_d, X_p, f)$. However, in $\mathsf{dummify}(V, X_d, X_p, f)$, the `recv()` on line 10 in $\mathcal{A}(\mathsf{User1})$ is replaced with a dummy value, so the execution from $\mathsf{dummify}(V, X_d, X_p, f)$ progresses. Subsequently, the majority check on line 11 succeeds as in l, because two of three notaries (Notary1 and Notary2) still attest the Adversary's key.

The second statement — that every trace starting from $\mathsf{dummify}(V, X_d, X_p, f)$, whose log is consistent with l relative to X_d, is in φ_V — is the most non-obvious part of our proof. This statement quantifies over a select set of traces, but not all, so we cannot directly use the program logic. Instead, we use the program logic to establish certain invariants and combine those invariants with our knowledge of l. Fix a trace t' with log l'. Assume l' consistent with l relative to X_d. We show $t' \in \varphi_V$ as follows:

1. Since l' is consistent with l relative to X_d, Server1 must execute line 8 of its program $\mathcal{N}(\mathsf{Server1})$ in t'. After this line, the access control matrix P_1 contains $(acct_1, J)$ for some J.
2. When $\mathcal{N}(\mathsf{Server1})$ writes (x, J) to P_1 at line 8, then J is the third component of a tuple obtained by decrypting a message received on line 4.
3. Since l' is consistent with l relative to X_d, and on l $\langle \mathsf{Server1}, 4\rangle$ synchronizes with $\langle \mathsf{Adversary}, 10\rangle$, J must be the third component of an encrypted message sent on line 10 of $\mathcal{A}(\mathsf{Adversary})$.
4. The third component of the message sent on line 10 by Adversary is exactly the term "Adversary". (This is easy to see, as the term "Adversary" is hardcoded on line 9.) Hence, $J = \mathsf{Adversary}$.
5. This immediately implies that $t' \in \varphi_V$ since $(acct_1, \mathsf{Adversary}) \in P_1$, but $\mathsf{Adversary} \neq$

User1.

Last, we prove (b) — that no proper subset X'_d of X_d satisfies sufficiency'. Pick any $X'_d \subseteq X_d$ such that X'_d satisfies sufficiency'. We show that $X'_d = X_d$. Let $X'_p = X \backslash X'_d$. Observe that if Server1 $\notin X'_d$, then Server1 is not in dummify(V, X'_d, X'_p, f) and, hence, on any counterfactual trace cannot write to P, thus precluding a violation. Therefore, Server1 $\in X'_d$. By sufficiency', for any f, the log l' of at least one trace t' of dummify(V, X'_d, X'_p, f) must be consistent with l relative to X'_d. This means that in t', the assertion on line 7 of $\mathcal{N}$(Server1) must succeed and, hence, on line 5, the correct password $pwd1$ must be received by Server1, independent of f. This immediately implies that Adversary (which sends that password on l) must be in X'_d, else some dummified executions of Server1 will have the wrong password and the assertion on line 7 will fail.

Extending this logic further, we now observe that because Adversary forwards a password received from User1 (line 4 of $\mathcal{A}$(Adversary)) to Server1, User1 $\in X'_d$ (otherwise, some dummifications of line 4 of $\mathcal{A}$(Adversary) will result in the wrong password being sent to Server1, a contradiction). Since User1 $\in X'_d$ and l' must be consistent with l relative to X'_d, the majority check on line 11 of $\mathcal{N}$(User1) must also succeed. This means that at least two of {Notary1, Notary2, Notary3} must be in X'_d, else the dummification of lines 8 – 10 of $\mathcal{N}$(User1) will cause the assertion check to fail for some f. Since $X'_d \subseteq X_d$, these two threads must be {Notary1, Notary2}. At this point we have established that each of {Adversary, User1, Server1, Notary1, Notary2} is in X'_d. Hence, $X'_d = X_d$.

Violation Structure (Necessity). If all programs execute norms, there will be no violation on any resulting trace due to the correctness of the protocol[1] as described next:

1. We prove that if access permission $(acct_1, k)$ is added in P_1 for principal k, then $k =$ User1. Initially only User1 and Server1 know the password pwd_1 stored in Server1's memory mem_1.
2. Consider the traces where $(acct_1, k) \in P_1 \ @ \ u$ for some k. It is an (easy to prove) invariant that Server1 only adds access permission for $acct_1$ if a principal sends an access request with the correct password pwd_1 stored in mem_1. Therefore, Server1 must have received the correct password for pwd_1 from some thread j.
3. We prove an invariant of $\mathcal{N}$(User1) that if it sends out a password encrypted under a public key, then the public key was verified by the notaries to be Server1's public key. Similarly, we prove an invariant of the norms of the notaries that they only certify correct keys. Consequently, User1 only sends the password encrypted under Server1's public key.

[1]The proof technique for Necessity follows the proof of secrecy for Kerberos protocol by Garg et al [93].

4. It is an invariant of $\mathcal{N}(\mathsf{Server1})$ that it never sends any password.
5. Therefore, only User1 and Server1 ever see the password pwd_1.
6. Hence, the password received by Server1 in step (2) must have been sent by $j = \mathsf{User1}$.
7. It is an invariant of $\mathcal{N}(\mathsf{User1})$ that if it sends a request to add an access permission (line 8), it does so for itself. Combining with (2), we deduce that $k = \mathsf{User1}$.

Expanding the steps. We adapt the definitions from the proof of secrecy for Kerberos by Garg et al [1] for a framework based on asymmetric encryption. Some of these definitions are given in Figure C.5. For more details, we refer the reader to [1]. Here $\mathcal{K}$ denotes a set of keys. $\texttt{OwnedIn}(i, \mathcal{S})$ means that thread i in in set $\mathcal{S}$, $\texttt{OrigRes}(s, \mathcal{S})$ means that the thread which created the term s lies in set $\mathcal{S}$, $\texttt{KeyRes}(\mathcal{K}, S)$ means that the inverse key corresponding to each key in set $\mathcal{K}$ is known only to threads in $\mathcal{S}$. $\texttt{KORes}(s, \mathcal{K}, u)$ combines the previous two predicates, $\texttt{SendsSafeMsg}(i, s, \mathcal{K})$ means that if thread i sends s in a message, then all occurrences of s in that message are protected by keys in $\texttt{Inv}(\mathcal{K})$ i.e. the private keys corresponding to the public keys in $\mathcal{K}$. $\texttt{SafeNet}(s, \mathcal{K}, u)$ means that prior to time u, every thread protects s in all messages it sends using keys in $\texttt{Inv}(\mathcal{K})$.

Figure C.5 also lists two axioms (POS) and (NET) from prior work. We introduce an additional axiom called (NM). This axiom (assuming non-malleability) states that if only principals in set $\mathcal{P}$ had access to the corresponding private keys for set $\mathcal{K}$ and have knowledge of terms in s, then if a message m containing secret s is encrypted under a key $k \in \mathcal{K}$ and sent at time u, then at a prior time u' some principal in $\mathcal{P}$ must have sent the message on the network.

We prove that if access permission $(acct_1, k)$ is added in P_1 for principal k, then $k = \mathsf{User1}$.

1. Consider the traces where $(acct_1, k) \in P_1 \ @ \ u$ for some k. We will first prove that Server1 only adds access permission for $acct_1$ if a principal sends an access request with the correct password pwd_1 stored in mem_1. Therefore, Server1 must have received the correct password for $acct_1$ from some thread j at time $u_1 < u$.

   Invariant of Server1:

$$\{\mathcal{N}(\mathsf{Server1})\}\langle u_b, u_e, i\rangle \forall u, k.((u_b < u \leq u_e) \wedge \texttt{Insert}(i, (acct_1, k)) \ @ \ u)) \supset$$
$$(\exists j, k, n1, pwd1, u_1.$$
$$(u_1 < u) \wedge \texttt{Mem}_1(\texttt{Hash}(uid1, pwd1)) \ @ \ u \wedge$$
$$(\texttt{Recv}(\mathsf{Server1}, j, (\texttt{Enc}(pub\_key\_\mathsf{Server1}, (n1, uid1, pwd1, k)))) \ @ \ u_1)$$

Definitions

$$\texttt{OwnedIn}(i, \mathcal{S}) = i \in \mathcal{S}$$
$$\texttt{OrigRes}(s, \mathcal{S}) = \forall i, u.\ \texttt{New}(i, s)\ @\ u \supset \texttt{OwnedIn}(i, \mathcal{S})$$
$$\texttt{KeyRes}(\mathcal{K}, \mathcal{S}) = \forall i, k.\ (\texttt{Has}(i, \texttt{Inv}(k)) \wedge k \in \mathcal{K}) \supset \texttt{OwnedIn}(i, \mathcal{S})$$
$$\texttt{KORes}(s, \mathcal{K}, \mathcal{S}) = \texttt{OrigRes}(s, \mathcal{S}) \wedge \forall u.\ \texttt{KeyRes}(\mathcal{K}, \mathcal{S})\ @\ u$$
$$\texttt{SendsSafeMsg}(i, s, \mathcal{K}) = \texttt{Send}(i, v) \supset \texttt{SafeMsg}(v, s, \mathcal{K})$$
$$\texttt{SafeNet}(s, \mathcal{K}, u) = \forall i, u'.\ (u' \leq u) \supset \texttt{SendsSafeMsg}(i, s, \mathcal{K})\ @\ u'$$
$$\texttt{HasOnly}(\mathcal{S}, s) = \forall i.\ \texttt{Has}(i, s) \supset \texttt{OwnedIn}(i, \mathcal{S})$$

Additional Axioms

(NET)
$$(\texttt{KORes}(s, \mathcal{K}, \mathcal{S}) \wedge \texttt{SafeNet}(s, \mathcal{K}, u_1) \wedge \neg\texttt{SafeNet}(s, \mathcal{K}, u_2) \wedge (u_1 < u_2)) \supset \exists i, u_3.\ (u_1 < u_3 \leq u_2) \wedge \texttt{OwnedIn}(i, \mathcal{S}) \wedge \neg\texttt{SendsSafeMsg}(i, s, \mathcal{K})\ @\ u_3 \wedge \forall u_4 \in (u_1, u_3).\ \texttt{SafeNet}(s, \mathcal{K}, u_4)$$

(POS)
$$(\texttt{SafeNet}(s, \mathcal{K}, u) \wedge \texttt{Has}(i, s)\ @\ u) \supset (\exists u'.\ (u' < u) \wedge \texttt{New}(i, s)\ @\ u') \vee (\exists k.\ (k \in \mathcal{K}) \wedge \texttt{Has}(i, \texttt{Inv}(k))\ @\ u)$$

(NM)
$$(\texttt{HasOnly}(\texttt{Inv}(\mathcal{K}), \mathcal{P})\ @\ u \wedge \texttt{HasOnly}(s, \mathcal{P})\ @\ u \wedge \exists i, i1.\ \texttt{Send}(i, i1, m')\ @\ u \wedge m' = \texttt{Enc}(k, m) \wedge (k \in \mathcal{K}) \wedge \texttt{Contains}(m, s)) \supset (\exists u', j, j1.\ (u' < u) \wedge j \in \mathcal{P} \wedge (\texttt{Send}(j, j1, m)\ @\ u'))$$

Figure C.5: Additional definitions and axioms (Garg et al [1])

which we abbreviate as:

$$\{\mathcal{N}(\mathsf{Server1})\}\langle u_b, u_e, i\rangle \forall u, k.((u_b < u \leq u_e) \wedge \texttt{Insert}(i, (acct_1, k)) \mathbin{@} u)) \supset \varphi$$

Using (HONTH) in conjunction with 1, assuming (Start1), we get:

$$\forall u'.\ (u' > -\infty) \supset \\ \forall u, k.((u_b < u \leq u_e) \wedge \texttt{Insert}(\mathsf{Server1}, (acct_1, k)) \mathbin{@} u)) \supset \varphi$$

Choosing $u_e = \infty$, we get

$$\forall u, k.\texttt{Insert}(\mathsf{Server1}, (acct_1, k)) \mathbin{@} u) \supset \varphi$$

We have previously assumed that Server1 inserts permission for principal k, therefore the consequent of the above implication holds. This implies that at some time $u_1 < u$, Server1 received the correct password pwd_1 for $acct_1$ from some thread j. In the sequel we use pwd_1 to denote the password stored in mem_1.

2. Initially only User1 and Server1 know the password pwd_1 stored in Server1's memory mem_1:

   $$\texttt{HasOnly}(\{\mathsf{Server1}, \mathsf{User1}\}, pwd_1) \mathbin{@} -\infty$$

   We also assume that $\forall u.\ \texttt{HasOnly}(\mathsf{Server1}, pvt\_key\_\mathsf{Server1}) \mathbin{@} u$.

3. Next we prove that only User1 and Server1 ever see the password pwd_1. First, we prove that $\texttt{SendsSafeMsg}(\mathsf{User1}, pwd_1, pub\_key\_\mathsf{Server1}) \mathbin{@} u$ for every u. We prove an invariant of $\mathcal{N}(\mathsf{User1})$ that if it sends out a password encrypted under a public key, then the public key was verified by the notaries to be Server1's public key. Note that User1 only sends out the password in line 13 of the code $\mathcal{N}(\mathsf{User1})$.

   Invariant of User1:

   $$\{\mathcal{N}(\mathsf{User1})\}\langle u_b, u_e, i\rangle\ \forall u, j1, n1.((u_b < u \leq u_e) \wedge \\ \texttt{Send}(\mathsf{User1}, j1, (\texttt{Enc}(pub\_key\_j1, (n1, uid1, pwd_1, \mathsf{User1})))) \mathbin{@} u \supset \\ (\exists n2, u_1.(u_1 < u) \wedge \\ \texttt{Recv}(\mathsf{User1}, \mathsf{Notary1}, \texttt{Sig}(pvt\_key\_\mathsf{Notary1}, (pub\_key\_j1, \mathsf{Server1}, n2))) \mathbin{@} u_1)$$

   We abbreviate the consequent of the above implication as $\varphi 2$.

Using (HONTH) in conjunction with 3 above, assuming (Start1), we get:

$$\forall u'.\ (u' > -\infty) \supset \forall u, j1, n1, pwd1.$$
$$((u_b < u \leq u_e) \wedge \mathtt{Send}(\mathsf{User1}, j1, (\mathtt{Enc}(pub\_key\_j1, (n1, uid1, pwd1, \mathsf{User1})))) \ @\ u$$
$$\supset \varphi 2$$

Choosing $u_e = \infty$, we get

$$\forall u, j1, n1, pwd1.\ \mathtt{Send}(\mathsf{User1}, j1, (\mathtt{Enc}(pub\_key\_j1, (n1, uid1, pwd1, \mathsf{User1}))))$$
$$@\ u \supset \varphi 2$$

4. Similarly, we prove an invariant of the norms of the notaries that they only certify correct keys. We assume that the notary repository is correct. By analyzing the notaries' threads $\mathcal{N}(\mathsf{Notary1}), \mathcal{N}(\mathsf{Notary2}), \mathcal{N}(\mathsf{Notary3}), \mathcal{N}(\mathsf{Notary4})$, we prove that if the notaries sign a key stating that it belongs to Server1 (or Server2), the key must belong to Server1 (or Server2 respectively). We show the invariant for one of the notaries, Notary1 which we abbreviate as:

$$\{\mathcal{N}(\mathsf{Notary1})\}\langle u_b, u_e, i\rangle \forall u, j1, k, pwd1.$$
$$(u_b < u \leq u_e) \wedge \mathtt{Send}(\mathsf{Notary1}, j1, (\mathtt{Sig}(pub\_key\_\mathsf{Notary1}, (pub\_key\_k, \mathsf{Server1}, n1))))$$
$$@\ u \supset pub\_key\_k = pub\_key\_\mathsf{Server1}$$

Consequently, User1 only sends the password encrypted under Server1's public key. This implies that User1 only sends safe messages containing pwd_1 and encrypted under Server1's public key.

5. Next, it is trivial to prove that $\mathtt{SendsSafeMsg}(\mathsf{Server1}, pwd_1, pub\_key\_\mathsf{Server1})\ @\ u$ for every u. This is because Server1 never sends out the password in any message.

6. Next we apply rely-guarantee reasoning similar to the secrecy proof for Kerberos by Garg et al [1] in order to show $\forall u.\varphi(u)$ where

$$\varphi(u)\ =\ \forall u.\ \mathtt{SafeNet}(pwd_1, pub\_key\_\mathsf{Server1}, u)$$

We instantiate the framework of rely-guarantee by choosing:

$$\iota(i) = (i = \mathsf{Server1}) \wedge (i = \mathsf{User1})$$

$$\psi(u, i) = \mathtt{SendsSafeMsg}(i, pwd_1, pub\_key\_\mathsf{Server1})$$

$$\varphi(u) = \texttt{SafeNet}(pwd_1, pub\_key\_\mathsf{Server1}, u)$$

In order to apply the method of rely-guarantee, we must show that the following hold for φ, ι, and ψ as defined above:

(1) $\varphi(-\infty)$

(2) $\forall i, u.\ (\iota(i) \wedge \forall u' < u.\ \varphi(u')) \supset \psi(u, i)$

(3) $(\varphi(u_1) \wedge \neg\varphi(u_2) \wedge (u_1 < u_2)) \supset$
$\exists i, u_3.\ (u_1 < u_3 \leq u_2) \wedge \iota(i) \wedge \neg\psi(u_3, i) \wedge$
$\forall u_4 \in (u_1, u_3).\ \varphi(u_4)$

(2) follows from steps 4 and 5 above.

To prove (3), we instantiate the axiom (NET) by choosing $s = pwd_1, \mathcal{K} = \{pub\_key\_\mathsf{Server1}\}$, and $\mathcal{S}_0 = \{\mathsf{Server1}, \mathsf{User1}\}$ to obtain:

$$\begin{aligned}
&(\texttt{KORes}(pwd_1, pub_key_\mathsf{Server1}, \mathcal{S}_0) \wedge \texttt{SafeNet}(pwd_1, pub_key_\mathsf{Server1}, u_1) \wedge \\
&\neg\texttt{SafeNet}(pwd_1, pub_key_\mathsf{Server1}, u_2) \wedge (u_1 < u_2)) \supset \\
&\quad \exists i, u_3.\ (u_1 < u_3 \leq u_2) \wedge \texttt{OwnedIn}(i, \mathcal{S}_0) \wedge \\
&\qquad \neg\texttt{SendsSafeMsg}(i, \mathcal{S}_0, pub_key_\mathsf{Server1}) \ @\ u_3 \wedge \\
&\qquad \forall u_4 \in (u_1, u_3).\ \texttt{SafeNet}(pwd_1, pub_key_\mathsf{Server1}, u_4)
\end{aligned}$$

We show that $\texttt{KORes}(pwd_1, pub\_key\_\mathsf{Server1}, \{\mathsf{User1}, \mathsf{Server1}\})$. Expanding the definition of KORes, we need to show that pwd_1 was generated by either User1 or Server1 (which is true by assumption since User1 generated pwd_1) and that pvt_key_Server1 is known only to {User1, Server1} (assumption). Therefore, $\texttt{KORes}(pwd_1, pub\_key\_\mathsf{Server1}, \mathcal{S}_0)$ holds and we eliminate that condition from the above formula to obtain:

$$\begin{aligned}
&(\texttt{SafeNet}(pwd_1, pub_key_\mathsf{Server1}, u_1) \wedge \neg\texttt{SafeNet}(pwd_1, pub_key_\mathsf{Server1}, u_2) \wedge \\
&(u_1 < u_2)) \supset \\
&\quad \exists i, u_3.\ (u_1 < u_3 \leq u_2) \wedge \texttt{OwnedIn}(i, \mathcal{S}_0) \wedge \neg\texttt{SendsSafeMsg}(i, \mathcal{S}_0, pub_key_\mathsf{Server1}) \ @\ u_3 \wedge \\
&\qquad \forall u_4 \in (u_1, u_3).\ \texttt{SafeNet}(pwd_1, pub_key_\mathsf{Server1}, u_4)
\end{aligned}$$

This proves the statement of (3) above. Hence, we deduce that $\forall u.\ \varphi(u)$, i.e,

$$\forall u.\ \texttt{SafeNet}(pwd_1, pub\_key\_\mathsf{Server1}, u) \tag{C.2}$$

Next, we fix any time parameter u_0', and try to show that $\texttt{HasOnly}(\mathcal{S}_0, pwd_1)\ @\ u_0'$. Following the definition of HasOnly assume that for some i, $\texttt{Has}(i, pwd_1)\ @\ u_0'$. It suffices

to show that $\texttt{OwnedIn}(i, \mathcal{S}_0)$. From (C.2) above, the assumption $\texttt{Has}(i, pwd_1)$ @ u'_0, and axiom (POS), we obtain:

$$(\exists u'.\ (u' < u'_0) \wedge \texttt{New}(i, pwd_1) \mathbin{@} u') \vee$$
$$(\exists k.\ (k \in \{pub\_key\_\mathsf{Server1}\}) \wedge \texttt{Has}(i, \texttt{Inv}(k)) \mathbin{@} u'_0)$$

We case analyze these two disjuncts. If $\exists u'.\ (u' < u'_0) \wedge \texttt{New}(i, pwd_1)$ @ u', then we obtain $i = \mathsf{User1}$. Since $\mathcal{S}_0 = \{\mathsf{User1}, \mathsf{Server1}\}$, $\texttt{OwnedIn}(i, \mathcal{S}_0)$ follows from definition of OwnedIn.

If $\exists k.\ (k \in \{pub\_key\_\mathsf{Server1}\}) \wedge \texttt{Has}(i, \texttt{Inv}(k))$ @ u'_0, then from the assumptions that User1 generated pwd_1 and Server1's public key is only known to Server1, i.e, $\texttt{KORes}(pwd_1, pub\_key\_\mathsf{Server1}, \mathcal{S}_0)$, we immediately obtain $\texttt{OwnedIn}(i, \mathcal{S}_0)$.

Since, in both case analyses we obtain $\texttt{OwnedIn}(i, \mathcal{S}_0)$, it follows that $\texttt{HasOnly}(\mathcal{S}_0, pwd_1)$ @ u'_0 for any u'_0. Since u'_0 is a parameter, this implies $\forall u'.\ \texttt{HasOnly}(\mathcal{S}_0, pwd_1)$ @ u', which is the property we wanted to prove, i.e.

$$\forall u.\ \texttt{HasOnly}(\{\mathsf{Server1}, \mathsf{User1}\}, pwd_1) \mathbin{@} u$$

7. From 6. above, pwd_1 is only known to $\mathcal{S}_0$. By assumption,

$$\forall u.\ \texttt{HasOnly}(\{\mathsf{Server1}\}, pvt\_key\_\mathsf{Server1}) \mathbin{@} u$$

Also initially we showed that a message was sent to Server1 which contained the password encrypted under Server1's public key. Instantiating the antecedent in (NM) with $\mathcal{P} = \mathcal{S}_0$, $s = pwd_1$, $\mathcal{K} = pub\_key\_\mathsf{Server1}$, we can infer that either User1 or Server1 sent the initial message containing the password encrypted under Server1's key.

8. It is an invariant of $\mathcal{N}(\mathsf{Server1})$ that it never sends any password.
9. It is an invariant of $\mathcal{N}(\mathsf{User1})$ that if it sends a request to add an access permission (line 8), it does so for itself. Invariant of User1:

$$\{\mathcal{N}(\mathsf{User1})\}\langle u_b, u_e, i\rangle \forall u, j1, n1.((u_b < u \leq u_e) \wedge$$
$$\texttt{Send}(\mathsf{User1}, j1, (\texttt{Enc}(pub\_key\_j1, (n1, uid1, pwd_1, k)))) \mathbin{@} u \supset$$
$$(k = \mathsf{User1})$$

Combining with (2), we deduce that $k = \mathsf{User1}$. Therefore, we have prove that if access permission $(acct_1, k)$ is added in P_1 for principal k, then $k = \mathsf{User1}$.

Bibliography

[1] D. Garg, J. Franklin, D. Kaynar, and A. Datta, "Compositional system security with interface-confined adversaries (technical report)," Carnegie Mellon University, Tech. Rep. CMU-CyLab-10-004, Feb. 2010. [Online]. Available: http://www.mpi-sws.org/~dg/papers/cmu-cylab-10-004.pdf (document), C.B.2, C.5, 6

[2] C. Kaufman, R. Perlman, and M. Speciner, *Network security: private communication in a public world*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995. 1.1.1

[3] R. L. Rivest and W. D. Smith, "Three voting protocols: Threeballot, vav, and twin," in *Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, ser. EVT'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 16–16. 1.1.1

[4] R. Küsters, T. Truderung, and A. Vogt, "Accountabiliy: Definition and Relationship to Verifiability," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*. ACM Press, 2010, pp. 526–535. 1.1.1, 1.1.2, 6.3, 6.3.1, 7.1.1

[5] D. C. Parkes, M. O. Rabin, S. M. Shieber, and C. Thorpe, "Practical secrecy-preserving, verifiably correct and trustworthy auctions," *Electron. Commer. Rec. Appl.*, vol. 7, no. 3, pp. 294–312, Nov. 2008. 1.1.1

[6] M. Backes, A. Datta, A. Derek, J. C. Mitchell, and M. Turuani, "Compositional analysis of contract-signing protocols," *Theor. Comput. Sci.*, vol. 367, no. 1-2, pp. 33–56, 2006. 1.1.1, 1.1.2, 6.3, 6.3.1

[7] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, ser. STOC '87. New York, NY, USA: ACM, 1987, pp. 218–229. 1.1.1

[8] H. Nissenbaum, "Accountability in a computerized society," *Science and Engineering Ethics*, vol. 2, no. 1, pp. 25–42, 1996. 1.1.1, 5.4.2

[9] B. Lampson, "Computer security in the real world," *Computer*, vol. 37, no. 6, pp. 37 – 46, June 2004. 1.1.1

[10] A. Haeberlen, P. Kouznetsov, and P. Druschel, "Peerreview: practical accountability for distributed systems," in *SOSP*, 2007, pp. 175–188. 1.1.1, 1.1.2, 6.3, 6.3.1

[11] D. J. Weitzner, H. Abelson, T. Berners-Lee, J. Feigenbaum, J. Hendler, and G. J. Sussman, "Information accountability," *Commun. ACM*, vol. 51, no. 6, pp. 82–87, Jun. 2008. 1.1.1

[12] R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely, "Towards a theory of accountability and audit," in *ESORICS*, 2009, pp. 152–167. 1.1.1, 1.1.2, 6.3, 6.3.1

[13] J. Gatins, "Flight," 2012. [Online]. Available: http://www.imdb.com/title/tt1907668/ 1.1.2, 5

[14] J. Y. Halpern and J. Pearl, "Causes and explanations: A structural-model approach. part i: Causes," *British Journal for the Philosophy of Science*, vol. 56, no. 4, pp. 843–887, 2005. 1.1.2, 1.3.1, 1a, 13, 14, 2, 2.3.1, 2.4, 2.5, 2.5, 3, 4.2, 4.2.2, 4.2.3, 10, 18, 4.4.3, 5, 6.3, 6.3.1, 7.2, C.A

[15] J. Y. Halpern, "Defaults and Normality in Causal Structures," *Artificial Intelligence*, vol. 30, pp. 198–208, 2008. [Online]. Available: http://arxiv.org/abs/0806.2140 1.1.2, 9, C.A

[16] N. Hall, "Structural equations and causation," *Philosophical Studies*, vol. 132, no. 1, pp. 109–136, 2007. 1.1.2, 1.3.1, 2, 2.3.1, 2.4, 4.2, 4.2.2, 4.2.5.2, 4.4.2, 7.2

[17] C. Hitchcock, "The intransitivity of causation revealed in equations and graphs," *Journal of Philosophy*, vol. 98, no. 6, pp. 273–299, 2001. 1.1.2, 1.3.1, 1a, 13, 14, 2, 2.3.1, 4.2, 4.2.1, 7, 5, 4.2.5.2, 7.2

[18] J. Y. Halpern, "A modification of the halpern-pearl definition of causality," *CoRR*, vol. abs/1505.00162, 2015. [Online]. Available: http://arxiv.org/abs/1505.00162 1.1.2, 13, 14, 2, 4.2, 4.2.1, 4.2.2, 4.2.4

[19] J. Feigenbaum, A. D. Jaggard, and R. N. Wright, "Towards a formal model of accountability," in *Proceedings of the 2011 workshop on New security paradigms workshop*, ser. NSPW '11. New York, NY, USA: ACM, 2011, pp. 45–56. 1.1.2, 6.3, 6.3.1

[20] J. Feigenbaum, J. A. Hendler, A. D. Jaggard, D. J. Weitzner, and R. N. Wright, "Accountability and deterrence in online life," in *Proceedings of the 3rd International Web Science Conference*, ser. WebSci '11. NY, USA: ACM, 2011, pp. 7:1–7:7. 1.1.2, 6.3, 6.3.1

[21] M. Bellare, D. Micciancio, and B. Warinschi, "Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions," in *EUROCRYPT*, 2003, pp. 614–629. 1.1.2

[22] G. Gössler, D. Le Métayer, and J.-B. Raclet, "Causality analysis in contract violation," in *Proceedings of the First International Conference on Runtime Verification*, ser. RV'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 270–284. 1.1.2, 6.3, 6.3.2

[23] S. Wang, A. Ayoub, R. Ivanov, O. Sokolsky, and I. Lee, "Contract-based blame assignment by trace analysis," in *Proceedings of the 2nd ACM International Conference on High Confidence Networked Systems*. NY, USA: ACM, 2013. 1.1.2, 6.3, 6.3.2

[24] R. Milner, "What is a process?" [Online]. Available: http://www.cs.rice.edu/~vardi/papers/milner09.pdf 1.2

[25] ——, *A Calculus of Communicating Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1982. 1.2.1

[26] P. Spirtes, C. N. Glymour, and R. Scheines, *Causation, prediction, and search*. MIT press, 2000, vol. 81. 1.2.1, 1.3.1, 2.3, 4, 2, 4

[27] J. Pearl, *Causality: models, reasoning, and inference*. New York, NY, USA: Cambridge University Press, 2000. 1.2.1, 1.3.1, 1a, 1.5, 2.3, 4, 4, 4.2, 4.3.3, 20, 5, 6.3

[28] C. Hitchcock, "Probabilistic causation," in *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed., 2012. 1.2.1, 4, 3, 5

[29] J. Livengood, "On causal inferences in the humanities and social sciences," Ph.D. dissertation, University of Pittsburgh, 2011. 1.2.1, 2, 2.3, 2.3.1, 3, 4, 4.2, 4.2.3, 4.3.4

[30] J. Y. Halpern and J. Pearl, "Causes and explanations: a structural-model approach: part i: causes," in *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, ser. UAI'01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 194–202. 1.3.1, 1a, 13, 14, 4.2, 4.2.3, 5, 6.3, 6.3.1

[31] ——, "Causes and explanations: A structural-model approach. part ii: Explanations," *The British Journal for the Philosophy of Science*, vol. 56, no. 4, pp. 889–911, 2005. 1.3.1, 6.1

[32] M. S. Moore, *Causation and Responsibility: An Essay in Law, Morals and Metaphysics*. Oxford University Press, 2009. 1.3.1

[33] H. L. A. Hart and T. Honoré, *Causation in the Law (Second Edition)*. Oxford, UK: Oxford University Press, 1985. 1.3.1, 4.2

[34] K. Shaver, *The attribution of blame: Causality, responsibility, and blameworthiness*. Springer Science & Business Media, 2012. 1.3.1, 6.2

[35] M. D. Alicke, "Culpable control and the psychology of blame." *Psychological bulletin*, vol. 126, no. 4, p. 556, 2000. 1.3.1, 6.2

[36] J. Feinberg, "Ethical issues in the use of computers," D. G. Johnson and J. W. Snapper, Eds. Belmont, CA, USA: Wadsworth Publ. Co., 1985, ch. Sua Culpa, pp. 102–120. [Online]. Available: http://dl.acm.org/citation.cfm?id=2569.2675 1.3.1, 5.4.2, 6.2

[37] L. Kenner, "On blaming," *Mind*, pp. 238–249, 1967. 1.3.1, 6.2

[38] D. Hume, "An Enquiry Concerning Human Understanding," *Reprinted Open Court Press, LaSalle, IL, 1958*, 1748. 1.3.1, 4, 4.2

[39] D. Lewis, "Causation," *Journal of Philosophy*, vol. 70, no. 17, pp. 556–567, 1973. 1.3.1, 4.2

[40] J. Collins, N. Hall, and L. A. Paul, *Causation and Counterfactuals*. MIT Press, 2004. 1.3.1, 4.2, 6.3

[41] J. L. Mackie, "Causes and Conditions," *American Philosophical Quarterly*, vol. 2, no. 4, pp. 245–264, 1965. 1.3.1, 4.2, 6.3

[42] R. Wright, "Causation in tort law," *California Law Review 73*, pp. 1735–1828, 1985. 1.3.1, 1.5, 4.2, 9, 6.3

[43] N. Hall, "Two concepts of causation," in *Causation and Counterfactuals*, J. Collins, N. Hall, and L. Paul, Eds. The Mit Press, 2004, pp. 225–276. 9, 10

[44] P. Dowe, "Causal Process Theories," in *The Oxford Handbook of Causation*, Helen Beebee, Christopher Hitchcock, and Peter Menzies, Eds. OUP Oxford, 2009. 1.3.2, 4.1, 7.2

[45] W. C. Salmon, "Causality without counterfactuals," *Philosophy of Science*, vol. 61, no. 2, pp. pp. 297–312, 1994. [Online]. Available: http://www.jstor.org/stable/188214 1.3.2, 4.1, 7.2

[46] C. Hitchcock, "Causal processes and interactions: What are they and what are they good for?" *Philosophy of Science*, vol. 71, no. 5, pp. 932–941, 2004. 1.3.2, 4.1, 7.2

[47] D. Lewis, "Events," in *Philosophical Papers Vol. II.* OUP, 1986, vol. 2, pp. 241–269. 12, 6, 4.4.3, 20

[48] M. Hopkins, "Strategies for determining causes of events," in *AAAI/IAAI*, 2002, pp. 546–552. 1a, 2.1, 2.3.1, 2.5, 4.2.3

[49] M. McDermott, "Redundant causation," *British Journal for the Philosophy of Science*, vol. 46, no. 4, pp. 523–544, 1995. 1c, 4.3.3, 6, 4.4.3, 20

[50] J. Clark and P. C. van Oorschot, "Sok: Ssl and https: Revisiting past challenges and evaluating certificate trust model enhancements," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 511–525. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.41 2, 6.2

[51] H. A. Simon, *The Sciences of the Artificial*, 1st ed. Cambridge, Massachusetts: MIT Press, 1969. 1.5

[52] H. Chockler and J. Y. Halpern, "Responsibility and blame: A structural-model approach,"

Journal of Artificial Intelligence Research, pp. 93–115, 2004. 1.5

[53] G. Gössler and L. Aştefănoaei, "Blaming in component-based real-time systems," in *Proceedings of the 14th International Conference on Embedded Software*, ser. EMSOFT '14. ACM, 2014, pp. 7:1–7:10. [Online]. Available: http://doi.acm.org/10.1145/2656045.2656048 1.5, 6.3.2

[54] A. Datta, D. Garg, D. Kaynar, D. Sharma, and A. Sinha, "Program actions as actual causes: A building block for accountability," in *Proceedings of the 2015 IEEE 28th Computer Security Foundations Symposium*, ser. CSF '15, 2015. 1.6

[55] A. Datta, D. Garg, D. K. Kaynar, D. Sharma, and A. Sinha, "Program actions as actual causes: A building block for accountability," *CoRR*, vol. abs/1505.01131, 2015. [Online]. Available: http://arxiv.org/abs/1505.01131 1.6

[56] B. Weslake, "A partial theory of actual causation," *British Journal for the Philosophy of Science*, p. To appear, 2015. 2, 2.3, 4.2, 4.2.3, 4.3.3, 6

[57] J. Schaffer, "Overdetermining causes," *Philosophical Studies*, vol. 114, no. 1-2, pp. 23–45, 2003. 2.1.1, 4.4.3, 8

[58] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen, "The essence of compiling with continuations," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, ser. PLDI '93. New York, NY, USA: ACM, 1993, pp. 237–247. [Online]. Available: http://doi.acm.org/10.1145/155090.155113 2.1.1, 5.2.1, A

[59] L. Lamport, "Ti clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, pp. 558–565, July 1978. 2.2, 5.2.2, 5.2.3, C.A.2.2

[60] R. W. Wright, "Causation, responsibility, risk, probability, naked statistics, and proof: Pruning the bramble bush by clarifying the concepts," *Iowa L. Rev.*, vol. 73, p. 1001, 1987. 2.5

[61] C. Glymour, D. Danks, B. Glymour, F. Eberhardt, J. Ramsey, R. Scheines, P. Spirtes, C. Teng, and J. Zhang, "Actual causation: a stone soup essay," *Synthese*, vol. 175, no. 2, pp. 169–192, 2010. [Online]. Available: http://dx.doi.org/10.1007/s11229-009-9497-9 3, 4.2, 4.3.4

[62] M. Tooley, *Causation: Fundamental Issues*. (Manuscript in progress). 4

[63] J. Woodward, *Making things happen: A theory of causal explanation*. Oxford University Press, 2003. 4, 1, 6.1

[64] E. Eells, *Probabilistic Causality*. CUP, 1991, no. 3. 5

[65] N. Lynch, R. Segala, and F. Vaandrager, "Hybrid i/o automata," *Information and Computation*, vol. 185, no. 1, pp. 105 – 157, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0890540103000671 4.1

[66] T. Henzinger, "The theory of hybrid automata," in *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*, Jul 1996, pp. 278–292. 4.1

[67] E. Brinksma, T. Krilavicius, and Y. S. Usenko, "A process-algebraic approach to hybrid systems," 2005. 4.1

[68] J. Y. Halpern, "Defaults and Normality in Causal Structures," *Artificial Intelligence*, vol. 30, pp. 198–208, 2008. [Online]. Available: http://arxiv.org/abs/0806.2140 4.2, 4.2.5.2, 7.2

[69] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler, "Explaining counterexamples using causality," in *Proceedings of the 21st International Conference on Computer Aided Verification*, ser. CAV '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 94–108. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02658-4_11 4.2.3, 6.1

[70] H. Chockler, J. Y. Halpern, and O. Kupferman, "What causes a system to satisfy a specification?" *ACM Trans. Comput. Logic*, vol. 9, pp. 20:1–20:26, June 2008. [Online]. Available: http://doi.acm.org/10.1145/1352582.1352588 4.2.3, 6.1

[71] D. Kahneman and D. T. Miller, "Norm theory: Comparing reality to its alternatives," *Psychological Review*, vol. 93, no. 2, pp. 136–153, 1986. 4.2.5.2

[72] T. Eiter and T. Lukasiewicz, "Complexity results for structure-based causality," *Artif. Intell.*, vol. 142, no. 1, pp. 53–89, Nov. 2002. [Online]. Available: http://dx.doi.org/10.1016/S0004-3702(02)00271-0 4.3.2

[73] M. Hopkins, "The actual cause: From intuition to automation," Ph.D. dissertation, University of California, Los Angeles, 2005. 4.3.2

[74] N. Hall, "Causation and the price of transitivity," *Journal of Philosophy*, vol. 97, no. 4, pp. 198–222, 2000. 4.3.3

[75] E. G. on Tort Law, *Principles of European Tort Law: Text and Commentary*. Springer, 2005. [Online]. Available: http://books.google.com/books?id=3Najct7xGuAC 4.3.5, 5.1

[76] J. Schaffer, "Trumping preemption," *Journal of Philosophy*, vol. 97, no. 4, pp. 165–181, 2000. 4.4.3

[77] D. Wendlandt, D. G. Andersen, and A. Perrig, "Perspectives: improving ssh-style host authentication with multi-path probing," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*. CA, USA: USENIX Association, 2008. 5.1, 6.2

[78] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. 3, no. 2, pp. 125–143, 1977. 5.2.3

[79] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, pp. 181–185, 1985. 11

[80] W. Rafnsson, D. Hedin, and A. Sabelfeld, "Securing interactive programs," in *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, ser. CSF '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 293–307. [Online]. Available: http://dx.doi.org/10.1109/CSF.2012.15 5.2.4

[81] A. Groce, S. Chaki, D. Kroening, and O. Strichman, "Error explanation with distance metrics," *International Journal on Software Tools for Technology Transfer*, vol. 8, no. 3, pp. 229–247, 2006. 6.1

[82] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: Localizing errors in counterexample traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '03. New York, NY, USA: ACM, 2003, pp. 97–105. [Online]. Available: http://doi.acm.org/10.1145/604131.604140 6.1, 7.1.3

[83] D. A. Lagnado and S. Channon, "Judgments of cause and blame: The effects of intentionality and foreseeability," *Cognition*, vol. 108, no. 3, pp. 754–770, 2008. 6.1, 6.2

[84] " Internet-Draft: Public Key Pinning Extension for HTTP," 2012. [Online]. Available: http://tools.ietf.org/html/draft-ietf-websec-key-pinning-01 6.2

[85] " Internet-Draft: Trust assertions for certificate keys (TACK)," 2012. [Online]. Available: https://tools.ietf.org/html/draft-perrin-tls-tack-02 6.2

[86] C. Soghoian and S. Stamm, "Certified lies: Detecting and defeating government interception attacks against ssl (short paper)," in *Proceedings of the 15th International Conference on Financial Cryptography and Data Security*, ser. FC'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 250–259. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27576-0_20 6.2

[87] M. Alicherry and A. D. Keromytis, "Doublecheck: Multi-path verification against man-in-the-middle attacks," in *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on.* IEEE, 2009, pp. 557–563. 6.2

[88] M. Marlinspike, "SSL and the future of authenticity," Black Hat USA, 2011. 6.2

[89] A. Barth, J. C. Mitchell, A. Datta, and S. Sundaram, "Privacy and utility in business processes," in *CSF*, 2007, pp. 279–294. 6.3, 6.3.2

[90] G. Gössler and D. L. Métayer, "A general trace-based framework of logical causality," in

Formal Aspects of Component Software - 10th International Symposium, FACS 2013, 2013, pp. 157–173. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-07602-7_11 6.3.2

[91] C. Hitchcock and J. Knobe, "Cause and norm," pp. 587–612, 2009. [Online]. Available: http://philsci-archive.pitt.edu/8352/ 7.1.2

[92] A. Datta, A. Derek, J. C. Mitchell, and D. Pavlovic, "A derivation system and compositional logic for security protocols," *Journal of Computer Security*, vol. 13, no. 3, pp. 423–482, 2005. C

[93] D. Garg, J. Franklin, D. Kaynar, and A. Datta, "Compositional system security with interface-confined adversaries," *Electron. Notes Theor. Comput. Sci.*, vol. 265, pp. 49–71, Sep. 2010. C, C.B.2, 1