2012 Joint Working Conference on Software Architecture & 6th European Conference on Software Architecture

An Architectural Approach for Cost Effective Trustworthy Systems

Ihor Kuz, Liming Zhu, Len Bass, Mark Staples, Xiwei Xu
NICTA, Level 5, 13 Garden St, Eveleigh NSW 2015, Australia
School of Computer Science and Engineering, University of NSW, NSW 2052, Australia
firstname.lastname@nicta.com.au

Abstract— **We describe a research program on design techniques to enable the cost-effective construction of trustworthy systems. The focus is on single-machine systems that can be formally verified to provide desired system-wide security and safety properties. Such systems are designed as compositions of small trusted components and large untrusted components whose behaviour is constrained by an underlying formally verified OS kernel. Past work has shown that building these systems is possible, now we wish to do so in a cost effective way. A key part of doing this is to design, as early as possible, an architecture that can provide the required trustworthiness properties. The research program envisions methods, models, analyses, and patterns to create and formally analyse such architectures. We present initial work on this program and discuss the gaps and research questions that will shape future research.**

Keywords - trusted systems, trusted connectors, security properties, confidentiality, trusted patterns.

I. Introduction

Constructing trustworthy large-scale systems is one of four grand challenges in information security enumerated in 2004 [1]. Trustworthiness encompasses functional correctness and also includes properties of security (confidentiality, integrity, availability) and safety (it also relates to reliability and maintainability, which we do not currently consider) [2]. Software is *trusted* when its failure can break specified trustworthiness properties.

The reliance of trustworthy systems on the specification of individual properties is understood in the trustworthy systems community but not so well outside of that community. As an example, you may trust an individual to keep a secret but not to be able to leap tall buildings in a single bound. Keeping a secret and leaping a building are two distinct properties with different techniques for verifying and different levels of trust in the verification techniques.

A common strategy for building trustworthy systems is to identify the system's *trusted computing base* (TCB), separate it from the rest of the codebase, ensure that it is correct and show that specific use of the TCB will guarantee trusted properties for the whole system. Ensuring correctness of the TCB can be done in a variety of ways, including testing, certification, code synthesis, and formal verification. The highest levels of assurance about the trustworthiness of a software system rely on formal verification, but in practice, when it comes to functional correctness, it is infeasible to formally verify the source code of large codebases. The current limit is around 10,000 lines of code. However, system design techniques can compose large-scale systems from few, small trusted components and larger, untrusted components [3].

We build on previous work developing the formally verified operating system microkernel, seL4 [4], and designing and verifying systems built on top of seL4 [5]. The context of this work is a single computer running software based on a formally verified foundation. This previous work demonstrates the feasibility of developing large trustworthy systems. Our goal now is to enable the construction of such trustworthy systems in a cost effective manner. A key part of doing this in a development project is to design, as early as possible, a system architecture that can provide the required trustworthiness properties. This architecture then drives the further development and verification of the whole system.

In the remainder of this paper, we lay out our vision of how an architect would construct a trustworthy system. We follow that with a description of existing results that lead us to believe that our vision is realisable. We then present preliminary results of an analysis with AADL and SPIN, ending with a description of the gaps that must be closed to realise our vision.

II. Our Vision of How an Architect would Construct a Trusted System

Our vision is that an architect should be able to design for formally verified trustworthiness properties just as they design for other properties, such as modifiability, performance, or interoperability. This will involve design tradeoffs between the levels of these properties and the levels of assurance in these properties. Figure 1 shows the process of designing, implementing, and verifying a trustworthy system. In this paper we focus on the architectural aspects of this process (steps 1-5), however, we describe the whole process for completeness and to provide sufficient context.

In order to accomplish our vision, we see the architect needing the following:

- A library of trusted patterns for each property, where a trusted pattern is a pattern that has been shown (e.g., using formal analysis) to provide a given property;
- Cost models to enable the estimation of the cost portion of a cost/benefit analysis for the chosen properties;
- Tools that support the verification of chosen properties for a proposed system architecture.

The architect starts by choosing the properties that must be provided by the system, and constructs an architecture taking advantage of the library of trusted patterns (step 1 of Figure 1). The cost of building an assurance case for the chosen properties can be estimated using the architect's existing cost modelling tools and the cost models associated with the chosen properties. The cost/benefit of the chosen trustworthy properties are traded off against other desired properties of the system (step 2 of Figure 1).

The architect then represents the architecture in an architecture description language and uses formal analysis (such as model checking) to verify that the architecture could support the desired properties. The verification relies on assumptions provided by an underlying trusted computing infrastructure, such as the formally proven microkernel (step 3 of Figure 1). This part of the process is iterative, as results of analyses and tradeoffs may require re-architecting the system to achieve the right balances (step 4 of Figure 1).

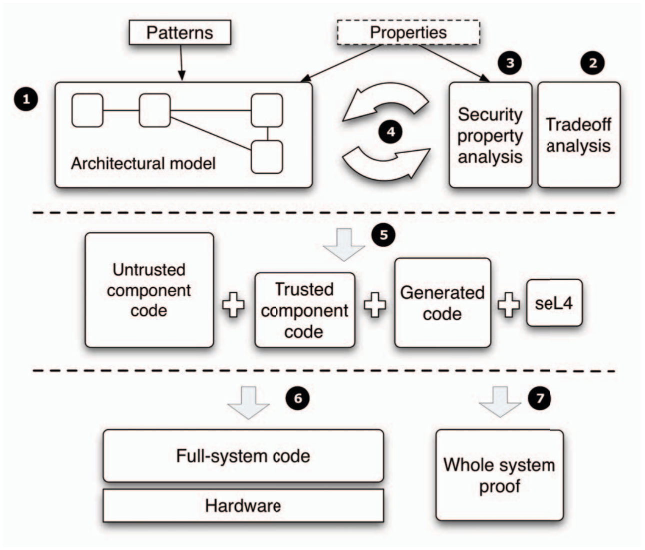

Figure 1. The process of developing a trustworthy system

Next, the development team elaborates the architecture by partially synthesising components, glue code and proofs (step 5 of Figure 1). These are combined with implementations of trusted and untrusted components to produce the final system (step 6 of Figure 1) and a proof (using formal verification) of the desired properties of the whole system (step 7 of Figure 1). The architecture, verified by the formal analysis and constrained by the underlying trusted infrastructure, dictates the isolation and communication between components. To achieve formal verification of the whole system, the architect will only need to verify the correctness of each new trusted component, and, due to the use of the trusted patterns and previously verified infrastructure, will be able show how the trusted components lead to the system's trustworthiness properties.

III. EXISTING RESULTS

We describe a pattern for ensuring a confidentiality property as an example of how we see the general problem being solved. It is a generalisation of previous work in our group of designing, implementing and verifying sel4-based systems [5]. The TCB in such systems includes seL4, the architecture framework, and all trusted components. Our interest here is twofold. First, the use of a security pattern in a system that has been implemented and whose design has been formally verified is evidence that our research agenda is achievable. Second, we identify the assumptions we are making about the trustworthiness of the infrastructure without which the pattern does not guarantee its security property.

This pattern, as illustrated in Figure 2, is for a connector between several data sources (e.g., networks) and a data sink, and ensures mutual confidentiality of the sources.

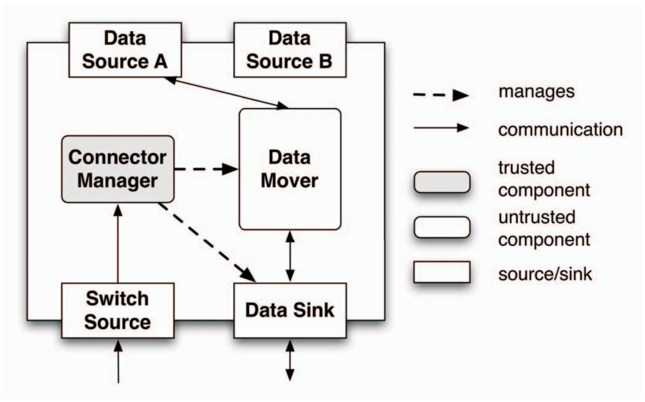

Figure 2. A trusted pattern for confidentiality

The pattern has two components and four data sources or sinks. The components are:

- An untrusted Data Mover that is responsible for moving data from a source to the sink. As implemented in [5] this was a virtualised instance of a Linux-based network router (which consists of millions of lines of code).
- A trusted Connector Manager. This is the component that manages the internal connections and the permissions that enable us to show the confidentiality property. In the implementation this was a small component consisting of approximately 1500 lines of code.

There are two data sources (A and B), one data sink, and one source of switching signals (Switch Source).

The connector is responsible for routing traffic from the active source to the Data Sink. The active source can be switched between Data Source A and B by sending a signal on the Switch Source. The security property is that the connector will not cause any messages entering (or leaving) through Data Source A to flow to Data Source B and vice versa.

The action of the trusted Connector Manager (assuming an existing connection between Data Source A, the untrusted Data Mover, and the Data Sink) is as follows.

- Delete the existing instance of the untrusted Data Mover.
- Zero out all memory that was accessible by the untrusted Data Mover (including any memory in the Data Sink).
- Create a new instance of the untrusted Data Mover.
- Allow it to read from Data Source B and write to the Data Sink.

As long as the base operating system can be trusted to enforce isolation between the components in the system, to enforce appropriate memory access control, and to perform system setup correctly, we can show that the desired confidentiality property is achieved. Note that while the whole connector has a large codebase, its TCB remains small (only the Connector Manager) and is within the bounds of formal verification.

The runtime cost of using this pattern instead of just the untrusted Data Mover is the performance overhead of starting, stopping and running the Data Mover (for example, in the implementation this was the cost of virtualising Linux). The benefit is that now the whole connector can be trusted to enforce the confidentiality property and can be used as a trusted component to construct larger systems.

IV. AADL AND PROMELA/SPIN

In our initial exploration of appropriate languages and tools for modelling and analysis of trusted patterns we have used AADL (Architecture Analysis and Design Language) [6] and PROMELA [7] (for the SPIN model checker). Here we outline this work.

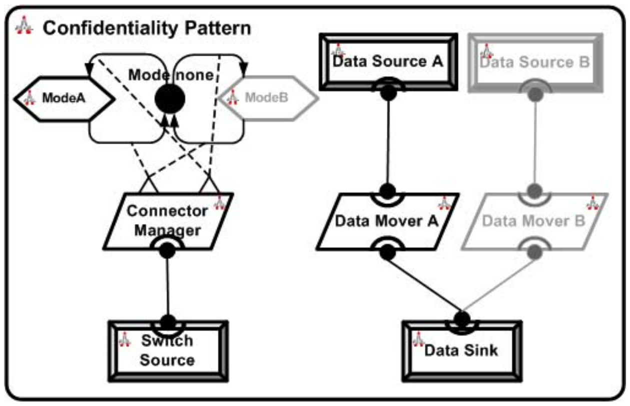

Figure 3. AADL model of the confidential connector pattern

AADL introduces formal modelling concepts for distinct components and their interactions. It provides the *mode* abstraction, to explicitly define different configurations of components and connections. Figure 3 shows the AADL model of the pattern from Figure 2. We used modes to represent the dynamic configuration of components in the pattern. Mode A (black) and Mode B (grey) represent states in which Data Source A is connected or Data Source B is connected, respectively. Mode None represents the transitional state while switching between Data Sources, where the Data Mover instance has been deleted.

Transitions between the modes are triggered by events emitted by the Connector Manager. The triggering of Connector Manager events is described using the AADL behaviour annex [8], as shown in Listing 1. This defines the behaviour of the Connector Manager as a state machine. Events (*switch[1]* and *switch[2]*) cause mode changes as part of the state transitions. The state transitions themselves are triggered by events from the Switch Source (*switch_in*).

Listing 1

```
annex behavior_specification {**
states
  s0: initial complete state;
  s_clear: state
transitions
  s0-[switch_in?(x)]->s_clear{switch_clear!;};
  s_clear-[]->s0{switch[x]!;};
**};
```

Our goal in modelling this pattern is to analyse whether it can support the confidentiality property of preventing messages from one Data Source flowing to the other. We employed model checking using PROMELA and the SPIN tool.

In our PROMELA model, all components in the pattern are modelled as active processes running concurrently in the initial state, except for Data Mover, which is started and stopped dynamically. The Connector Manager can create a Data Mover by running its process, and delete a Data Mover by terminating its process. Synchronous channels are used for control flow (*connect*/*disconnect* messages), while asynchronous channels are used for data flow. In particular, memory that is accessible to the Data Mover is modelled by a buffer of the asynchronous data channels. Clearing such memory is modelled by flushing the appropriate buffer. The behaviour of the Connector Manager trusted component is modelled by a set of explicitly sequenced instructions, shown in Listing 2, while the behaviour of untrusted components are modelled using a *do* clause that allows the components to perform allowed actions in any sequence, as shown in Listing 3.

Listing 2

```
active proctype CM(){
  mtype receive;
  do
  ::SS_CM?receive ->
    if
    ::(receive == A) -> ctrl_CM_DM!disconnect;
    ctrl_CM_B!disconnect;ctrl_CM_T!disconnect;
    flush(data_DM_B); flush(data_B_DM);
    flush(data_DM_DS); flush(data_DS_DM);
    run DataMover(data_DM_A, data_DM_R);
    ctrl_CM_A!connect; ctrl_CM_DS!connect;
    mode = A;
    ::(receive == B) -> …;
    fi
  od
}
```

Listing 3

```
active proctype DataSourceA(){
          int data;
idle:     ctrl_CM_A?connect; goto connected;
connected:
  do
  :: ctrl_CM_A?disconnect -> goto idle;
  :: data_DM_A?data; data_A = data;
     assert(data_A!=b);
  :: data_A_DM!a;
  od
}
```

Given the PROMELA model we use SPIN to simulate the system's execution and verify the confidentiality property. We represent the confidentiality requirement as a taint property[1]. We use two constants *a* and *b* to model the data from Data Source A and Data Source B respectively. The Data Sink either forwards the data it received from the Data Source (*a* or *b*), or sends its own data (modelled by a third constant *s*). Basic assertions (e.g., shown in line 8 of Listing 3) are used to check whether A is tainted by data flows from B or vice versa.

We have performed an analysis of this model, with the result that the confidentiality property relies on the specific behaviour of the trusted Connector Manager. For example, if we flush the memory *after* creating the Data Mover (lines 9 and 10 in Listing 2), the assertion would be violated.

The conclusions from our initial modelling and analysis attempt are the following. Firstly, AADL on its own does not support the necessary modelling of component behaviour. The behaviour annex models the low-level behaviour, but does not directly support the dynamic manipulation (creation and deletion) of the components. Secondly, AADL does not have sufficient capability to enable the type of formal analysis that we wish to perform and, consequently, we relied on a mapping of AADL to PROMELA for subsequent analysis of the confidentiality property.

V. Further Research

We have presented some of the preliminary results in pursuing our research vision. In order to successfully achieve our goals we will need to also do the following:

- Expand the trusted pattern set. We have described with our example pattern one of a collection of patterns for trustworthy systems. We must develop a library of such patterns, building on prior work by others (e.g. [9]).
- Develop technical guidance and cost guidance to enable the architect to select and perform architectural analysis. Existing work on modelling and analysing architecture for security properties [10][11] will be used where possible.
- Develop techniques to compose trusted patterns and to reuse previously verified properties of these patterns in analysing the resulting architecture.
- Define how architectural analyses relate to the code-level formal verification of the resulting implementation. In particular, the models used for architectural analysis should correspond to models used for verification of the resulting system and thus drive subsequent proof efforts.
- Implement and formally verify the glue code by which components are composed. This is an important part of the work, since, on the one hand, it provides a framework with implementation-specific properties that we can rely on when analysing and verifying the architecture, and on the other hand, it is one of the links from the architecture analysis to the full system verification.
- Create a cost model for the development and formal verification of individual trusted components and of whole systems. We have previously developed a detailed descriptive model of the middle-out development and verification process used for seL4 [12] and will build on this experience in developing an architecture-level model.

[1] This is a weaker form of the confidentiality property. Colleagues are working on a stronger form based on non-interference, which we aim to adopt at a later date.

VI. Summary

Constructing large trustworthy systems has been a goal for at least four decades. In this paper we have outlined our research program for making this goal a reality and presented initial results that we have achieved in pursuing this goal for specific security properties. It is feasible to leverage a formally verified TCB to establish assurances for large systems containing large untrusted components. We believe software architecture can enable system design and analyses to make this more cost effective. While we currently tackle single-computer systems, solving the problems discussed in this paper will form the basis for examining even more complex, distributed, trustworthy systems.

Acknowledgments

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] S. W. Smith & E. H. Spafford, "Grand challenges in information security: Process and output", *IEEE Security & Privacy*, Jan/Feb, 2004.

[2] A. Avizienis, J. C. Laprie, B. Randell & C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing", *IEEE Transactions on Dependable and Secure Computing*, 1(1), 2004.

[3] J. Alves-Foss, W. S. Harrison, P. Oman & C. Taylor, "The MILS architecture for high-assurance embedded systems", *International Journal Of Embedded Systems,* 2(3-4), 2007.

[4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch & S. Winwood. "seL4: Formal verification of an OS kernel", *Proceedings of 22nd SOSP*, ACM, 2009.

[5] J. Andronick, D. Greenaway & K. Elphinstone, "Towards proving security in the presence of large untrusted components", *Proceedings of the 5th International Conference on Systems Software Verification*, USENIX, 2010.

[6] *Architecture Analysis & Design Language (AADL)*, AS5506 Rev. A, Society of Automotive Engineers, 2009.

[7] G. J. Holzmann, *The SPIN Model Checker*, Addison-Wesley, 2004.

[8] R. B. Franca, J. P. Bodeveix, M. Filali & J. F. Rolland, "The AADL behaviour annex – Experiments and roadmap", *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, IEEE Computer Society, 2007.

[9] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann & P. Sommerlad, *Security Patterns: Integrating Security And Systems Engineering,* Wiley, 2006.

[10] J. Bau & J. C. Mitchell, "Security modeling and analysis", *IEEE Security & Privacy,* 9(3), 2011.

[11] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, M. Roveri, & R. Wimmer, "A model checker for AADL", *Proceedings of the 22nd International Conference on Computer Aided Verification,* 2010.

[12] J. Andronick, R. Jeffery, G. Klein, R. Kolanski, M. Staples, H. Zhang, L. Zhu, "Large-scale formal verification in practice: A process perspective", *Proceedings of the 34th International Conference on Software Engineering*, 2012.