Towards a Verified Component Platform

Matthew Fernandez, Ihor Kuz, Gerwin Klein, June Andronick
NICTA and the University of New South Wales, Sydney, Australia
first-name.last-name@nicta.com.au

Abstract

This paper describes ongoing work on a new technique for reducing the cost of assurance of large software systems by building on a verified component platform. From a component architecture description, we automatically derive a formal model of the system and a semantics for the runtime behaviour of generated inter-component communication code. We can prove wellformedness properties of the architecture automatically and provide a framework in which users can reason about their component code and its behaviour. By leveraging the isolation properties and communication guarantees of a formally verified platform, correctness arguments for critical components will be able to be derived independently and composed together to reason about system-level correctness.

1. Introduction

Formal software verification, while clearly desirable for critical software systems, is an expensive and time intensive process; prohibitively so for systems on the order of one million lines of code [Moore 2003]. The code that can affect correctness, safety or security of the system is referred to as the trusted computing base (TCB). It typically includes both operating system and some userspace functionality, and easily exceeds this measure. To gain trust in such systems, we must either reduce their size or reduce the cost of assurance.

Component-based software engineering has been used successfully to make the design and implementation of large systems more tractable [Szyperski 1997], and it is already used as a methodology for managing complexity in many critical domains [Broy et al. 2007]. Using the same technique to divide and conquer the verification challenge has the potential to dramatically reduce the cost and increase the effectiveness of assurance.

By constructing an operating system and its userspace applications such that critical functionality and non-critical functionality are placed in separate components, verification of the correctness of the system as a whole can be decomposed into the correctness of only the critical components, their isolation from the non-critical components, and their interaction. The feasibility of building real systems with this methodology has been demonstrated in a small case study in [Andronick et al. 2010]. The TCB of such a componentised system is then reduced to the operating system kernel, the critical components, initialisation code and the component platform. The code of the non-critical components may be untrusted and even assumed to be malicious. This paper focuses on the component platform's correctness, and its link to correct initialisation. The long-term goal is to then build on these guarantees to perform full-system verification.

In addition to the obvious benefits of code reuse and design flexibility, a component platform is typically comprised of significantly less code than the non-critical components it supports. With a component platform that has been verified once, non-critical components can be freely added, removed or reconfigured without requiring verification changes, as long as they do not change the interaction with trusted components. Systems can reuse verified components from previous systems and retain their locally proven guarantees; something that is not possible with code reuse across monolithic systems.

In this paper, we report on our initial progress in building a formally verified component platform. In particular, we build on top of the seL4 microkernel, with the aim of later leveraging its functional correctness [Klein et al. 2009] and isolation properties [Sewell et al. 2011, Murray et al. 2013] to build componentised systems with strong isolation guarantees. As a capability-based kernel, seL4 is ideal for high assurance environments because it is possible to derive upper bounds on the authority in a given system. We have adapted an existing component platform, CAmkES [Kuz et al. 2007], that can be used to design static userspace systems on seL4. Using a component platform to abstract the low-level mechanisms of seL4 facilitates design and imple-

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.

PLOS '13 November 03-06 2013, Farmington, PA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2460-1/13/11. . . $15.00
http://dx.doi.org/10.1145/2525528.2525535

mentation of a system, and provides well-defined boundaries between components by using the address space isolation features of seL4.

CAmkES is part of the TCB in such a component system and the correctness of the system as a whole is predicated on the correctness of CAmkES. In the work presented here, we address the correctness of this platform by automatically deriving formal specifications of (i) the architecture of a system and (ii) runtime semantics representing the behaviour of the generated communication stubs. We also generate, from the formal architecture specification, a description of the initial configuration, which can be used as input to an existing tool [Boyton et al. 2013] to correctly and automatically initialise the system into this configuration. All the formal specifications are in the form of Isabelle/HOL [Nipkow et al. 2002] theories that are machine checked.

Ongoing work focuses on additionally generating two proofs about artefacts produced by the CAmkES platform. The first is intended to show that the generated communication stubs refine the derived runtime semantics of a given system and the second is intended to show that the generated initial configuration of the system corresponds to the derived architecture semantics.

The intended outcome is a verified component platform, supporting full system verification to the level of source code, by allowing users to decompose the reasoning about correctness of large systems.

Section 2 provides a brief introduction to the CAmkES platform and an example of generated code. Section 3 gives an overview of the code and theories we generate. More specific details on the generation of architecture semantics, runtime semantics and initialisation specification are described in Sections 4, 5 and 6 respectively. Section 7 discusses the limitations of our approach and future directions.

2. CAmkES

CAmkES systems consist of components and connections between these components. Each component has a collection of incoming and outgoing interfaces and connections must link an outgoing interface of one component with an incoming interface of another component. There are three types of interface supported by CAmkES: procedures (function calls), events (asynchronous signals) and dataports (shared memory). The interfaces at either end of a connection must be of the same type. Connections specify a communication mechanism, but abstract this from the user.

In this section we will use the CAmkES system depicted in Figure 1 as a simple example of interacting components. The system involves three component instances; `c` reads values from a key-value store in `s`, while `f` interposes on their communication and prevents the values associated with certain keys from being read. From a user's perspective, `c` invokes functions of `f`, which in turn invokes functions of `s`.

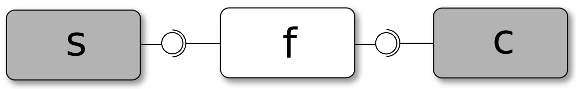

Figure 1. Example CAmkES system

Treating `c` and `s` as untrusted components, we wish to prevent `c` from reading certain values from `s`. By placing `f` as a trusted component between `c` and `s` we would like to be able to show our property using only the behaviour of `f` and the structure of the system. Figure 1 can be seen as exemplary of a system involving trusted and untrusted components with a desirable security property.

```
 1 procedure Lookup {
 2   int get_value(in int key);
 3 }
 4 component Client {
 5   control;
 6   uses Lookup inf;
 7 }
 8 component Filter {
 9   provides Lookup external;
10   uses Lookup backing;
11 }
12 component Store {
13   provides Lookup db;
14 }
15 assembly {
16   composition {
17     component Filter f;
18     component Client c;
19     component Store s;
20     connection RPC one(from c.inf,
21       to f.external);
22     connection RPC two(from f.backing,
23       to s.db);
24   }
25 }
```

Figure 2. Example ADL

CAmkES systems are described using an architecture description language (ADL). The syntax of the language is straightforward, and Figure 2 shows the ADL corresponding to the example from Figure 1. It describes an interface type, `Lookup` (lines 1-3), three component types, `Client` (lines 4-7), `Filter` (lines 8-11) and `Store` (lines 12-14) and a system involving instances of these types (lines 15-25). `Client` has an active thread of control (line 5) and makes calls to functions from the interface `Lookup` (line 6). `Filter` contains an implementation of the interface `Lookup` (line 9) and also makes calls to an outgoing instance of this interface (line 10). `Store` contains a further implementation of the same interface (line 13). The system, defined in the `composition` block (lines 16-24), contains an instance of each component type (lines 17-19) and two connections, one from `c`'s outgoing interface to `f`'s incoming interface (lines 20-21) and the other from `f`'s outgoing interface to `s`'s incoming interface (lines 22-23). Both connections use a remote procedure call (RPC) mechanism. The `assembly` element (line 15) designates the top-level entity of the system.

The first stage of compiling a CAmkES system uses a code generator that parses the given ADL description and generates glue code. This glue code implements the interfaces as described in the ADL in terms of underlying platform mechanisms. This glue code is then compiled together with the user's component code to form a collection of userspace applications. These applications are configured by an initialiser task that runs as the first application after the kernel has booted.

To make this more concrete, Figure 3 shows the glue code that is generated for the interface `inf` in `c` for the ADL example in Figure 2. The glue code consists of an implementation of the interface function, `get_value`, prefixed by the name of the interface, `inf`. The implementation marshals the parameters of the function into an in-memory buffer (lines 4-9) and uses seL4's inter-process communication mechanism to send this data via a capability to an endpoint, `EP` (lines 11-13). This endpoint is connected to component `f`, that provides the interface. The glue code waits for a response to this communication (line 14), unmarshals the return value from the function (lines 16-19) and then returns control to the user's code (line 21). The corresponding glue code for `f` (omitted here) is the inverse of this, namely unmarshalling the parameters, calling the user's implementation of `get_value` and then marshalling the response. The majority of CAmkES glue code performs similar mechanical operations, which makes code generation relatively straightforward.

```
1   int inf_get_value(int key) {
2     void *buf = BUFFER_BASE;
3
4     /* Marshal the method index */
5     int call = 0;
6     buf = marshal(buf, &call, sizeof(int));
7
8     /* Marshal all the parameters */
9     buf = marshal(buf, &key, sizeof(int));
10
11    /* Call the endpoint */
12    unsigned int msglen = buf - BUFFER_BASE;
13    kernel_Send(EP, msglen);
14    kernel_Wait(EP, NULL);
15
16    /* Unmarshal the response */
17    buf = BUFFER_BASE;
18    int r;
19    buf = unmarshal(buf, &r, sizeof(int));
20
21    return r;
22  }
```

Figure 3. Example glue code

The net result is that the user writes component code as if each component was directly accessing the functionality of other connected components. The implementation of the glue code can be modified without necessitating changes to the user's component code. This abstraction also facilitates porting the component code to other operating systems.

3. Generated Artefacts

The design of our framework is depicted in Figure 4, with the user-provided elements shaded in light grey and the generated artefacts in dark grey. We provide an overview of the process here and then provide further details in the following sections.

From an architecture description, such as that in Figure 2, we generate a formal architecture semantics, glue code, a formal semantics for the glue code and an initialisation specification in the form of a capability distribution. The user provides code for each component and specifications for the execution of trusted components. The component code, glue code and capability distribution are compiled to form a bootable userspace image. We can obtain a semantics for the system as a whole by composing the architecture semantics, the glue semantics and the user's component specifications.

The larger, bidirectional arrows labelled ① and ② in the figure represent formal refinement proofs. We expect the user to provide a proof ② that their trusted component specifications correspond to their component code or to axiomatise this. We intend to automatically provide all other steps and proofs in the diagram. The resulting artefacts provide evidence for the correspondence between the system semantics and the binary, as indicated by arrow ③. The formal composition implied by ③ is ongoing work, whose initial steps are described in [Andronick et al. 2010].

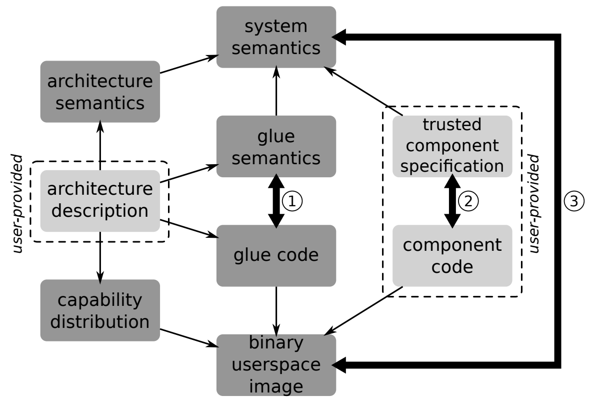

Figure 4. System artefacts

4. Architecture Semantics

The interpretation of ADL during the compilation process described in Section 2 is informal in the sense that we know what the resulting system looks like, but there are no constraints on what is a valid description, what it means for two components to be connected and what properties such a system has. Essentially the code generator has rules for the syntax of the language, but no semantics ascribed to the syntactic elements. Part of the work presented here has been defining a formal semantics for ADL. This gives formal meaning to an ADL description in Isabelle/HOL. Figure 5 shows a fragment of the Isabelle/HOL theory corresponding to the ADL in Figure 2. The definition of the procedure Lookup describes a list containing a single method that accepts an integer input parameter (lines 5-7) and returns an integer value (lines 2-3). The component `Client` is defined to have a thread of control (line 10) and a single outgoing interface of type Lookup (line 11). Definitions are also generated for `Filter`, `Store` and the `assembly` block, but have been omitted for reasons of space.

Both specifications, informal and formal, express the same information, but the formal specification, with its attached semantics, can be used to reason about structural properties of the system. We have developed a tool for automatically generating these formal specifications from ADL.

```
1  definition Lookup :: procedure
2  where "Lookup ≡ [⦇ m_return_type =
3      Some (Primitive (Numerical Integer)),
4        m_name = ''get_value'', m_parameters =
5        [⦇ p_type = Primitive (Numerical Integer),
6           p_direction = InParameter,
7           p_name = ''key'' ⦈] ⦈]"
8
9  definition Client :: component
10 where "Client ≡ ⦇ control = True,
11     requires = [(''inf'', Lookup)],
12     provides = [], dataports = [], emits = [],
13     consumes = [], attributes = [] ⦈"
```

Figure 5. Formalised ADL

ADL allows a component system to be described that can never be implemented, or one that violates the assumptions of CAmkES. To constrain the possible system descriptions we have defined wellformedness predicates for each ADL element. The constraint on wellformedness of a `composition` block is shown in Figure 6. It expresses basic properties required of the composition, including that the composition must contain at least one component with a thread of control (line 6), all components themselves are wellformed (lines 13-14) and all connections link valid interfaces of the same type (lines 16-17). During specification generation, we also generate a proof that the specification is wellformed, guaranteeing that the system described can be created.

5. Runtime Semantics

To reason about the execution of components and properties that are true of the system at runtime, we need a model of what the glue code actually *does*. We generate such a model, shown as the glue semantics in Figure 4, in a formal imperative language for concurrent processes with synchronous message passing. The language allows us to express component execution as a series of local state modifications and communication steps. Since component code

```
1  definition
2    wellformed_composition :: "composition ⇒ bool"
3  where
4    "wellformed_composition c ≡
5   (* This system contains ≥ 1 active component. *)
6      (∃x ∈ set (components c). control (snd x)) ∧
7   (* All references resolve. *)
8      refs_valid_composition c ∧
9   (* No namespace collisions. *)
10     distinct (map fst (components c) @
11       map fst (connections c)) ∧
12  (* All components are valid. *)
13     (∀x ∈ set (components c).
14       wellformed_component (snd x)) ∧
15  (* All connections are valid. *)
16     (∀x ∈ set (connections c).
17       wellformed_connection (snd x))"
```

Figure 6. Wellformed composition

is user-supplied, the local state of components is simply a type variable `'cs` in the Isabelle formalisation. The global execution state of the system is then a map from component instance name `inst` to `comp` which encodes the current program counter of that instance and this local state:

```
type_synonym 'cs global_state =
  "(inst, 'cs comp × 'cs local_state) map"
```

To encode glue code behaviour, we generate program fragments in the formal language above for each interface function in each component. In addition to the user's components specified in ADL, we model events and dataports as artificial additional components in the system, in order to capture their associated state and semantics. The state of an event is a boolean variable indicating whether the event is pending or not. The state of a dataport is a map from natural numbers to values, representing the contents of the shared memory at any given address offset. Thus the local state of a generalised component is a value of either the user-supplied state type in the case of a user component or one of the previously described types in the case of an artificial component:

```
datatype 'cs local_state
  = Component 'cs
  | Event bool
  | Memory "(nat, variable) map"
```

We generate definitions describing the effect on the system state of invoking each exposed piece of glue code functionality. Figure 7 shows the definitions corresponding to the glue code from Figure 3. The first definition, `Call_Client_inf_get_value`, provides a general description of the interface invocation in the `Client` component type. It accepts a connection to send on, `ch`, a projection function, $\texttt{key}_P$, to extract the integer to send from the local state and an embedding function, `embed`, for updating the local state with the return value. The definition performs two steps, a `Request` (lines 7-9), that corresponds to the marshalling and send system call in the glue code, and a Response (lines 10-13), that corresponds to the wait system call and return. The second definition, Call_c_inf_get_value, curries the first with a function returning a precise outgoing connection, one, to provide a definition specific to the glue code for the instance c.

```
1  definition Call_Client_inf_get_value ::
2   "(Client_channel ⇒ channel) ⇒
3    ('cs local_state ⇒ int) ⇒
4    ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
5    (channel, 'cs) comp"
6  where "Call_Client_inf_get_value ch key_P embed ≡
7    Request (λs. {⦇q_channel = ch Client_inf,
8     q_data = Call 0 (Integer (key_P s) # [])⦈})
9     discard ;;
10   Response (λq s. case q_data q of Return xs ⇒
11    {(embed s (case hd xs of Integer v ⇒ v),
12    ⦇a_channel = ch Client_inf, a_data = Void⦈)}
13    | _ ⇒ {})"
14
15 definition Call_c_inf_get_value ::
16  "('cs local_state ⇒ int) ⇒
17   ('cs local_state ⇒ int ⇒ 'cs local_state) ⇒
18   (channel, 'cs) comp"
19 where "Call_c_inf_get_value ≡
20   Call_Client_inf_get_value
21    (λc. case c of Client_inf ⇒ one)"
22
```

Figure 7. Glue code semantics

The user can compose these generated definitions with their own definitions of the behaviours of a trusted component. These user-provided definitions represent the trusted component specification in Figure 4. The user can also choose to omit such definitions, e.g. for untrusted components. In this case, we generate a broad non-deterministic default behaviour that encompasses any local state modification and any invocation of its interfaces, representing a potentially maximally misbehaving component.

6. System Initialisation

Sections 4 and 5 have described how we derive an abstract formal model of the system architecture and code. We want to use this model to reason about properties like information flow enforcement and have these properties hold at the source code implementation level. It is not sufficient to reason about the component and glue code. We need a trusted path from an ADL specification to a correctly initialised access control configuration of the underlying kernel in the running system.

To see why this is necessary, consider an initialisation process that correctly configures a given CAmkES system, but additionally leaves capabilities allowing unconnected components to communicate. With this, an information flow property derived on the abstract model can easily be violated in the implementation because the model does not capture this extra communication path.

To ensure this does not happen, and to guarantee we run what the architectural semantics describe, we utilise capDL, a language for describing capability distributions on seL4 [Kuz et al. 2010]. The aim of the language is to describe complete access control system configurations by capability distributions alone. Such capDL descriptions are proved to map to a corresponding access control policy [Boyton et al. 2013], that can then be used to reason about the integrity and confidentiality of the system [Sewell et al. 2011, Murray et al. 2013].

From the ADL of a component system, we automatically generate a capDL description of that system, providing an automated, trusted path from component specification to capability description. The code generation assigns a separate address space to each component, and provides each with the appropriate seL4 capabilities to access its declared communication channels, but not more than these. We then use an existing tool for correctly and automatically initialising seL4 systems from capDL descriptions [Boyton et al. 2013].

7. Limitations, Status and Future Work

The main limitation of our system is that CAmkES can only be used to describe and instantiate static system architectures. Systems that involve creating and destroying components at runtime cannot be described in ADL. We believe this restriction is acceptable, as the high assurance domains we are targeting typically use static system designs for other reasons already [Broy et al. 2007].

A practical limitation of using CAmkES is that it is not widely used, with the result that few existing systems can take advantage of this work. However, given seL4's ability to be used as a hypervisor, it is possible to run virtualised instances of commodity operating systems, such as Linux, as components in a CAmkES system. This allows for incremental deployment, with non-critical legacy applications running in an untrusted virtualised environment, and critical components extracted to run in the verified environment.

The system initialisation described in Section 6 provides a formal theorem that the running system matches the capability distribution provided to the initialising task. To extend this to correspondence between the running system and the component architecture description, we intend to define a formal relation between the derived architectural model and the seL4 capDL model. Automatically producing a proof of correspondence at the time of code generation should also be feasible. This will achieve a formal chain between formal architecture description and running system configuration.

While our semantics of glue code can already be used to reason about the behaviour of component systems, the connection between these semantics and the C implementation is currently unverified. That is, the translation from the architecture description to glue code is trusted to be correct. To remove this assumption, we intend to show refinement between the runtime semantics and the generated code, using existing infrastructure for deriving formal semantics from C code [Tuch et al. 2007, Greenaway et al. 2012]. We intend to fully automate this refinement proof, since the generated glue code is system specific.

Our approach will be to show refinement for basic blocks that appear as patterns in the glue code and then to generate a proof for specific glue code as a composition of these blocks. Essentially, the glue code is a combination of marshalling/unmarshalling code and interaction with the underlying OS kernel. The primitives of these operations can be verified once manually, and then composed automatically. Work on this is currently in progress.

8. Related Work

Designing high assurance systems by isolating software components is an approach closely related to MILS [Alves-Foss et al. 2006] and separation kernels [Rushby 1981; 1984]. There is also existing work proposing running untrusted services on a microkernel as part of a larger, trusted system [Hohmuth et al. 2004]. We are essentially utilising seL4 as a highly flexible separation kernel, but we are additionally aiming to give support for formally connecting such MILS architectures directly to the final implementation of the system, including generated component platform code.

Our approach of verifying the code generation post hoc is a form of translation validation [Pnueli et al. 1998, Necula 2000]. While the problems we are tackling relate to efforts like the CompCert compiler [Leroy 2006], the code we are verifying is much more constrained. In particular the generated code contains no recursion and all loops can be bounded statically. While this makes the code verification simpler, we are pursuing an extensible system verification framework that composes with user-provided code and specifications, which is an aspect that has not yet appeared in implementation verification projects outside the proof-carrying code paradigm [Necula 1997].

Correctness of component-based systems is not a novel idea in itself and there is much existing work on this subject [Yellin and Strom 1997, Giannakopoulou et al. 2002, Plasil and Visnovsky 2002, Adamek 2003]. This existing work overwhelmingly focuses on the correctness of specific component code and interactions. While this is important for system correctness, it assumes correctness of the component platform itself. While previous work has acknowledged this weakness [Fisler and Krishnamurthi 2005], it still remains an outstanding problem. This assumption is what our work targets.

9. Summary

We have described a way of designing and implementing component-based systems suitable for high assurance environments. In particular, we have shown how to generate formal semantics for architecture configurations and generated communication glue code in the CAmkES component platform, and we have laid out our vision of how this can be combined with automatic refinement proofs to achieve a formally verified component platform that supports local reasoning about user-provided components.

If we are to continue building complex, large software systems for safety- and security-critical domains, we believe the only way to provide cost effective trust in such systems is to decompose the verification challenge. Using the process we have described in this work, verification of a system can be performed piecewise and, by formally isolating large untrusted components, full verification of systems consisting of millions of lines of code can be made tractable.

References

J. Adamek. Static analysis of component systems using behavior protocols. In *OOPSLA*, pages 116–117, Anaheim, CA, USA, Oct 2003.

J. Alves-Foss, P. W. Oman, C. Taylor, and S. Harrison. The MILS architecture for high-assurance embedded systems. *Int. J. Emb. Syst.*, 2:239–247, 2006.

J. Andronick, D. Greenaway, and K. Elphinstone. Towards proving security in the presence of large untrusted components. In G. Klein, R. Huuck, and B. Schlich, editors, *5th SSV*, Vancouver, Canada, Oct 2010. USENIX.

A. Boyton, J. Andronick, C. Bannister, M. Fernandez, X. Gao, D. Greenaway, G. Klein, C. Lewis, and T. Sewell. Formally verified system initialisation. In Lindsay Groves, Jing Sun, editor, *15th ICFEM*, Queenstown, New Zealand, Oct 2013. Springer.

M. Broy, I. H. Krüger, A. Pretschner, and C. Salzman. Engineering automotive software. *Proc. IEEE*, 95:356–373, 2007.

K. Fisler and S. Krishnamurthi. Decomposing verification around end-user features. In *VSTTE 2005*, pages 74–81. Springer, Oct 2005.

D. Giannakopoulou, C. S. Păsăreanu, and H. Barringer. Assumption generation for software component verification. In *17th ASE*, pages 3–12, Edinburgh, Scotland, UK, Sep 2002.

D. Greenaway, J. Andronick, and G. Klein. Bridging the gap: Automatic verified abstraction of C. In L. Beringer and A. Felty, editors, *3rd ITP*, volume 7406 of *LNCS*, pages 99–115, Princeton, New Jersey, Aug 2012. Springer. ISBN 978-3-642-32346-1.

M. Hohmuth, M. Peter, H. Härtig, and J. S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *11th SIGOPS Eur. WS*, Leuven, Belgium, Sep 2004.

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 *22nd SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM. doi: 10.1145/1629575.1629596.

I. Kuz, Y. Liu, I. Gorton, and G. Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems*, 80(5): 687–699, May 2007.

I. Kuz, G. Klein, C. Lewis, and A. Walker. capDL: A language for describing capability-based systems. In *1st APSys*, pages 31–36, New Delhi, India, Aug 2010.

X. Leroy. Formal certification of a compiler back-end, or: Programming a compiler with a proof assistant. In J. G. Morrisett and S. L. P. Jones, editors, *33rd POPL*, pages 42–54, Charleston, SC, USA, 2006. ACM.

J. S. Moore. A grand challenge for formal methods: A verified stack. In B. K. Aichernig and T. Maibaum, editors, *Formal Methods at the Crossroads: from Panacea to Foundational Support*, pages 161–172. Springer, 2003.

T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symp. Security & Privacy*, pages 415–429, San Francisco, CA, May 2013. ISBN 10.1109/SP.2013.35.

G. C. Necula. Proof-carrying code. In *24th POPL*, pages 106–119, Paris, France, Jan 1997.

G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94, Vancouver, British Columbia, Canada, 2000.

T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Trans. Softw. Engin.*, 28(11):1056–1076, Nov 2002.

A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *4th TACAS*, pages 151–166, Lisbon, Portugal, Mar 1998. Springer.

J. Rushby. A trusted computing base for embedded systems. In *Proceedings of 7th DoD/NBS Computer Security Conference*, pages 294–311, Sep 1984.

J. M. Rushby. Design and verification of secure systems. In *8th SOSP*, pages 12–21, Pacific Grove, CA, USA, Dec 1981.

T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *2nd ITP*, volume 6898 of *LNCS*, pages 325–340, Nijmegen, The Netherlands, Aug 2011. Springer. doi: http://dx.doi.org/10.1007/978-3-642-22863-6_24.

C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, Essex, England, 1997.

H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *34th POPL*, pages 97–108, Nice, France, Jan 2007. ACM.

D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Trans. Progr. Lang. & Syst.*, 19(2):292–333, Mar 1997.

This material is based on research sponsored by Air Force Research Laboratory and the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-12-9-0179. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory, the Defense Advanced Research Projects Agency or the U.S. Government.