Composing Patterns to Construct Secure Systems

Paul Rimba, Liming Zhu, Len Bass, Ihor Kuz
Software Systems Research Group, NICTA
School of Computer Science and Engineering, UNSW
NSW, Australia
Email: {firstName.lastName} @nicta.com.au

Steve Reeves
Department of Computer Science
The University of Waikato
Hamilton, New Zealand
Email: stever@waikato.ac.nz

***Abstract*—Building secure applications requires significant expertise. Secure platforms and security patterns have been proposed to alleviate this problem. However, correctly applying patterns to use platform features is still highly expertise-dependent. Patterns are informal and there is a gap between them and platform features. We propose the concept of reusable verified design fragments, which package security patterns and platform features and are verified to provide assurance about their security properties. Design fragments can be composed through four primitive tactics. The verification of the composed design against desired security properties is presented in an assurance case. We demonstrate our approach by securing a Continuous Deployment pipeline and show that the tactics are sufficient to compose design fragments into a secure system. Finally, we formally define composition tactics, which are intended to support the development of systems that are secure by construction.**

***Keywords*—security; verification; patterns; composition; assurance**

I. Introduction

Building a secure application is difficult and requires significant expertise and effort. A secure application requires a secure design, a secure implementation of that design, and a secure platform on which the implementation executes. Furthermore, it is not sufficient that an application be secure, it must also be seen to be secure. That is, the argument that an application is secure is an essential portion of convincing others of the level of security of an application. We examine these four elements in slightly more detail now.

- *Secure platform* — A secure platform provides the foundation for building large security-critical applications [1]–[3]. Such a platform consists of a combination of hardware and software that provide security mechanisms and policies usable by the application. While this provides a solid base for building secure applications, it does not guarantee that applications constructed on top of this platform are secure. That requires expertise.
- *Secure design* — The use of security patterns [4], tactics [5] and best practices helps design security-critical applications. A security pattern is an encapsulation of expert knowledge and best practices in the area of secure software design [6]. While security patterns significantly reduce the required expertise, there are still several difficulties in applying them. Firstly, a single security pattern normally cannot meet all the security requirements of an application. Therefore, a developer needs to apply multiple security patterns to a design through design composition. Composition of security patterns is challenging because each pattern targets specific security requirements and has its own elements (e.g. actors involved and their interactions). Secondly, security patterns are written independently of the specifics of the underlying platforms. This leaves a gap between security patterns and the underlying platform.
- *Secure implementation* — Applications that have strong security requirements also require assurance that their implementation satisfies those requirements. Formal verification is one way which can provide the highest level of assurance. However, formally verifying a large complex application is very costly. Therefore, reducing the effort of verification by verifying the application design at an early stage of application development can be beneficial.
- *Security assurance* — Another way of providing assurance is through assurance cases - a structured form of informal argument. The safety-critical system community uses assurance cases extensively to structure arguments demonstrating safety claims about systems [7]. Assurance cases have also been used to support security claims. Weinstock *et al.* [8] state that a security assurance case presents arguments, supported by evidence, of claims that systems exhibit certain security properties. Security assurance cases can rely on arguments or evidence derived from the use of analytical techniques and tools. These techniques and tools can differ from one portion of the assurance case to another.

In this paper, we propose an approach that integrates the above elements at the design level. Our approach first formalises security patterns' realisation for a particular secure platform into reusable design fragments. These design fragments are verified against the security properties they intend to support. Then a design fragment can be selected and composed with an existing application design to benefit from the additional security properties embodied in the selected design fragment. We provide assurance to the improved secure design of the applications through both security assurance cases and design-level verification. In this paper, we are not concerned with the implementation of a design (so our designs are to be thought of as formal specifications in the usual software engineering sense), but rather with the construction and assurance of the design. This means that the implementation needs to be correctly implemented relative to the design (*qua* specification) and this can be done by existing means like refinement [9] (which is to be preferred, since it embodies the idea of *correct-by-construction*) or *post hoc* verification.

This approach bridges the gap between security patterns and the underlying platform by providing platform-specific design fragments. A design fragment is represented by a model that allows for design-level verification. This model consists of component specification, which includes each component's behaviours, and the interactions between components in the model. Design fragments reduce the required step of manually translating the resulting application design to a specific platform for implementation purposes. Each design fragment is associated with reusable design-level verification procedures, which are statements that are used to verify that the design fragment satisfies the desired security properties. These security properties are derived from the informal security-related claim of the security pattern. When a design fragment is selected and composed with an existing application design or another design fragment, the key challenge is the specifying the composition. We tackle this through the identification and specification of four pattern-composition primitives, which we call composition tactics. A composition can be specified by a configuration file linking the components and the primitive tactics used.

Our design fragments currently target platforms that adhere to the capability-based security model [10], although our approach should be applicable to other platforms as well.

Besides verifying individual design fragments, we also verify the overall application design to provide assurance that its security goals are satisfied. We perform design-level verification as a step to reduce the cost of potential subsequent implementation verification. In verifying the application design, we reuse the verification procedures that are associated with design fragments in order to reduce the overall effort. The use of a verification procedure for a design fragment helps identify localized problems, and the reuse of that procedure for the application design helps to ensure that the overall application achieves its security goals. We embed these steps inside a security assurance case that allows for different verification techniques for different aspects of the design. The result of verification, performed using the verification procedures of the design fragments, provides evidence for one or more security claims in the assurance case.

We demonstrate our approach by securing a Continuous Deployment (CD) pipeline [11] and show that our four composition tactics are sufficient to compose design fragments into a large secure system.

In order to make this work practical, we work under two real-life constraints. Firstly, formally verifying, from scratch, all the components and systems that we use is infeasible. Secondly, we have to work with existing components to ensure that the changes are minimal. In our work, we trust some specialized components, which can then be formally verified. Securing a real-life system using formal or semi-formal techniques is both challenging and necessary to make these techniques usable.

The contributions of this paper are two-fold: 1) the concept of a design fragment as a specialization of a security pattern for capability-based platforms; 2) a pattern-based composition approach to build and verify an application design on a capability-based platform.

II. Related Work

Capabilities, introduced by Dennis and Van Horn [10], have several advantages for building security applications. These include solving the Confused Deputy problem [12], and making it easier to apply the Principle of Least Privilege [13]. The Confused Deputy problem is a case where a program's authority is misused to perform an action that it has permission for but is not supposed to do. The Principle of Least Privilege states that only we must only grant the least permissions possible to perform a particular task. There are a number of secure platforms that adhere to the capabilities, either for single machines or distributed systems, including EROS [14], seL4 [1], Amoeba [15], and Password-capability [16].

Serscis Access Modeller (SAM) [17] is a modelling tool and notation that is intended for designing and verifying a model of an application for capability-based systems. It verifies certain security properties of the system by exploring all possible ways of access propagation. We use SAM's notation for our design. We have identified some limitations of SAM, which will be discussed later in Section III-A.

Existing security patterns can be used to guide design but their description and security property claims are highly informal [4], [18]. Prior work advocates the use of formal methods to verify security properties of patterns [18], [19]. Heyman *et al.* [19] presented an approach to formally verify security requirements for security patterns using Alloy. This includes *expectations*, which describe the expected behaviors of the component, and *residual goals*, which describe security concerns that need to be addressed but are not in the scope of the pattern. An example of a residual goal can be seen in the Authentication Enforcer pattern [20], which requires that user identifiers be unique. Ensuring the uniqueness of the identifiers is essential but is outside the scope of the pattern. Konrad *et al.* [18] allow user to define the behavior of a pattern in UML and then transform it into Promela for verification with the SPIN [21] model checker. In contrast to our approach, these approaches does not consider the security models, mechanisms and properties provided by the underlying security platform.

Some efforts [22], [23] in the model-driven community focus on developing platform-specific security profiles, used in application design to facilitate security property analysis and to map models to platform-specific code. We focus on providing reusable capability-specific design fragments for patterns that can be composed during application design. Furthermore, we support design-level analysis and assurance of systems to be built on specific platforms. These, to the best of our knowledge, are not a major concern in these previous efforts.

The link between architecture tactics and code [24] or design patterns [25] has been explored through code mining. One observation is that there is wide variation in how tactics are implemented in different situations [25]. In our work we choose one instantiation, for a capability-based platform.

Other research also advocates the importance of the underlying platform to fundamentally improve security [2], [3]. These authors propose a system architecture intended to support layered assurance, from design to implementation. This system architecture requires new hardware mechanisms to provide a more complete assurance argument, starting from the hardware. However, securing an application starting from hardware, through new hardware mechanisms, to software is very costly. We instead rely on general purpose hardware and a secure underlying platform to build verified secure applications.

In the design pattern community, there is a term 'idioms' [26], which refers to a low-level code implementing a design pattern in a particular programming language. We refer to these as language idioms. Our design fragments are specializations of security patterns in a particular platform, so they are design idioms.

Previous efforts in the design pattern community have focused on composition of design patterns. Yacoub *et al.* [27] categorize previous composition techniques as behavioral or structural. Patterns are composed based on the object interactions in the behavioral composition techniques while the structural composition techniques compose patterns by modeling their structures as class diagrams and compose them based on the structure. These efforts focus on instantiation of design in code. Furthermore, they point out that composition considering both behavior and structure will be complex. Composing security patterns to achieve security properties requires considering both behavioral and structural composition.

III. Approach

Our pattern composition approach to building secure applications is shown in Fig. 1. It consists of several steps. First, we collect security patterns from the literature [4], [6], [20], [28] and assemble them into a catalogue. Then, these patterns are analyzed and realized as concrete capability-specific design fragments (from now on referred to simply as *design fragments*). We model the design fragments using Serscis Access Modeller (SAM) [17], a modeling tool for capability-based systems. We verify the properties of these design fragments, derived from the informal security claim of the security patterns, at the design level, to check whether security properties are preserved and what assumptions must be made of the underlying platform. In the design of secure systems, providing assurance that the systems have satisfied the security requirements is essential. We justify the security properties of our application through security assurance cases. We utilise the verified design fragments in composition with each other to build up an application design that satisfies the security requirements. We propose and define four composition primitives, which we call composition tactics, to assist in composing design fragments together. Finally, we verify the application design to provide assurance that the system provides the intended properties. The verification provides evidence that helps justify our security assurance cases.

A. Serscis Access Modeller (SAM)

Serscis Access Modeller (SAM) [17] is a modelling tool and notation for defining and verifying capability-based systems. We use SAM to model our instantiation of security patterns as design fragments. A SAM model represents the distribution of capabilities across the components that make up a capability-based system. SAM has a textual representation and a graphic representation. The textual representation consists of three specifications, namely component, initial capability distribution, and security goal. In the component specification, components are represented as classes defined in a Java-like language. The behaviors of each component are represented as functions. The key characteristic of the behavior that we

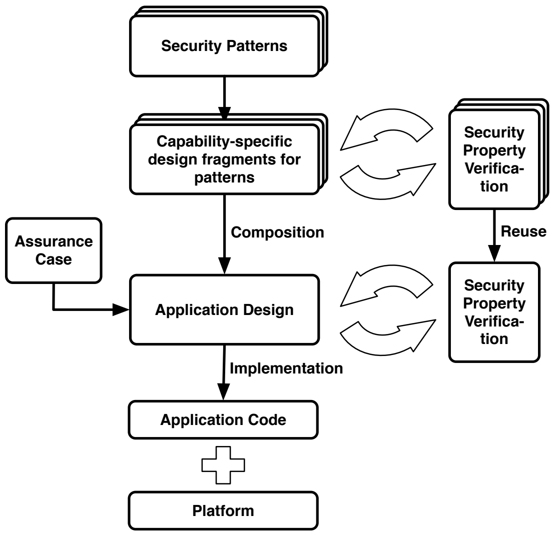

Security Patterns
Capability-specific design fragments for patterns
Security Property Verifica-tion
Composition
Reuse
Assurance Case
Application Design
Security Property Verifica-tion
Implementation
Application Code
Platform

Fig. 1: Overview of our approach

are interested in is the invocation of other components and the capability propagation through those invocations. The initial capability distribution specification defines the capabilities held by each component at the start of the system's execution. The security goal specification consists of datalog [29] rules and queries that are used to verify the system. We refer to these rules and queries as verification procedures and will be described in more detail in Section III-D.

The graphical representation of SAM shows the state of the system/model where all the possible capabilities have been propagated and shows whether there is a security violation. A model is a directed graph with nodes representing components and directed edges (arrows) representing access. An arrow pointing to a component represents holding a capability to it. Holding a capability to a component in SAM provides all the permissions, *i.e.* read, write, invoke and grant, to that component. In SAM notation, an arrow has different styles (dotted and solid) and different colors (green and black). A solid arrow signifies that the capability is obtained as part of the initial capability distribution, while a dotted arrow means that it is obtained at runtime. This is useful for tracing how access has been passed from one component to another. Consider a system with two components *A* and *B*. A green arrow from *A* to *B* means that *A* has invoked functions of *B*, while a black arrow indicates *A* does not invoke functions of *B*. Furthermore, components can be trusted (black) or untrusted (blue). Trusted components have assumptions about their behavior (*i.e.* they will only call methods as instructed) while untrusted components may call any method of the components that they have access to. In addition, untrusted components may try to pass around any capabilities they possess to the components they can access. (This notation is demonstrated later in Fig. 5.) Note that a SAM model defines the upper bounds on the behaviors of the trusted components and is an over-approximation of the real system.

One of the limitations of SAM is that it cannot model timing properties. SAM can be used to verify that an attacker does not invoke a function in a component. However, it is not the right tool to verify that the attacker will not learn about the content of a file via timing attacks. This limits the type of systems that can be modelled and verified with SAM as it cannot verify availability properties of a system.

The second limitation is that SAM cannot model dynamic deletion. Once the model is executed, a component cannot be removed. Deletion of a component can be done during design time (*i.e.* before the model is executed). It does, however, allow dynamic creation of components.

B. Design Fragments for Security Patterns

Design fragments are specializations of security patterns for capability-based platforms. We develop the design fragments by extracting important information from a security pattern and manually creating a SAM model based on this information. The extracted information includes the goal of the pattern, the actors involved (modeled as components), the main functionality of each actor, the interaction between different actors in the pattern, and the implicit assumptions of the pattern.

Each design fragment is associated with a verifiable SAM model, a reusable design-level verification procedure, and assumptions about the properties provided by the underlying platform. The verification procedure is used during the security property analysis stage (Section III-D), which will provide feedback to improve the design fragment in order to achieve the desired security properties. This feedback can be in the form of counterexamples or security violations identified during the analysis process. The assumptions can be discharged by mechanisms supported by the underlying platform or carried through as operational constraints.

C. Composition of Design Fragments

Designing a secure system may require the composition of several security patterns, each concerned with different security properties. The first step is selecting the appropriate security patterns, *i.e.* those that support the security requirements of the application and that help mitigate attacks. These patterns are then specialized into design fragments, which can then be composed with each other or applied to an existing application design (in both cases we call this *composition*). The challenge in composing design fragments is producing a design that provides the intended security properties and does not break any security properties already present in the application.

There are two things we reuse when composing design fragments. First, we reuse the structure and behavior of the design fragment. Second, we reuse the verification procedure that we applied to the design fragment including both the query and datalog rules defined for the design fragments (see Section III-D).

Each system, including design fragments, is made up of components and connections. A connection has three attributes, namely the source component, target component and access rights (capabilities). The access rights attribute signifies the capabilities the source component has over the target component. We include this attribute so that it can be generalised to cater for more systems. As SAM follows the pure capability model in which having an access means you possess all the rights (*e.g.* read, write, grant), the access rights attribute will always be the same for each connection. This is in fact over-approximating the system and we consider the worst-case scenario. A more fine-grained control on access rights, such as following the principle of least privilege, will result in a system that is at least as secure or even more secure. Consider a particular component that requires only read access to complete its tasks: if a system is secure even if this component has all the access rights, the system will at least be as secure or more secure if the component has only read rights. Therefore, we ignore this attribute in this paper.

In order to define this precisely, we make the following definitions.

Definition 1 *A* system *is $\Sigma = \langle C, N\rangle$, where C is a component set $\{c_1, c_2, ..., c_n\}$, where each c_i is a different component, and N is a connection set, $\{n_1, n_2, ..., n_m\}$, where each n_j is a different connection and where each $n_j = \langle c_i, c_k\rangle$ for some $c_i, c_k \in C$.*

Each component set is divided into three disjoint subsets, namely S, T and all the rest. S is a set of components which are the secrets or attributes that need to be protected in the system. T refers to a set of components which are trusted to have access to a secret. Each of the trusted components needs to be verified to ensure that it behaves as specified. The rest of the components are neither a secret nor components trusted to have access to a secret. We refer to systems with these two subsets defined (and one derived), and with the property that no untrusted component can access any secrets, as *acceptable systems*.

Definition 2 *An acceptable system $\Sigma_{acc} = \langle C, N, S, T\rangle$ extends a system $\Sigma = \langle C, N\rangle$, where S is a set of components that need to be protected (secrets) and T is a set of trusted components and $\forall n \in N \,.\, \forall p \in C \setminus T \,.\, \forall s \in S \,.\, n \neq \langle p, s\rangle$.*

We have identified four primitives to compose design fragments, which we call composition tactics. These tactics are *connect*, *disconnect*, *create* and *delete*. Each tactic affects either the component and/or the connection set and thus affects the verification procedures of the composite system. As we mentioned in Section III-B, each system (including design fragments) is associated with verification procedures. These procedures are datalog rules and queries that are used to verify that a system has certain intended security properties, which are specified using the template that we will define in Section III-D. Based on this template, components that are not in the trusted set cannot have access to a secret. Therefore, we can say that a secret is protected by its trusted components. To be more precise, each verification procedure acts on the component sets.

1) Connect Tactic: The aim of the connect tactic is to combine two systems together to form a larger system. This is analogous to creating an edge between two nodes in a graph. Structurally, the source component is granted a capability to the target component and invokes a particular function of the target component. For precision, we define connect in Definition 3.

Definition 3 *Connects to*
If $\Sigma_a = \langle C_a, N_a, S_a, T_a\rangle$ and $\Sigma_b = \langle C_b, N_b, S_b, T_b\rangle$ then connecting Σ_a and Σ_b means creating $\Sigma_c = \langle C_a \cup C_b, N_a \cup N_b \cup \langle c_a, c_b\rangle, S_a \cup S_b, T_a \cup T_b\rangle$, where $c_a \in C_a$ and $c_b \in C_b$ and $\langle c_a, c_b\rangle \notin N_a \cup N_b$ with the restriction that: if $c_b \in S_b$

then $c_a \in T_b$.

2) Disconnect Tactic: The disconnect tactic aims to remove a connection of source component to target component. This is analogous to removing a connection between two nodes in a graph. Structurally, a capability to the target component is revoked from the source component. (It may be that this gives us two completely disconnected systems.)

Definition 4 *Disconnects from*
If $\Sigma = \langle C, N, S, T \rangle$, then disconnecting $c_a \in C$ from $c_b \in C$ means creating $\Sigma' = \langle C, N', S, T \rangle$, where $N' = N \setminus \langle c_a, c_b \rangle$ and $\langle c_a, c_b \rangle \in N$

With regards to the verification procedure, we check whether the target component is part of the secret set of the system. If it is, we remove the source component from the target component's trusted set. If it is not an element in the secret set, the disconnect tactic does not affect the verification procedure.

3) Create Tactic: The create tactic is used to create a new component in the system. This is analogous to creating a new node in a graph. Structurally, a new component is initialized without holding any capabilities. With respect to the verification procedure, this tactic enlarges the component set of the system.

Definition 5 *Creates*
If $\Sigma = \langle C, N, S, T \rangle$, then creating a new $c \notin C$ means creating $\Sigma' = \langle C \cup \{c\}, N, S, T \rangle$.

4) Delete Tactic: The delete tactic removes a component in the system. This is analogous to deleting a node in a graph. Structurally, the target component is removed from the system at design time. However, the component needs to be isolated before the delete tactic is allowed. A component is isolated when it has no capability with respect to other components in the system and there is no capability pointing towards that component. One way to isolate a component is to revoke all the capabilities that the target component has and also revoke any capabilities that points towards this component. This tactic reduces the size of the system's component set.

Definition 6 *Deletes*
If $\Sigma = \langle C, N, S, T \rangle$, then deleting $c \in C$ means creating $\Sigma' = \langle C \setminus \{c\}, NS, T \rangle$ only if $\forall c_i \in C \;.\; \langle c, c_i \rangle \notin N \wedge \langle c_i, c \rangle \notin N$

A (partial) system representation consists of a set of components and connections. Any representation can be created by using the create and connect primitives. Two representations can be composed by selectively disconnecting and deleting components and then adding nodes and connections to accomplish the composition. Thus, once the semantics of the four primitives are specified, higher-level composition tactics between two representations can be specified in terms of these primitives. Our future work includes defining several high-level composition tactics.

D. Security Property Verification

Each design fragment has associated verification procedures. These verification procedures are datalog statements and rules that represent the security goal of the design fragment. We use Points-To analysis using Binary Decision Diagrams (BDD) [30] to verify the security properties of both the design fragments and the application design resulting from the composition of design fragments. Points-To analysis establishes which components point to (*i.e.* have access to) other components. This generates a mapping of which components have access, including access gained during execution, to other components. We assert the datalog query, that is part of the design fragment, on the mapping generated by the Points-To analysis to check whether there exists a capability/access from a component to another (presumably a secret). Asserting the datalog query ensures that the design fragment satisfies its desired security properties. For example, asserting the datalog `!hasRef(<component1>, <component2>)`, checks whether component1 can access component2. If component1 can access component2, then the desired property is not satisfied.

Verifying individual design fragments gives assurance about the security properties of the fragments. Furthermore, it helps identify localized problems to an individual design fragment before those problems propagate to the whole application design through composition. Since these fragments are composed together to form the application design, analysis needs to be performed on the whole application as well as on individual fragments. This is crucial because analyzing the composite design indicates that it retains the intended security properties, despite transformations introduced in design or composition. The analysis performed on these designs will not only provides feedback to improve the designs but also provides evidence about the security property the design fragments provide.

Reusing design fragment verification procedures for the application design might require their modification. This modification is the effect of applying the composition tactic as defined in Section III-C. This is done to reflect the goals of the application, which might differ from those of the individual design fragments. Such modification is done during composition. For instance, if the goal of a design fragment is to prevent unauthorized access to the encryption key while the goal of the application is to prevent unauthorized access to the encryption key and encrypted file at the same time, the goal of the application subsumes that of the design fragment. The datalog rule needs to be modified to check for access to both key and file simultaneously. Then, we analyze the security properties of the application to ensure that its goals are satisfied.

1) Security Goal Template: We define a template for describing the security goal of a system to make it easier to specify the verification procedures to check the intended security goal. The template requires four parameters, namely *source*, *secret*, *trusted* and a boolean value *access*. *Source* represents a set of components that we would like to perform the check on. *Secret* is a set of components that we would like to protect while *trusted* is a set of components that are trusted to have access to the secret. *Access* is a boolean that defines whether *source* should have or not have access to *secret*. The template is as follows:

```
Source has/no access to secret, except
trusted
```

The template is then represented as datalog rule, shown in Listing 1. This datalog rule is used to check whether the specified *source* has any access to the specified *secret*. Lines 2-4 exclude the trusted components from the check and lines 5-7 check whether the source has access to the secrets. Note that ',' here represents the 'AND' operator. Line 8 asserts that the security property is not breached.

Listing 1: Datalog Rule Template

```
1 secBreached(?Src,?S1,..,?Sn):-
2    !MATCH(?Src,trusted1),
3    ..............
4    !MATCH(?Src,trustedN),
5    hasRef(?Src,?S1),
6    ..............
7    hasRef(?Src,?Sn).
8 assert !SecBreached(?Src,<S1>,..<Sn>).
```

E. Assurance Case

An assurance case is a structured argument that a system has certain properties. It provides confidence that a system will function as intended through evidence and reasoning (arguments) that link the pieces of evidence to the claims. We use the Claim-Argument-Evidence (CAE) notation [8] for our assurance case. A claim represents a desired security property that the system should achieve. Argument is an explanation of how the evidence supports the claim to be true. Evidence is a proof that the system has certain property and can be obtained through testing, analysis or verification.

When building the assurance case, we determine the security property we are interested in and set that as the root claim. Then, we identify potential attacks or loopholes and make claims about how the system remains secure despite these possible attacks. We support these claims with arguments organised according to the structure of the architecture and through evidence provided by analyses.

When there is a security violation in the proposed system, we introduce verified capability-specific design fragments (Section III-B) into the system to mitigate that attack. After this, we re-analyze the security properties of the composite design. The output of security property analysis feeds into the construction of the assurance case as evidence that supports the claims about system security.

F. Designing the Application

The design of the application is driven by the items in the assurance case and the threats to the security property identified by the Points-To analysis. As we will see in the case study, we hypothesize a design and perform an analysis based on the assurance case. When a threat to the overall requirement is discovered, we choose a design fragment that should mitigate that threat. We compose the design fragment into the overall design using our primitive composition tactics and repeat the analysis.

This process is repeated until either the threat to the overall requirement is eliminated or we have exhausted all possible design fragments and possible compositions.

IV. Case Study - Securing Devops Pipeline

Many companies have embraced the concept of Continuous Deployment [11], which aims to deploy code changes to a production environment multiple times a day. Each change automatically goes through a set of tools that perform activities like integration build, deployment (and testing) to various testing environments, and deployment to production environments. We call the tool chain performing these activities a Continuous Deployment (CD) pipeline.

A key challenge in a continuous deployment pipeline is the security of the pipeline itself. First, different roles in the development team and the operation team should have different access to different parts of the pipeline. For example, a developer should not be able to deploy to production directly without her changes going through the pipeline. Certain build and test jobs can only be triggered by certain roles. Second, the testing and production environment should have total isolation. Major real world outages have happened because a component in the testing environment is accidentally connected to production database. Third, a compromised or misconfigured continuous deployment pipeline may have malicious code or unwanted debugging/experimental code that ends up being deployed to production. A typical CD pipeline is not designed with all the above security requirements in mind. Thus, the aim of the case study is to use our approach to enhance the security design of a CD pipeline satisfying the security properties derived from the above requirements. Our target platform is the Amazon Web Services (AWS) and security model is capability-based. Although AWS is not a formally verified platform, we treat its security mechanisms as trusted because of our real-life constraints.

A. Background

Continuous Deployment pipelines vary from one company to the other, depending on their existing practices. However, each of these pipelines has a common sequence of stages, which include building the code, testing the code and deploying the code to production. Each stage may require different tools and shares some commonly used tools.

Jenkins[1] is an open-source application for continuously building and testing software applications. It is typically used as a build server that performs and orchestrates several steps in a CD pipeline, which include pulling the source code, building application binary from source code, running the test suites, creating an image and storing the image into a repository or storage. In our case study, we model the storages as AWS Simple Storage Service (S3) Buckets[2].

In the application building stage, we build the code and package the binary, also called as the build artifact, into an image. This image is then stored in a storage, such as AWS S3.

There are different types of tests required during the testing stages. These include unit tests, integration tests and end-to-end tests of various lengths. They are often run inside different environments. We setup the testing environment and run the tests on the application image to ensure that the application demonstrates its intended functionalities. A deployer is required to setup the various testing environments, which includes installing the application and its dependencies inside

[1]Jenkins—http://jenkins-ci.org/

[2]S3—http://aws.amazon.com/s3/

a virtual machine, configuring and running the application, and triggering the tests. AWS Opsworks[3] is a service that helps to set up the environment and to install the application.

If the application image passes all the tests, it will then be deployed to the production environment. Usually, a release manager, who is a human operator, has to approve the deployment of a particular image into the production environment. Upon approval, a deployer will then deploy the image to the production environment.

B. Existing Security Mechanisms

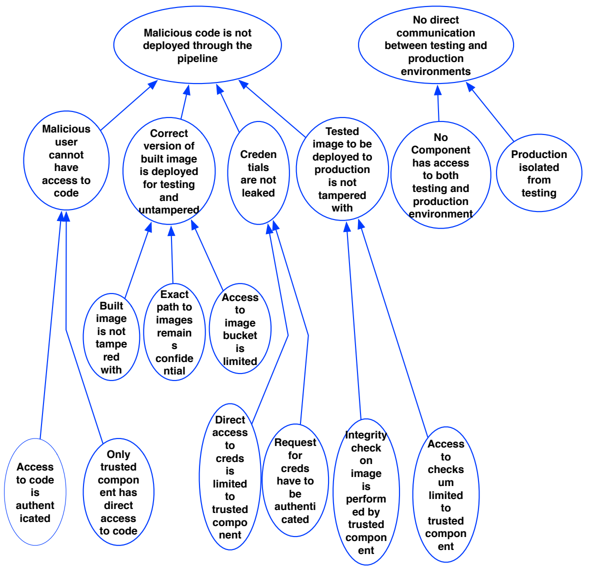
There are several sources of security mechanisms that we can utilise in our pipeline, namely from the platform, operating system and the build server - Jenkins. The AWS security model is not explicitly a capability-based model, however, it has the characteristics of such a model, and so our approach and design fragments map well to it. The AWS Identity and Access Management (IAM)[4] security mechanism binds authority to users and roles. An EC2 instance[5] running as a particular user or assuming a particular role will have the corresponding authority to access resources or invoke AWS operations. IAM also provides the ability to change or assume roles at runtime. Assuming a role creates a security credential tuple, consisting of an AWS access key id, AWS secret access key and temporary security token, which is similar to a capability since it can be passed to other instances granting them new access right. In our SAM models of Continuous Deployment pipeline, the components are either EC2 instances or AWS resources (such as S3 buckets), and the capabilities are AWS role-based security tokens.

The operating systems that are running on the EC2 instances also provide certain security mechanisms to govern what operations, such as installing an application, can be performed inside the instance. As mentioned in Section IV-A, we use AWS Opsworks to setup the environment inside the instances. Opsworks requires root access in order to perform these operations. For this reason, we disregard the security mechanisms offered by the operating system as they are overridden by Opsworks' root access.

Jenkins provides authentication and authorization mechanisms. Authentication can be achieved by creating a user database for Jenkins. Matrix-based security is commonly used as Jenkins' authorization strategy. It allows administration of specific pre-defined access rights to users or groups.

C. Securing the Continuous Deployment Pipeline

The high-level security requirements that we want to satisfy are that malicious code is not deployed through the pipeline and that there is no direct communication between components in the testing and production environments. The first high-level security requirement can be broken down into four requirements, one for each of the three stages (see Section IV-A) of the pipeline and one requirement about credentials. These requirements are the malicious user cannot have access to code, the correct version of build image is deployed for testing and is untampered with, the tested image to be deployed to production is untampered with and credentials are not leaked. These requirements can be further broken down to be more specific. The second high-level security requirement can be broken down into two requirements, namely that there is no component that has access to both production and testing environments and the production environment is isolated from the testing environment. After identifying these, we build an assurance case (Fig. 2) in order to capture and demonstrate that the pipeline satisfies its intended requirements. We claim that malicious code is not deployed through the pipeline as the root claim in our assurance case. In order for that claim to be true, we need to make sure that all the sub-claims (*i.e.* the broken down requirements) are true as well. We present a subset of the assurance case in Fig. 2; we cannot present the full case due to space limitations.



Malicious code is not deployed through the pipeline
No direct communication between testing and production environments
Malicious user cannot have access to code
Correct version of built image is deployed for testing and untampered
Creden tials are not leaked
Tested image to be deployed to production is not tampered with
No Component has access to both testing and production environment
Production isolated from testing
Built image is not tampe red with
Exact path to images remain s confide ntial
Access to image bucket is limited
Access to code is authent icated
Only trusted compon ent has direct access to code
Direct access to creds is limited to trusted compo nent
Request for creds have to be authenti cated
Integrity check on image is perform ed by trusted compon ent
Access to checks um limited to trusted compon ent

Fig. 2: Continuous Deployment Assurance Case (subset)

We start off by modelling our existing pipeline model without any security consideration and then harden the security of the pipeline. The flow of our pipeline starts off with a developer making a change in the form of a code commit. Jenkins then pulls that code commit and builds the code. Building the code results in a build artifact, which is an application binary. Jenkins then packages the build artifact into an image and stores it in an AWS S3. After that, Jenkins triggers a deployer to deploy the built image to a testing environment for testing. The deployer is a simple application that uses AWS OpsWorks to set up the testing environment, deploy and start the image inside the testing environment. The tests are then triggered. The logs from the test are then stored in an S3 bucket.

In this model, Jenkins is the main orchestrator that has access to *codeBucket*, *credsBucket*, *imageBucket*, *configBucket* and *deployer*. *codeBucket*, *credsBucket*, *imageBucket* and *configBucket* which are all AWS S3 buckets, which we model to have put and get functions. Fig. 3 shows the initial model of the pipeline. We identify *codeBucket*, *credsBucket* and *configBucket* as the secrets that we want to protect and only Jenkins (trusted) can have access to them. Furthermore, we identify *imageBucket* as a secret and we trust Jenkins and *deployer*

[3]Opsworks—http://aws.amazon.com/opsworks/
[4]IAM—http://aws.amazon.com/iam/
[5]EC2—http://aws.amazon.com/ec2/

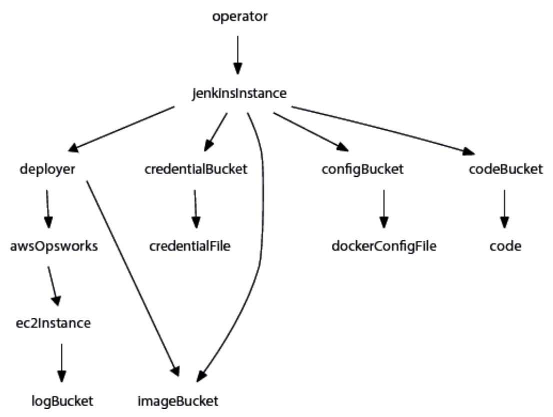


Fig. 3: Initial Design of the CD pipeline

Verification Procedure 2: Initial design

```
bktBreached(?Src,?Target):-
    !MATCH(?Src,?Target),
    !MATCH(?Src,<Jenkins>),
    hasRef(?Src,?Target).
imgBreached(?Src,?Target):-
    !MATCH(?Src,?Target),
    !MATCH(?Src,<Jenkins>),
    !MATCH(?Src,<deployer>),
    hasRef(?Src,?Target).
assert !bktBreached(?Src,<codeBucket>).
assert !bktBreached(?Src,<credsBucket>).
assert !bktBreached(?Src,<configBucket>).
assert !imgBreached(?Src,<imageBucket>).
```

to have access to it. In order to check that these properties are satisfied, we write two datalog rules, `bktBreached` and `imgBreached`, as shown in Verification Procedure 2. From here on, we refer to a Verification Procedure as a VP.

The initial model satisfies these rules, with the assumption that Jenkins is a trusted component. Satisfying these rules provides evidence to support a subset of the sub-claims in the assurance case, in particular "only trusted component has direct access to code","only a trusted component has access to an image" and "direct access to credentials is limited to a trusted component". One major weakness of this model is that we rely on the fact that Jenkins is trusted. If Jenkins is infiltrated and thus considered as untrusted, the security properties of the pipeline will not hold, given that Jenkins is the main orchestrator in the model. Therefore, we have to model Jenkins as an untrusted component to ensure that the pipeline is secure even if Jenkins is infiltrated.

We model Jenkins as an untrusted component in SAM, whereby an untrusted component may invoke any methods on the components they have access to and try to pass around any capabilities they possess to the components they can access. We aim to harden the model with Jenkins being unknown. The first step is to add authentication to authenticate the component that is retrieving the code, config file and credential file. We compose the current model with the authentication enforcer [20] design fragments. The authentication enforcer (Fig. 4) aims to create a single point of access to receive interactions of a subject and verify the identity of the subject. The security property of the authentication enforcer design fragment is that the user store should remain confidential. We write a datalog rule, shown in VP 3 to check that only *authenticationEnforcer* can have access to *userStore*.

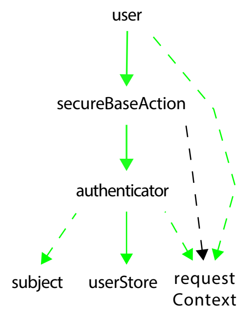


Fig. 4: Authentication Enforcer Design Fragment

Verification Procedure 3: Authentication enforcer

```
uStoreBreached(?Src,?T):-
    !MATCH(?Src,?T),
    !MATCH(?Src,<authenticationEnforcer>,
    hasRef(?Src,?T).
assert !uStoreBreached(?Src,<userStore>).
```

We need to insert the authentication enforcer design fragment in between Jenkins, *codeBucket*, *credsBucket* and *configBucket*, in order to moderate their interactions. First, we use the connect tactic, connecting Jenkins to *secureBaseAction*. Then we use the disconnect tactic to detach *codeBucket*, *credsBucket* and *configBucket* from Jenkins. As each of *codeBucket*, *credslBucket* and *configBucket* is a secret, we remove Jenkins from each of their trusted component sets. Finally, we connect *secureBaseAction* to *codeBucket*, *credsBucket*, and *configBucket*. The resulting model is shown in Fig. 5. As we trust the *secureBaseAction* component, we add it into the trusted component set of each of the connected components. We modify the affected datalog rule (`bktBreached` in this case) every time the tactic is applied. The resulting rule is shown in VP 4. We verify that the composite design satisfies the `uStoreBreached`, `imgBucketBreached` and `bktBreached` rules.

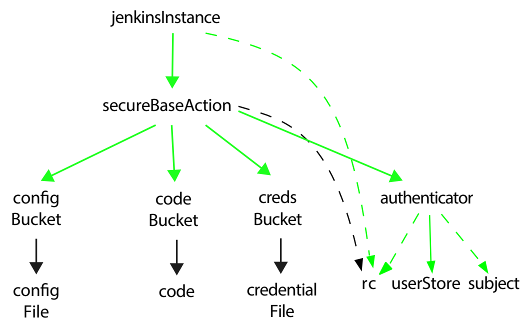


Fig. 5: Jenkins with the Authenticator Enforcer Pattern

Verification Procedure 4: Bucket (modified)

```
bktBreached(?Src,?T):-
    !MATCH(?Src, ?T),
    ~~!MATCH(?Src,<Jenkins>),~~
    !MATCH(?Src,<secureBaseAction>),
    hasRef(?Src,?T).
```

In order to ensure that the build artifact is packaged into an image correctly, we alleviate Jenkins from this duty and have a trusted image builder. Furthermore, we want to be able to detect whether or not the image is tampered with during the testing stage. We need an *imageBuilder* that will build the image, request an integrity check calculation (checksum) from *integrityChecker* and store the checksum in a database, *checksumStore*, which we want to protect and which is thus classified as a secret. In order to achieve this, we create *imageBuilder* and connect Jenkins to it. Then, we connect *imageBuilder* to *integritychecker* and *checksumStore*. As *checksumStore* is a secret and we trust *imageBuilder* to have access to it, we add the image builder to the trusted component set of *checksumStore*. Thus, we write a new datalog rule (shown in VP 5) to verify that no other component, except those in its trusted component set (*i.e. imageBuilder*), can have access to *checksumStore*.

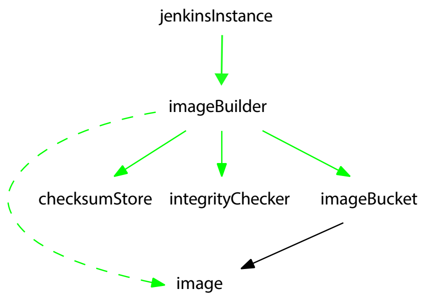


Fig. 6: Jenkins with Image Builder and Integrity Checker

Verification Procedure 5: Checksum store

```
cSumBreached(?Src,?T):-
   !MATCH(?Src, ?T),
   !MATCH(?Src,<imageBuilder>),
   hasRef(?Src,?T).
assert!cSumBreached(?Src,<checksumStore>).
```

In order to satisfy the claim that the correct version of the built image is deployed for testing and is untampered with, we need to satisfy the claim that the exact path to images remains confidential. This is to ensure that the correct version of the image is deployed. There are two ways of protecting confidentiality of data: encryption and obfuscation. The encrypted storage pattern [31] aims to harden the confidentiality of a system. It encrypts data before storing it, and the encryption key must be stored securely. This mitigates the impact of the loss of a file to an attacker, because the content of the file remains confidential, as it has been encrypted. Fig. 7 shows the capability-specific design fragment of the encrypted storage pattern. The *encryptedStorage* component has access to the *storage*, *encryptedDecrypt* and *key* components. The user (invoker) sends an encrypt command, together with the data, to *encryptedStorage*. *encryptedStorage* loads the value of the key into *encryptDecrypt* and sends an encryptData command, together with the data, to it. The encrypted data is then returned to the *encryptedStorage*, which sends it back to the invoker, and it is stored in *storage*. The security property that is of interest is the confidentiality of the data. Since the data is encrypted, access to the encryption key needs to be minimized. Thus, only *encryptedStorage* is given access to the key. The security property is reflected in VP 6. It checks whether any component in the design fragment, with the exception of *encryptedStorage*, has access to the *key*.

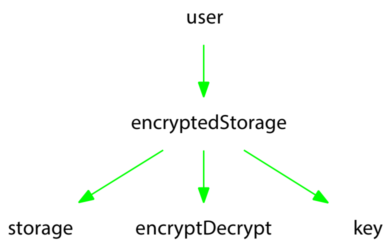


Fig. 7: Encrypted Storage Design Fragment

Verification Procedure 6: Encrypted storage

```
keyBreached(?Src,?T) :-
   !MATCH(?Src, ?T),
   !MATCH(?Src,<encryptedStorage>),
   hasRef(?Src,?T).
assert !keyBreached(?Src,<key>).
```

We compose our existing model of the pipeline with the encrypted storage design fragment by connecting *deployer* to *encryptedStorage* with the *connect* tactic. The *deployer* will generate a time-bound temporary url of the image location in a bucket, a mechanism provided by AWS S3, and then invoke the encryptData function of *encryptedStorage* to encrypt the temporary url. The temporary url not only obfuscates the real location of the image, but also put a time-bound to limit access to the image. However, we can only model the invocations and cannot check this property in SAM. The deployer then invokes AWS OpsWorks, which instructs the ec2instance to pull the image from the encrypted url and then run that image on the instance. As we are not using the *storage* component, we disconnect *storage* from *encryptedStorage* and delete the storage. Once the test is completed, the release manager will be notified and has to make a decision whether or not the latest build should be deployed to production. The release manager will perform an integrity check on the latest build image to ensure that the image is untampered with. The integrity check can be done by calculating the checksum for the build image and comparing it with the entry in the *checksumStore*. If it matches, he can approve the latest build image to be deployed to the production environment through a different deployer. Having different deployers for the testing and production environments help ensure that there is no invocation between these two environments. We assume that the EC2 instances have been already been launched by AWS OpsWorks. Fig. 8 shows the final model of the testing environment of the pipeline.

Finally, we want to ensure that there is no direct communication between the testing and production. The execution domain pattern [20] aims to restrict a process to specific resources by defining logical execution environments (domains). We define three different domains, which are testing, production and shared. The shared domain consists of utility components that are used by components in both production and testing domains. To ensure that no component in the testing domain has access to the production domain, we write a datalog rule to check this property.

In order to demonstrate this property, we connect *ec2Instance* from the testing environment to *ProductionDB* in the production environment. Fig. 9 shows that there is a red arrow, which signifies a security violation, from *ec2Instance* to *ProductionDB*. This connection is considered a bad access

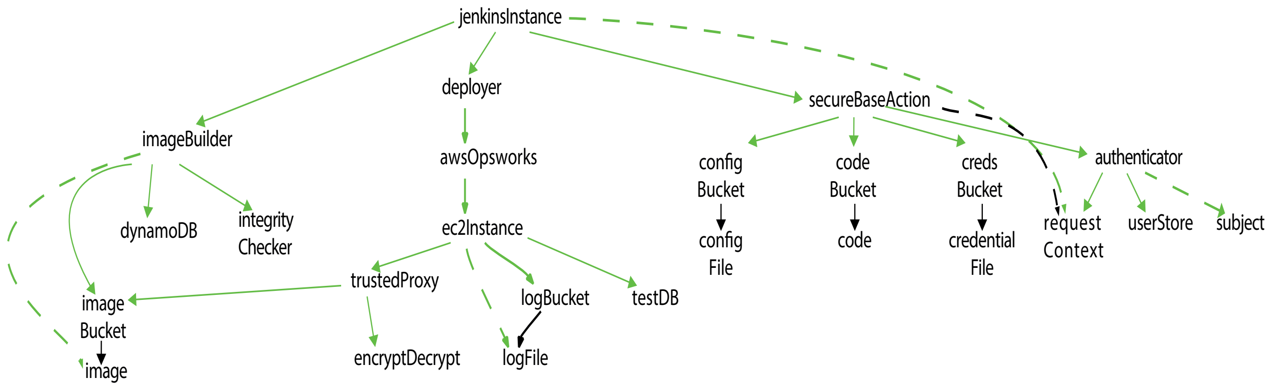


Fig. 8: The testing environment of the Continuous Deployment (CD) pipeline.

Verification Procedure 7: No cross domain access

```
haveBadAcess(?Src,?T):-
  hasRef(?Src,?T),
  hasIdentity(?Src, "Testing"),
  hasIdentity(?T, "Production").
```

and is caught by the datalog rule specified above (VP 7).

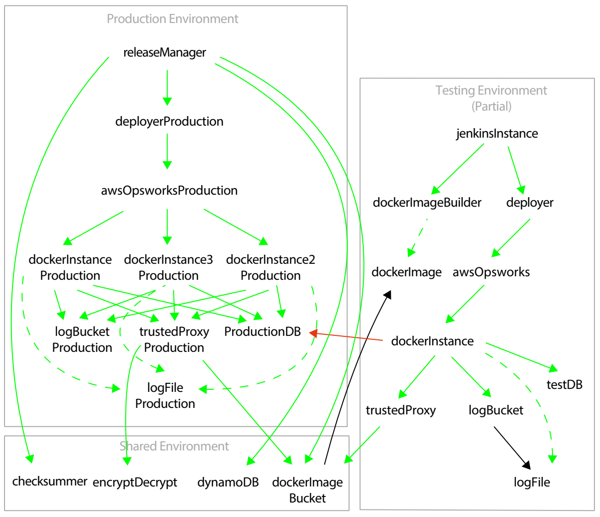


Fig. 9: Testing to Production disallowed

The final pipeline design satisfies its security properties even though Jenkins and all the buckets and ec2instances are untrusted. Each verification procedure of the pipeline provides evidence to support the subclaims in the assurance case that we present in Fig. 2. Due to space limitations, we show only a subset of the assurance case.

D. Path to implementation from model

One of the steps we perform in securing the CD pipeline is to split off the image builder from Jenkins and perform an integrity check on the image built. We will host the image builder on an independent EC2 instance. The image builder will offer image building as a service and gets the required files as parameters of the service invocation. As it will be hosted in an AWS EC2 instance, there are three security mechanisms that we can utilize, namely the operating system security, firewall (security group[6]) and IAM related permissions. AWS also provides a mechanism to administer inter-AWS services' permission. As the image builder will need to write to an S3 bucket and DynamoDB[7], we create a role for the image builder in IAM. This role is associated with a security policy that gives it permission to perform put object (write) on a particular S3 Bucket, identified by a unique Amazon Resource Name (ARN)[8] and permission to perform actions on DynamoDB, identified by an ARN as well. After this is set up, we launch a new EC2 instance and grant it this IAM role. This is currently the only way to trigger this functionality. After the instance is launched, we setup the security group. The security group can be used to restrict incoming and outgoing traffic to and from the instance. We open one port to allow invocation of the service and limit the request only to those that originate from the Jenkins instance private IP Address. Furthermore, we should allow outgoing traffic to the integrity checker private IP to enable image builder to invoke the integrity checker service. Finally, there is the security provided by the operating system that is running on the EC2 instance. However, we ignore the operating system security mechanisms of the instances in this paper. AWS recommends that password-only authentication should be disabled and to use SSH authentication. Other AWS services that we intend to use are AWS storage service (S3) and NoSQL database service (DynamoDB). Both of these can be secured by specifying appropriate IAM policies to allow/deny certain actions to be performed on these services.

V. Discussion

The four primitive tactics, which are defined in Section III-C, have been exercised and utilized multiple times in the case study. Complex security-critical systems can be built using combinations of these primitive tactics. However, due to the tactics being very primitive, it might be tedious to perform commonly used combinations, such as the proxy tactic. The proxy tactic is a higher-level tactic that helps to insert a new component in between two connected components. This can be achieved through multiple application of connect and disconnect tactics. Building a catalog of these higher-level tactics is crucial as it may ease the composition approach. We will pursue this in our future work.

We rely on the assurance case being correct for the composite design properties. The assurance case helps in

[6]Security group—http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html

[7]DynamoDB—http://aws.amazon.com/dynamodb/

[8]ARN—http://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html

determining which security properties contribute to the final properties and thus affects the verification procedure to be run on the composite design. However, the assurance case might be modified to provide more coverage or strengthen a property. The consequence of the modification will affect the properties of the system.

One of the common pitfalls for model-checking verification is lack of scalability, due to state explosion. However, we perform design-level verification which is on smaller models compared to code-level verification. Berndl *et al.* [30] have demonstrated that Points-To analysis using BDD can scale to verify code of large systems.

The trusted components need to be rigorously tested (ideally formally verified) to minimize (or remove) issues due to programming errors. This is essential as the security of the system will most likely fail if a trusted component is compromised. Our aim is to identify security issues as early in the software development lifecycle as possible. We acknowledge that there are some properties that are important but not meaningful to reason about at the design level, such as denial of service. Access propagation is a property that it is meaningful to analyse and reasoned about at the design level.

A potential weakness of design-level verification is that the security property might break at the code level or physical level. This can be due to misunderstanding of the design, coding errors, or to implementations exposing information that had been abstracted in the design. One way of mitigating this is to generate code for the architectural framework connecting different components. Using such a framework can provide high-level assurance, assuming that the generated code comes with assurance that the refinement to code does not break the design-level security property. The component code must also be correct. Attacks at the physical level can also violate assumptions made by design abstractions. For example, covert channel attacks can break confidentiality properties. Alternative mitigations and arguments are required to provide assurance that a system can resist such attacks, but we have not directly addressed this issue in this paper.

Security patterns are our source for the definition of capability-specific design fragments. However, as patterns are informal and ambiguous, it can be difficult to know exactly what properties are being claimed and what should be verified. Moreover, it is difficult to compare a formal representation of the pattern as a design fragment, to an informal one (textual). A design fragment for a security pattern is one version of the pattern specialized for a specific platform. We do not claim that a design fragment of a pattern for a platform is the only possible representation of the pattern for that platform.

VI. Future Work

Our future goal is to prove that our tactics are compositional. This means that we have to prove that if two subsystems are known to be secure and they are combined using the connect tactic, then the resulting system will, by construction and necessarily, be secure too. And the same needs to be proved of the other tactics. Once this is done we no longer need the (perhaps expensive) verification process any more.

Furthermore, we can drop the requirement that our design fragments are verified using the same verification method (since there will be no need for such things). This will make specifying (or designing) secure systems much more flexible.

Our definitions of the tactics in section III-C makes a start in this direction. At this stage, (and we provide all this for a strong indication of our future direction) if we take the previous definitions, then we believe that we can make claim to the correctness of the accompanying conjectures.

Definition 7 *Protected by (version one)*
For any $\Sigma_{acc} = \langle C, N, S, T\rangle$, any $s \in S$ is protected by T iff $\forall p \in C \setminus T \,.\, \langle p, s\rangle \notin N$

Definition 8 *Protected by (version two)*
For any $\Sigma_{acc} = \langle C, N, S, T\rangle$, any $s \in S$ is protected by T iff $\forall c \in C \,.\, \langle c, s\rangle \in N \implies c \in T$

Conjecture 9 *Definitions 7 and 8 are equivalent.*

Conjecture 10 *In acceptable systems, all secrets are protected by trusted components, i.e. in $\Sigma_{acc} = \langle C, N, S, T\rangle$, every element of S is protected by T, and we say S is protected by T.*

Conjecture 11 *In acceptable systems, connect is compositional. This means that if two systems being connected have their respective secrets protected by their respective trusted sets, then performing connect will preserve the property whereby for the composed system the components in its secret set are protected by components in its trusted set.*

We call the tactic defined in Definition 3 a *strong connect*, because this tactic preserves the security property of both systems being connected under the restrictions that: (1) a component in the trusted set cannot make a new connection to a secret of the other; and (2) a component that is not a secret and not trusted cannot make a new connection to a secret. It does not make much sense to have an atomic trusted component, *i.e.* a component that is trusted for nothing. The second restriction helps preserve the protected property that is defined in Definitions 7 and 8. So, we can guarantee that the security properties of composed systems are preserved when strongly connected.

However, having a strong connect means that there are many other possible compositions that in fact we would like to have but which the strong connect does not allow due to its restrictions. To account for that, we might need a *weaker* connect and that will be formalised in our future work. For the weaker connect, we drop the restrictions that is a feature of strong connect to allow for more connections. The price we pay, of course, is that we lose compositionality, *i.e.* we cannot *guarantee* that security properties will be preserved.

There are some subtleties of the connect tactic. These bring up questions such as: 1) Should secrets be allowed to connect to secrets? 2) Are cycles allowed between secrets and trusted components? 3) Should a trusted component of one system automatically become trusted by another system when they are connected together? By Definition 3, we allow secrets to be connected to secrets and trusted components. For the third question, we have taken a simplified approach in the above definition by disallowing a trusted component to connect to a secret. This is a reason why our *strong connect* guarantees that the secrets will always be protected by their trusted components. However, this is also a reason why the *strong* *connect* is restrictive. Performing a *weaker connect* might then allow the alteration of the trusted set of a secret, based on user input (i.e. user wants to connect a component to a secret, and trust the component to have access to the secret).

Other future work includes: building a catalogue of higher-level tactics, which are combinations of the primitive composition tactics defined in Section III-C is crucial; we intend to provide more automation to compose two systems together, utilizing the composition tactics; finally, we would like to measure the performance overhead incurred by the security mechanisms and would therefore require the implementation of the secure Continuous Deployment pipeline.

VII. Conclusion

Building secure applications requires significant expertise and effort. To alleviate this problem, we introduced the concept of reusable verified design fragments and the compositional use of them in secure application design. These design fragments package security patterns and platform features and are verified to provide assurance about their security properties. Our approach enables the composition of the design fragments through four primitive tactics and verifies the composed design against desired security properties, which are presented in an assurance case. We demonstrated our work by securing a real-life example of a Continuous Deployment (CD) pipeline. Through the example, we have demonstrated how we can harden aspects of the initial system design by iteratively improving security in response to increasingly severe attacks, using the four composition tactics.

ACKNOWLEDGEMENTS

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: Formal verification of an os kernel," in *the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: ACM, 2009.

[2] P. G. Neumann and R. N. M. Watson, "Capability revisited: A holistic approach to bottom-to-top assurance of trustworthy systems," in *the Fourth Annual Layered Assurance Workshop*, Texas, USA, 2010.

[3] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *the 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[4] J. Yoder and J. Barcalow, "Architectural patterns for enabling application security," in *the 4th Conference on Patterns Language of Programming (PLoP)*, Washington, DC, USA, 1997.

[5] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Boston, MA, USA: Addison-Wesley Professional, 2012.

[6] E. Fernandez-Buglioni, *Security Patterns in Practice: Designing Secure Architectures Using Software Patterns*, 1st ed. Hoboken, NJ, USA: Wiley Publishing, 2013.

[7] R. Bloomfield and P. Bishop, "Safety and assurance cases: Past, present and possible future an adelard perspective," in *Making Systems Safer*, C. Dale and T. Anderson, Eds. Springer London, 2010, pp. 51–67.

[8] C. Weinstock, H. F. Lipson, and J. Goodenough. (2013) Arguing security - creating security assurance cases.

[9] J. Derrick and E. Boiten, *Refinement in Z and Object-Z: Foundations and Advanced Applications*, ser. Formal Approaches to Computing and Information Technology. Springer, May 2001. [Online]. Available: http://www.cs.ukc.ac.uk/pubs/2001/1200

[10] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, March 1966.

[11] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Boston, MA, USA: Addison-Wesley Professional, 2014.

[12] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 4, 1988.

[13] J. Saltzer and M. Schroeder, "The protection of information in computer systems," *the IEEE*, vol. 63, no. 9, 1975.

[14] J. S. Shapiro, J. M. Smith, and D. J. Farber, "Eros: A fast capability system," in *the Seventeenth ACM Symposium on Operating Systems Principles*, ser. SOSP '99. New York, NY, USA: ACM, 1999, pp. 170–185.

[15] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: A distributed operating system for the 1990s," *Computer*, vol. 23, no. 5, pp. 44–53, May 1990.

[16] M. D. Castro, R. D. Pose, and C. Kopp, "Password-capabilities and the walnut kernel," *The Computer Journal*, vol. 51, no. 5, 2008.

[17] T. Leonard, M. Hall-May, and M. Surridge, "Modelling access propagation in dynamic systems," *ACM Transactions on Information and System Security (TISSEC)*, vol. 16, no. 2, September 2013.

[18] S. Konrad, B. H. C. Cheng, L. A. Campbell, and R. Wassermann, "Using security patterns to model and analyze security requirements," in *the Requirements Engineering for High Assurance Systems (RHAS)*, 2003.

[19] T. Heyman, R. Scandariato, and W. Joosen, "Reusable formal models for secure software architectures," in *the 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, Helsinki, Finland, 2012.

[20] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating Security and Systems Engineering*. Hoboken, NJ, USA: Wiley Publishing, 2006.

[21] G. Holzmann, "The model checker spin," *Software Engineering, IEEE Transactions on*, vol. 23, no. 5, 1997.

[22] J. Jürjens, "UMLsec: extending UML for secure systems development," in *the 5th International Conference on The Unified Modeling Language*. London, UK: Springer-Verlag, 2002.

[23] T. Lodderstedt, D. A. Basin, and J. Doser, "Secureuml: A uml-based modeling language for model-driven security," in *the 5th International Conference on The Unified Modeling Language*, London, UK, 2002.

[24] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar, "A tactic-centric approach for automating traceability of quality concerns," in *the 34 International Conference on Software Engineering (ICSE)*, 2012.

[25] M. Mirakhorli, P. Mäder, and J. Cleland-Huang, "Variability points and design pattern usage in architectural tactics," in *the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012.

[26] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Hoboken, NJ, USA: Wiley Publishing, 1996.

[27] S. Yacoub and H. Ammar, *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*. Boston, MA, USA: Addison-Wesley Professional, 2003.

[28] C. Steel, R. Nagappan, and R. Lai, *Core security patterns: best practices and strategies for J2EE, Web services, and identity management*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[29] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 1, no. 1, 1989.

[30] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee, "Points-to analysis using BDDs," in *the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, New York, NY, USA, 2003.

[31] D. M. Kienzle, M. C. Elder, D. Tyree, and J. Edwards-Hewitt, "Security patterns repository, version 1.0," 2002.