# Confidentiality in the Process of (Model-Driven) Software Development

Michael Johnson
Departments of Mathematics and Computing
Macquarie University
Australia
Michael.Johnson@mq.edu.au

Perdita Stevens
School of Informatics
University of Edinburgh
Scotland
Perdita.Stevens@ed.ac.uk

## ABSTRACT

Much is now understood about how to develop software that will have good security properties in use. We claim that a topic which needs more attention, in particular from the Bx community, is security, especially confidentiality, in the software development process itself. What is then at issue is not what particular users of the software may be allowed to know, but rather, what particular developers of the software may be allowed to know. How can software development processes guarantee to respect confidentiality without compromising effective development?

The question is of general interest across software engineering, but model-driven development (MDD) seems a particularly promising arena in which to address it, because of MDD's focus on separation of concerns. In MDD, different people work with separate models, where (ideally) each model records all and only the information necessary to those who work with it. When necessary, the models are reconciled by bidirectional transformations, which automate a process which would otherwise have to be undertaken manually by the groups of experts meeting and studying both their models in order to bring them back into consistency. In model-driven development confidentiality issues become particularly clear and tractable, and bidirectional transformations have a key technical role. We hope to encourage the community to take up this challenge, and in this paper we begin our own analysis of a selection of the issues, focusing particularly on developing a threat model and some examples of secure restoration of consistency.

## CCS CONCEPTS

**• Mathematics of computing**; **• Security and privacy → Social aspects of security and privacy**; **• Software and its engineering → Software development process management**;

## KEYWORDS

Security, Model-driven software development, Confidentiality, Cospan

*<Programming'18> Companion, April 9–12, 2018, Nice, France*
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5513-1/18/04…$15.00
https://doi.org/10.1145/3191697.3191714

**ACM Reference Format:**
Michael Johnson and Perdita Stevens. 2018. Confidentiality in the Process of (Model-Driven) Software Development. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3191697.3191714

## 1 INTRODUCTION

Novel cyber-security breaches usually depend upon detailed understanding of the internal structure of the system being attacked. Famously, Barnaby Jack [21] purchased outright two automatic teller machines so that he could analyse the internal structure of their code and ultimately develop what became known as "Jackpotting" exploits. The Greek Vodafone breaches [22] depended fundamentally upon esoteric understanding of the AXE programming language and the code used in Ericsson R9.1 system software for telecommunication switches. The development of novel and extremely difficult-to-detect industrial robot exploits [23] depended upon purchasing and reverse engineering multiple industrial robots. There are of course many more examples.

Often the required structural information is obtained by reverse engineering. Sometimes it comes from painstaking information gathering over extended periods after a system has initially been breached. And more and more frequently, especially in cases of state-sponsored cyber attack, it comes from software engineers who are familiar with the internal structure of the system through their involvement in its development or maintenance.

Surprisingly, despite all this, there seems to have been remarkably little work by our security community on processes for preserving confidentiality during software development and maintenance. Traditionally trust between developers, and between developers and customers, has simply been regarded as a good thing; in agile methodologies, especially, it is regarded as crucial [20], and supported by practices such as "customer on site" and "no code ownership". However, this sometimes results in conflating trusting someone to do something (reliance) with trusting them with information (transparency), concepts which we sometimes need to separate. [1]

An unwillingness to share information freely can indeed be a "bad smell", an indication that teamwork is failing and cultural issues need to be addressed. On the other hand, there are times when it is legitimately undesirable that everyone should have access to every piece of information. The clearest examples arise where development is being done collaboratively by different organisations, which may naturally want to share only as much as is essential

[1] See e.g. http://www.scaledagileframework.com/safe-core-values/.

to the collaboration. Within a single organisation, some artefact, such as a key algorithm that gives the organisation a competitive advantage, should sometimes be available only to those with need-to-know, in order to limit insider threat. Less obviously, there is a case for limiting access, e.g. to specific implementations of interfaces, in order to avoid hidden dependencies on things that might change later. It may not be easy to share the information that should be shared for effective work, while still keeping other information private; this depends, in part, on the development approach chosen.

Model-driven development (MDD) is an approach to software development which prioritises separation of concerns. Decisions about the software are taken by different groups of experts, working in separate models, where each model records the information necessary to that group, and allows them to record their decisions. When necessary, the models are reconciled by bidirectional transformations, which automate a process which would otherwise have to be undertaken manually by two groups of experts meeting and studying both their models in order to bring them back into consistency. In most settings where MDD has been used to date, there is no pressing need to avoid users of one model having full knowledge of someone else's model; rather, the main motivations for automating the restoration of consistency are efficiency and reliability of the development process. However, as MDD expands from the niches where it was originally used, and especially as models are used to articulate the connections between organisations with different interests, situations are encountered where aspects of one model may be confidential.

Our contributions are as follows.

(1) We draw together existing work on confidentiality in the software development process (Section 2).
(2) We sketch a threat model applicable to collaborative MDD (Section 3).
(3) We describe an approach to confidentiality in MDD which we have found to be useful in practice (Section 4).
(4) We examine and distinguish two proposals for using bidirectional transformations to enhance security and consider their potential interactions (Section 5).
(5) We suggest a research agenda, hoping to stimulate the community to further work on this topic (Section 6).

## 2 RELATED WORK

There is, of course, a considerable literature on security concerns in model-driven development; see for example [10] for an early overview of work including [1], which exploits bx towards developing systems with specified security properties. However, this literature concerns, not the security of the *process* of development, but the security properties of the *system* developed. The former is our concern in this paper.

Where software development is undertaken within a single organisation, it is usually assumed that all developers are, or at least should be if the organisation is healthy, fully trustworthy; the possibility that this may not be the case is a particular kind of *insider threat* [3]. Despite a considerable amount of work on insider threat, little of it pertains to the software development process, and that little focuses on integrity. For example, it is recommended to review commit logs to configuration management tools to detect the insertion of malicious code, to require separate people to write and review code (separation of duties – of course this practice has other advantages) and to use reference monitors to detect insiders tampering with deployed software [19, 24]. Organisational reporting mechanisms also receive attention in the literature. It is recognised that the risks increase when those with insider knowledge may be outsiders, e.g. contractors (see [22] for discussion of a prominent example) and legal measures such as NDAs may be used in an attempt to reduce risk to confidentiality [25]. Nevertheless, when software is developed in a single organisation, it is normal that all developers have unlimited read (at least) access to software development artefacts. The perception is that it is important that all are trustworthy and trusted, and that departures from this are, and should be regarded as, anomalous. Part of what drives this is the difficulty in restricting access without impeding development. Typically software architecture does not afford a clean separation between what is confidential and what is not, and a potential hazard is that those with access may not themselves understand the distinction clearly and may give away confidential information without meaning to, or conversely, may impede collaboration by over-caution.

Protection of data used within software development has received more attention than protection of other artefacts. For example, it is widely understood that good practice is to avoid using production data during development and testing, or at least, to take precautions such as masking it or reducing (subsetting) it. Even so, this best practice is often not followed; for example, a 2009 report [15] found "80% of respondents in the US and 77% in the UK report that they use real production data as part of their application development and testing process", in the majority of cases without masking or reducing, and with less stringent data protection procedures in the development and test environment than the production environment.

Trust receives more attention when it is more difficult to obtain: in circumstances where individuals do not know one another well. Let us consider in turn *distributed software development*, and its extreme case *global software development*; interoperation between distinct organisations; open source developments; and crowdsourcing.

The literature on trust in distributed software development mostly focuses on how to increase trust [8], because people (for reasons that are sometimes considered good, sometimes bad) find it more difficult to trust people in other places. Where development is distributed over different countries, this can be even more difficult. For example, Jalali et al.[16] discuss trust in global software engineering. Their focus is on the human relationships involved: increasing trust between developers situated in different countries is seen as a way to ensure that they are willing to exchange information freely about the details of their work. They highlight that trust has both cognitive and affective elements. Affective trust is founded on human relationships and is difficult to establish in a distributed development environment; the paper discusses ways to do so.

It is once we move out of the realms of software developed by a single organisation that authors become more willing to consider that perhaps not all developers *should* be fully trusted. A strand of research exemplified by Gallivan's work [13] argues that the human aspect of trust is dispensable in the presence of appropriate organisational controls. He concludes that open source software development does not, in fact, furnish examples of widely distributed trust, as is sometimes claimed. Rather, there is a core group of developers who trust *one another*, and rigid control mechanisms that obviate the need for these developers to trust peripheral developers or posters. This in turn is what enables trust in the software.

Dubey et al.[9] address the problem of confidentiality in crowd-sourced software development. They are concerned with cases where programming tasks are to be put in a marketplace, but their most natural descriptions might give away confidential information to the "gig economy" workers. They propose a natural language processing based approach, where certain terms are identified as potentially sensitive, task descriptions are automatically sanitised before posting, and the resulting work desanitised before being incorporated in the final software.

Intriguing though this approach is, what seems most remarkable is that, as far as we have been able to discover, there is not more work addressing technical aspects of confidentiality in software development involving parties with possibly conflicting interests. In particular, Dubey et al. highlight that we so far lack a threat model, a gap we will address here.

The one interesting exception is a single paper by Foster, Pierce and Zdancewic on "Updatable security views" [12]. In a setting of asymmetric lenses, they propose what is in effect a type system for part of the Boomerang language in which parts of the source and view can be labelled with elements of a lattice representing integrity or confidentiality levels. A source, containing information to which confidentiality or integrity requirements pertain, is related by a lens to a "security view" which may be updated by an untrusted user. The lens is permitted by the type system only if it preserves appropriate information-flow properties. A notable issue is the interaction between integrity and confidentiality requirements, partially addressed in this paper by dynamic analysis. For example, the type system may ensure that an updated view is a valid argument to a put function only if it agrees with the get of the current source in certain respects, corresponding to source data whose integrity must be preserved. However, forbidding the put is information to the user of the view: for example, it may reveal that there is a relationship between apparently innocuous data in the view, and important hidden data in the source. This may be important *if the user of the view could not know it already.* Here it becomes important to think about the threat model. The paper [12] assumes that only the user of the view is untrusted: the author of the lens has full access to the source.

## 3 THREAT MODEL

Broadly, to develop a threat model relevant to the software development process we need to:

(1) determine what **setting** we are assuming, i.e. the scope of the threat model;
(2) identify (and, in a particular instance, rank) the **potential harms** to guard against;
(3) make explicit assumptions about the capabilities and knowledge of an **adversary**.

After this has been done it makes sense to consider security policies and mechanisms that can prevent the identified harms. Let us consider these elements in turn.

### 3.1 Setting

Assume that two organisations, which we call A and B, need to co-operate over the building and/or operation of some software. They need to exchange information; but at least one of them possesses, or expects to possess, information that should not be passed to the other.

This covers a wide variety of set-ups. We will generally assume, for interest, that the information the organisations need to exchange is in some sense rich: it does not suffice for software built by A to call functions in a fixed API maintained by B, for example. Our paradigmatic example, from the field of model-driven development, is that A owns a model $m$, B owns a model $n$, and these models need to be kept consistent in an appropriate sense.

A and B might of course be different parts of the same organisation: what matters is that it is not valid to assume that their interests are perfectly aligned.

### 3.2 Potential Harms

It is important to note that the "information" whose confidentiality and integrity we are concerned about, while including items that might be traditionally thought of as "data" stored in the system, can also include aspects of the way the system itself is structured, or the way data is stored. Knowledge of such things can facilitate the development of exploits, and in many proprietary systems which operate "on-premises" (rather than being distributed to clients), those details are often carefully guarded. Indeed, one of the motivations for this paper is that in consultancy work that we have done, the tension between the need to keep internal structures secret and the desire to build software to facilitate interoperation has, when traditional software engineering practices are used, threatened organisations' willingness to continue with the project.

Let us elaborate this point by developing a list.

*What might be confidential?* Many different artefacts involved in software development may, in whole or part, need to be regarded as confidential. They might include:

- aspects of requirements, e.g., identities of commissioning customers;
- metamodels, for example, of domain-specific modelling languages for defining families of products whose functionality may be confidential;
- design models;
- software architecture;
- code;
- tests;
- configurations;
- model transformations;
- traceability information;[2]
- data, of course.

We do not at present have full understanding of the threats posed by leakage of information about these artefacts, severally

[2]Thanks to the reviewer who pointed out that we had omitted this

or in combination. We should also ask: what kind of information in each artefact needs to be protected? This will affect the actions that can be taken in response; for example, Dubey at al.'s work [9] assumed that sanitisation, e.g. the replacement of a term by a more general term, would suffice – when is this the case? They point out that images might also need to be sanitised. However, in some cases structural information, devoid of all linguistic content, can still need to be held confidential. Unfortunately, it is difficult to determine *what* structural information can be leaked without compromising security. When dealing with outsiders, organisations often err on the side of caution – for example, in our experience, it can be difficult to get companies to agree to have their real design models made public, even if all strings in them are deleted – but the implications for distributed software development do not seem to have been explored.

*Reasons for confidentiality requirements.* Having considered *what* might be confidential, let us next consider *why*.

- Protection of intellectual property/trade secrets, such as [14]:
  - the need to protect information that would genuinely harm your organisation if it leaked, e.g., an algorithm that gives you a competitive advantage, or any information that might enable attacks on integrity or availability;
  - the need to be seen to protect information in order to maintain legal control over it, e.g., so as to patent it;
  - protection of someone else's IP, i.e. you have agreed to protect something for someone else, perhaps as a condition of using it.
- Protection of personal data, e.g., the content of a database containing information about members of the public, or (in certain organisations) information about employees and their roles.
- Reputational risk, e.g., potential embarrassment at the release of artefacts whose quality might be criticised.
- (Especially where A and B are different parts of the same organisation.) Avoidance of conflict of interest, e.g., need to maintain "Chinese walls" between information that should not be shared in order to avoid insider trading.

A general observation is that even if there is, at corporate level, a highly trusting relationship between organisations A and B, there may still be strong confidentiality requirements on the software artefacts. For example, an individual currently working at A or B might subsequently move: their individual interests may not be perfectly aligned with the organisation's interests.

## 3.3 Adversary Model

Take the position of organisation A, in the case where A and B cooperate to maintain consistency between models $m$ and $n$. We may want to consider adversaries with capabilities drawn from the following:

(1) full knowledge of model $n$ (read access to $n$)
(2) ability to change $n$ (e.g. to probe $m$)
(3) ability to cause consistency restoration to be done at will
(4) knowledge of the consistency restoration process, including the definition of consistency
(5) ability to change the consistency restoration process, including the definition of consistency.

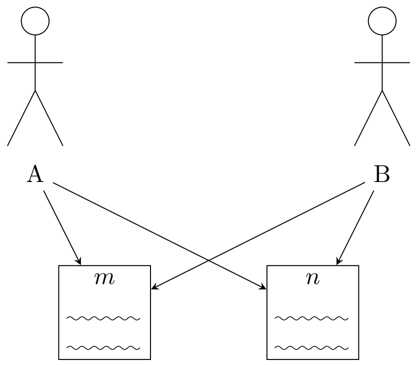

**Figure 1: Without automation, consistency restoration involves full trust**

## 4 STEPS TOWARDS SECURE RESTORATION OF CONSISTENCY

In this section we focus on a specific problem case. Two organisations A and B need to cooperate, and this involves maintaining a certain consistency relationship between two models, $m$ and $n$. How can they do it? We will assume that the adversary model gives the attacker (at B from A's point of view, and dually) all the capabilities except the ability to change the consistency restoration process: we will suggest designing that process so as to resist releasing confidential information.

*Initial situation.* In the ideal, full trust, non-automated situation this would involve people from both A and B sitting down periodically, looking at $m$ and $n$ side by side, and modifying one or both to bring them back into consistency. ($m$ and $n$ might be database schemas, UML models, user interaction models, bodies of code...). This requires engineers from both A and B to be given full knowledge of the contents of models $m$ and $n$. (See Figure 1.)

*Adding an automated (possibly bidirectional) transformation.* Where the process of restoring consistency needs to be automated, this situation is formalised in the bidirectional transformation literature by defining a consistency relation $R \subseteq M \times N$, where $M$ and $N$ are the sets of possible models, i.e. $m \in M$ and $n \in N$. We often use the simplifying assumption that each time consistency is restored, one of $m$, $n$ (perhaps the one that has just been changed) is *authoritative* and should not be changed automatically. (This helps to avoid automated consistency simply undoing a change that has been deliberately made by a developer.) We formalise this as adding a pair of consistency restoration functions $\overrightarrow{R} : M \times N \longrightarrow N$ and $\overleftarrow{R} : M \times N \longrightarrow M$ satisfying certain laws to ensure sensible round-trip behaviour. The problem, from the point of view of confidentiality, is that each of these consistency restoration functions needs to be given both models as arguments. This is essential in the general case, because each model includes information which is not present in the other.

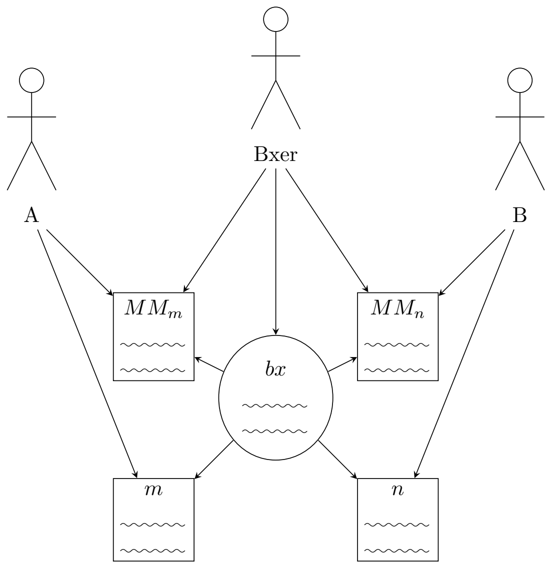

**Figure 2: Using a bidirectional transformation, consistency restoration involves less trust**

Now, if consistency restoration is done automatically by means of such a transformation, we have the potential to avoid engineers from A and B needing to look at one another's models, which is, arguably, an advance. Each of them can instead simply invoke the consistency restoration function, and, instead of seeing the other model, will observe only the effect on their own model of consistency restoration with the other model.

Somebody, however, has to write the bidirectional transformation.[3] Typically, this development is done using examples of the models, and with full access to the metamodels that define the languages from which the models are drawn. Any debugging of the transformation is normally done using the models that illustrate buggy behaviour. Thus, those responsible for the development and maintenance of the bidirectional transformation still require full access to both *m* and *n*. To some extent this may be avoided: even if access to full metamodels is required (and work on model polymorphism may change this in future [6]), it is possible in principle for the transformation to be developed and debugged using hypothetical – minimised/sanitized/obfuscated – examples of the models. There are obvious and severe disadvantages, such as the effort required to construct such examples, especially where the engineers who do have access to the full models may not deeply understand the working of the transformation and may therefore find this hard. There is scope for research in easing this process.

Now, even if we arrange that the developer of the bidirectional transformation does not need access to both models, the transformation itself still does. (Figure 2.) Does this matter? It may: for one thing, the transformation has to run somewhere, and using today's technologies, that means that some machine has to host the transformation and hence information about both models. Access to the machine has to be adequately controlled. For another thing, the effect of the transformation on one model gives away information about the content of the other. How much information could a malicious engineer at B extract about *m* by using different variants of *n* to "probe" *m* by repeatedly applying the consistency restoration function and observing the effect on her variants of *n*? In a particular setting, this could be tackled formally. If the consistency restoration functions themselves are fixed, we expect to be able to place limits on the deducible information: informally[4], if the transformation never looks at a particular, sensitive, part of a model, then information from that part should not leak. If, however, our malicious engineer is able to modify the bidirectional transformation too, and if this is written in terms of the whole metamodels, and so in principle has access to the whole models, we should not expect any useful confidentiality result. This is reminiscent of the problems whose exploration led to the invention of differential privacy for statistical queries over databases, following the understanding that no other known approach held much promise for maintaining confidentiality of a database which could be freely queried, even if queries were limited in, for example, the number of tuples they would return or omit.

[3] It might be objected – it was objected, by a reviewer – that machine learning, for example, might obviate this necessity. We think the general point still stands: today, even "unsupervised" learning requires, in practice, human access to the data over which the learning takes place. Conceivably, in future, another level of abstraction might indeed help, but we doubt this will affect the point any time soon.

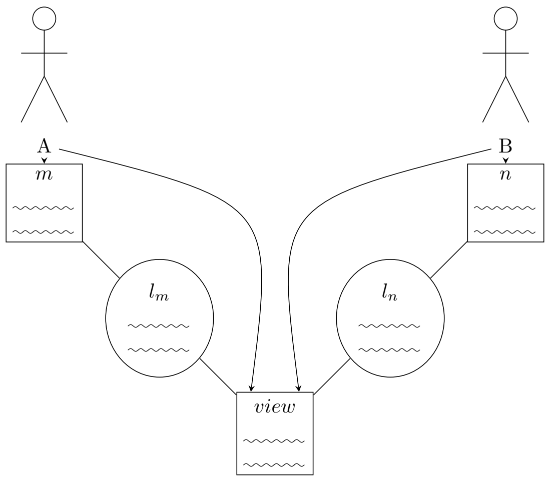

**Figure 3: Using a common view, trust can be strictly limited**

We have been discussing the symmetric case of a bidirectional transformation, that in which each model contains information not captured in the other, and therefore, in which consistency restoration in either direction requires access to both models. Cases in which one model includes all the information included in the other – the second is a *view* onto a *source* – are somewhat simpler. They can be captured by (asymmetric) *lenses* [11]. Then a *get* function takes just a source and yields a view, while a *put* function takes both a source and a view, and yields a modified source. Since the *get* function does not require read access to the view model, this might look like an advance, but by itself it does not help us. Putting lenses with a common view together, however, does.

[4] but there is some delicate (future) work to be done to make a correct, formal version of the statement!

*Using a common view.* In many – but not all – cases, a given bidirectional transformation can be expressed using a pair of lenses having a common view, as explained in [27]. Each of $m$ and $n$ is regarded as the source of its own lens, and consistency is defined and restored via this pair of lenses. Informally, $m$ and $n$ are regarded as consistent if, and only if, the views, via their respective lenses, are identical; forward consistency restoration is done by forming the view of $m$ via the *get* function of its lens, and then using this view and $n$ as arguments to the *put* function of the other lens in order to generate a new version of $n$. Backward consistency restoration is dual.

The practical importance of this procedure is that if the owners of $m$ and $n$ can agree on a common view which does not contain information that either regards as confidential from the other, they can use this to cooperate without needing to share more than is in the view. Each can have their own developers build their own lens connecting the common view to their own model, assuring that it does not leak confidential information. (Figure 3.)

While not every bidirectional translation can be expressed in this way – the paper [27] discusses the property of being *simply-matching* which captures those set-based bidirectional transformations which can be, and in the Bx Example Repository, example Wikipedia Translation [26] is a fairly realistic example of one that cannot be – it seems from our industrial work that many bidirectional transformations that arise in practice can be. The paper [17] developed some necessary conditions for symmetric delta lenses to be representable through a common view (and uncovered some of the links with least change lenses). Most recently [18] characterises those symmetric delta lenses which can be so represented.

Dubey et al.'s code-outsourcing with sanitisation [9] can be seen as an example, and amusingly, so can the use of an insider threat ontology [4].

## 5 SECURE RESTORATION OF CONSISTENCY WHILE MAINTAINING CONFIDENTIALITY

It is interesting that despite the paucity of work on the application of bidirectional transformations to the security of software engineering practices, there are two quite different proposals – one presented in Section 4 above and the other in reference [12] – for using bidirectional transformations to enhance security. In this section we compare and contrast the two of them. First we note their differences, and then we speculate upon how they might work together.

To begin we return to the updatable security views of Foster et al [12], first mentioned at the end of Section 2. The work presented there is intended to maintain the confidentiality and integrity of identified data in the source of an asymmetric lens. It should be noted that that work, at this stage, only deals with asymmetric lenses. It is about preserving the confidentiality of information stored in the source of the lens, and possibly permitting consistency restoration while preserving the integrity of identified information in the source. The development of such lenses depends, in the usual framework of bidirectional programming, upon software engineers having full knowledge of the models involved.

In contrast, the cospan approach presented in Section 4 concentrates on how to *build* a bx so as to preserve the confidentiality of the models involved. It is intended for building symmetric lenses, while ensuring that the details of the models at each end of the symmetric lens can remain confidential and only be accessed directly by the organisation that owns them, along with that organisation's own software engineers. The flow of data between the organisations is strictly controlled with the only explicit transfers occurring via the common view, and that view is the precisely stated material that the organisations have agreed to share.

So, in summary, updatable security views identify, among the data stored in a lens source, levels of data confidentiality and integrity, and then maintain those levels during operation, while the cospan approach is a software engineering technique for developing symmetric lens interoperations while preserving confidentiality.

Since the two techniques address substantially different security aspects, one might be interested in whether they can work together to achieve even better security outcomes.

In one direction, there appears to be little advantage. Although, as noted above, the development of updatable security views depends upon software engineers being given full access to all of the models, there would be little to be gained by attempting to factor that development process using cospans. Updatable security views are asymmetric lenses. In the asymmetric case full access to models is not a prime concern because the master-slave relationship of the two components of an asymmetric lens implies that one organisation already controls all of the data, and that organisation will, in general, trust its own software engineers.

On the other hand, there do appear to be advantages in considering using updatable security views as component lenses in a cospan. Decomposing a proposed bidirectional transformation into a cospan of lenses, each of which might protect its source's data confidentiality and integrity using techniques like those proposed in [12] would, when the bidirectional transformation is a symmetric lens, improve confidentiality in the development process (through the cospan approach), and provide integrity and confidentiality guarantees during interoperation (through the updatable security views).

In our experience, when organisations have been able to agree on the data on which to interoperate, there has been little concern about confidentiality as long as the organisations can be assured, via the common view interface, that only those data are revealed. But there have been occasions where contractual obligations were required to ensure that certain updates of the common data by one party or the other are *not* allowed. This is for integrity reasons, sometimes related to undesired effects of the Puts of the component asymmetric lenses, usually because the Puts don't meet least change requirements, and it was part of the motivation for the half-duplex interoperations developed in [5]. Using updatable security views as the asymmetric lenses could provide an automatic enforcement mechanism for the contractual obligations, and would provide a natural integration of confidentiality and integrity mechanisms into the development process, rather than having external contractual agreements that need to be separately monitored.

## 6 RESEARCH ROADMAP

This paper seeks to stimulate work in software development and maintenance processes that preserve confidentiality. Increasingly, systems interoperation occurs across organisational, hence trust, boundaries. We have outlined how greater confidentiality can be achieved using known MDD techniques in such a setting. Drawing on both theoretical considerations and our industrial consultancy experience, here is a non-exhaustive list of questions we think should be addressed.

(1) We have noted from the applications that surprisingly often a practical situation could indeed be represented as a cospan of lenses with a common view. Why? In which domains?
(2) What formal support can be offered for the identification of a common view and the development of lenses onto it, with guarantees about information flow? (Note that the last is not trivial, e.g. when the lenses are not *very well-behaved* [11] dependencies between parts of a model can cause intuitively surprising effects.)
(3) Can a formalisation making use of trajectory pairs as in the organisational dimension of [7] give us any leverage, either on confidentiality or on integrity?
(4) We have discussed *bi*-directional transformations, but some interoperations involve more than two systems, and this is now getting attention in MDD (e.g. there is an upcoming Dagstuhl, Sem. 18491). What are the confidentiality implications?
(5) (When) can techniques like these be useful in the design and development of software that traditionally would be thought of as one unified, albeit structured, body of code? Are there advantages to be obtained by building teams which maintain confidentiality between themselves while working on one project for one client, and whose code bases interact via confidentiality preserving transformations?
(6) Foundational concerns are not the only important ones. Trust is a complex matter [2] and changes to the software development process can have unintended consequences. While we hope that foundational work, by clarifying what is confidential, can enable greater affective trust [16] and more effective development, that is a matter for empirical study.

## 7 CONCLUSIONS

The application of bidirectional transformations to enhance security properties of software, and particularly of the software development process, is promising, and as yet little developed.

In this paper we have identified confidentiality concerns that arise within model-driven development, which should be addressed. We have sketched a threat model. We have argued for the interface (cospan) model as a pragmatic and often useful guard against some of these threats. We have reviewed the only other application of bidirectional transformations to security that we have been able to find, and considered its possible application along with the cospan model. And we have proposed new directions for foundational and other research. In future, we plan to exploit the threat model more fully, and to expand the discussion to better cover other security concerns, such as integrity and authorisation, which in a bx setting are interestingly related to confidentiality.

Security remains one of the major challenges for software engineering. Modern software engineering is more and more concerned with the management of interactions between systems. And model driven software development, along with bidirectional transformations, promises improved separation of concerns in the software engineering process, with significant opportunities for properly controlling system interactions.

## ACKNOWLEDGMENTS

We thank the reviewers for their helpful comments, both for improving this paper and for future work.

## REFERENCES

[1] Howard Chivers and Richard F. Paige. 2009. XRound: A reversible template language and its application in model-based security analysis. *Information & Software Technology* 51, 5 (2009), 876–893. https://doi.org/10.1016/j.infsof.2008.05.006
[2] Karen Clarke, Gillian Hardstone, and Mark Rouncefield (Eds.). 2006. *Trust in Technology: A Socio-Technical Perspective.* Springer.
[3] Matthew Collins, Michael Theis, Randall Trzeciak, Jeremy Strozer, Jason Clark, Daniel Costa, Tracy Cassidy, Michael Albrethsen, and Andrew Moore. 2016. *Common Sense Guide to Mitigating Insider Threats* (fifth ed.). Technical Report CMU/SEI-2016-TR-015. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=484738
[4] Daniel Costa, Matthew Collins, Samuel J. Perl, Michael Albrethsen, George Silowash, and Derrick Spooner. 2014. An Ontology for Insider Threat Indicators: Development and Application. In *STIDS (CEUR Workshop Proceedings)*, Vol. 1304. CEUR-WS.org, 48–53.
[5] C.N.G Dampney and Michael Johnson. 2001. Half-duplex Interoperations for Cooperating Information Systems. In *Advances in Concurrent Engineering*.
[6] Thomas Degueule, Benoît Combemale, Arnaud Blouin, Olivier Barais, and Jean-Marc Jézéquel. 2017. Safe model polymorphism for flexible modeling. *Computer Languages, Systems & Structures* 49 (2017), 176–195. https://doi.org/10.1016/j.cl.2016.09.001
[7] Zinovy Diskin, Hamid Gholizadeh, Arif Wider, and Krzysztof Czarnecki. 2016. A three-dimensional taxonomy for bidirectional model synchronization. *Journal of Systems and Software* 111 (2016), 298–322. https://doi.org/10.1016/j.jss.2015.06.003
[8] Siva Dorairaj and James Noble. 2013. Agile Software Development with Distributed Teams: Agility, Distribution and Trust. In *AGILE*. IEEE Computer Society, 1–10.
[9] Alpana Dubey, Kumar Abhinav, and Gurdeep Virdi. 2017. A framework to preserve confidentiality in crowdsourced software development. In *ICSE (Companion Volume)*. IEEE Computer Society, 115–117.
[10] Eduardo Fernández-Medina, Jan Jürjens, Juan Trujillo, and Sushil Jajodia. 2009. Model-Driven Development for secure information systems. *Information & Software Technology* 51, 5 (2009), 809–814. https://doi.org/10.1016/j.infsof.2008.05.010
[11] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* 29, 3 (2007), 17.
[12] J. Nathan Foster, Benjamin C. Pierce, and Steve Zdancewic. 2009. Updatable Security Views. In *CSF*. IEEE Computer Society, 60–74.
[13] Michael Gallivan. 2001. Striking a balance between trust and control in a virtual organization: a content analysis of open source software case studies. *Inf. Syst. J.* 11, 4 (2001), 277–304. https://doi.org/10.1046/j.1365-2575.2001.00108.x
[14] Christopher George and Raymond Millien. 2015. Protecting IP in an agile software development environment. *IPWatchdog* (December 2015). http://www.ipwatchdog.com/2015/12/28/protecting-ip-agile-software-development/id=64171/ Retrieved 17/11/17.
[15] Ponemon Institute. 2009. Data security in development and testing. https://www.microfocus.com/media/report/ponemon-institute—data-secur_tcm6-7227.pdf. (July 2009).
[16] Samireh Jalali, Çigdem Gencel, and Darja Smite. 2010. Trust dynamics in global software engineering. In *ESEM*. ACM.
[17] Michael Johnson and Robert Rosebrugh. 2017. Universal Updates for Symmetric Lenses. *CEUR Proceedings* 1827 (2017), 39–53.
[18] Michael Johnson and Robert Rosebrugh. 2018. Cospans and Symmetric Lenses. (2018). In this volume.
[19] Richard Kissel, Kevin Stine, Matthew Scholl, Hart Rossman, Jim Fahlsing, and Jessica Gulick. 2008. *Security Considerations in the System Development Life Cycle.* Technical Report Special Publication 800-64 Revision 2. NIST. https://csrc.nist.gov/publications/detail/sp/800-64/rev-2/final
[20] Orla McHugh, Kieran Conboy, and Michael Lang. 2012. Agile Practices: The Impact on Trust in Software Project Teams. *IEEE Software* 29, 3 (2012), 71–76.

[21] Robert McMillan. 2010. Barnaby Jack hits ATM jackpot at Black Hatx. *ComputerWorld* (July 2010). https://www.computerworld.com/article/2519671/computer-hardware/barnaby-jack-hits-atm-jackpot-at-black-hat.html
[22] Vassilis Prevelakis and Diomidis Spinellis. 2007. The Athens Affair. *IEEE Spectrum* (June 2007).
[23] Davide Quarta, Marcello Pogliani, Mario Polino, Federico Maggi, Andrea Maria Zanchettin, and Stefano Zanero. 2017. An Experimental Security Analysis of an Industrial Robot Controller. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 268–286.
[24] Ron Ross, Michael McEvilley, and Janet Carrier Oren. 2016. *Systems security engineering*. Technical Report Special Publication 800-160. NIST. https://csrc.nist.gov/publications/detail/sp/800-160/final
[25] Bret J. Stancil. [n. d.]. What To Look Out For In Software Development NDAs. https://www.toptal.com/it/what-to-look-out-for-in-software-developer-ndas. ([n. d.]). Retrieved 17/11/17.
[26] Perdita Stevens. [n. d.]. Wikipedia Translation v0.1 in Bx Examples Repository. http://bx-community.wikidot.com/examples:home. ([n. d.]). Date retrieved: 26/2/18.
[27] Perdita Stevens. 2012. Observations relating to the equivalences induced on model sets by bidirectional transformations. *EC-EASST* 049 (2012).