The Oz-E Project: Design Guidelines for a Secure Multiparadigm Programming Language

Fred Spiessens and Peter Van Roy

Université catholique de Louvain,
Louvain-la-Neuve, Belgium
{fsp, pvr}@info.ucl.ac.be

Abstract. The design and implementation of a capability secure multiparadigm language should be guided from its conception by proven principles of secure language design. In this position paper we present the Oz-E project, aimed at building an Oz-like secure language, named in tribute of *E* [MMF00] and its designers and users who contributed greatly to the ideas presented here.

We synthesize the principles for secure language design from the experiences with the capability-secure languages *E* and the W7-kernel for Scheme 48 [Ree96]. These principles will be used as primary guidelines during the project. We propose a layered structure for Oz-E and discuss some important security concerns, without aiming for completeness at this early stage.

1 Introduction

The Oz language was designed to satisfy strong properties, such as full compositionality, lexical scoping, simple formal semantics, network transparent distribution, and so forth. Security, in the sense of protection against malicious agents, was not a design goal for Oz. In this paper, we give a road map for making a secure version of Oz, which we call Oz-E. Our approach is not to add security to Oz, but to remove insecurity. We start with a small subset of Oz that is known to be secure. We add functionality to this subset while keeping security. The ultimate goal is to reach a language that is at least as expressive as Oz and is secure both as a language and in terms of its implementation. It should be straightforward to write programs in Oz-E that are secure against many realistic threat models.

Structure of the Paper

This paper is structured into five parts:

- Section 2 summarizes the basic principles of language-based security, to set the stage for the rest.
- Section 3 discusses a possible structure for Oz-E and a migration path to get there.

P. Van Roy (Ed.): MOZ 2004, LNCS 3389, pp. 21–40, 2005.

– Section 4 discusses some concerns that will influence the design of Oz-E.
– Section 5 gives some practical scenarios with fragments of pseudocode.
– Section 6 summarizes the paper and the work that remains to be done.

Readers unfamiliar with the terminology of capabilities (authority, permission, etc.) are advised to have a look at the glossary at the end of the paper.

2 Basic Principles of Language-Based Security

We distinguish between three kinds of principles: mandatory, pragmatic, and additional. All principles serve a common goal: to support the development of programs that use untrusted modules and entities to provide (part of) their functionality, while minimizing their vulnerability to incorrectness and malicious intents of these entities.

To avoid excess authority the Principle of Least Authority (POLA) – explained in another paper in this book [MTS05] – has to be applied with scrutiny and supported by the language. POLA is not just about minimizing and fine-graining the authority that is directly provided to untrusted entities, but also about avoiding the abuse – by adversaries or incorrect allies – of authority provided to relied-upon entities.

The latter form of abuse is known in the literature as the *luring attack* or the *confused deputy* [Har89]. A deputy is an entity that gets authority from its clients to perform a task. A confused deputy is a deputy that cannot tell the difference between its own authority and the authority it is supposed to get from its clients.

Figure 1 (left) shows what can go wrong with Access Control Lists (ACL's). The client wants the deputy to write its results to the file "`file#123`". Assume the deputy has the authority to write to this file. Should the deputy write to the file? It may be that the client is abusing the deputy to write somewhere that the client itself should not be allowed to write. There is no simple way to solve this problem with ACL's.

Figure 1 (right) shows how capabilities solve the problem. Instead of providing a mere designation, the client now provides a capability that bundles the

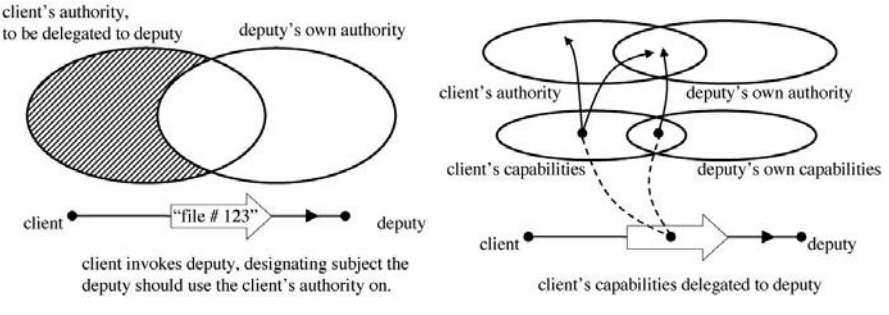

Fig. 1. ACL's vs. Capabilities

designation with the authority to write the file. The deputy can use this capability without second thoughts, because the client can no longer trick the deputy into writing into a place it should not.

2.1 Mandatory Principles

This section explains the principles that form the *minimum necessary* conditions to *enable* secure programming, following the object-capability approach [MS03].

No Ambient Authority. All authority is to be carried by capabilities: unforgeable entities that combine designation with permissions. To enable the individual entities to control the propagation of authority, the language has to cut off every other way of getting authority. All entities come to live with no default authority, and capabilities can only be acquired in the following ways:

- By endowment and parenthood (as defined in Sect. 7).
- By introduction: an entity can initiate an exchange of capabilities with another entity, by exerting a permission of a capability that designates the other entity (Sect. 7).

The language thus has to make sure that no authority can be acquired in any other way, whether via globally or dynamically scoped variables, or via memory probing and forging. This means that the language has to be *purely lexically scoped* and *completely memory safe*.

No Authority Generation. The availability of two capabilities can result in more authority than the simple sum of both authorities. This phenomenon is called "authority amplification".

Except for the purpose of authentication – which will be handled in section 2.2 – authority amplification is very dangerous and should be avoided when

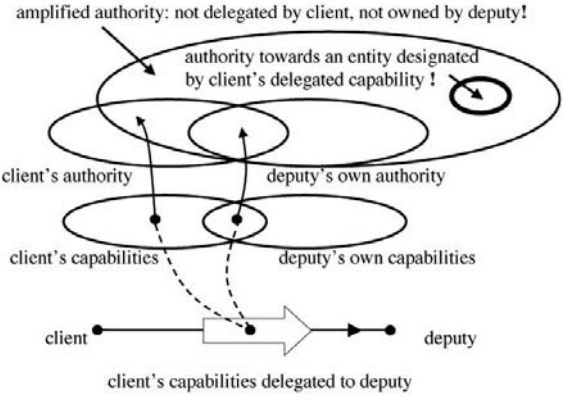

Fig. 2. Authority amplification can confuse deputies

possible. It is as if "ambient authority" becomes available to an entity, thereby turning the entity into a confused deputy, because the extra authority is not provided by the client nor by the deputy.

Figure 2 shows what can happen. The client passes a capability to the deputy. This capability designates an entity. Authority amplification will increase the authority that the deputy has over this entity. This is shown by the small bold oval. In that way, designation and authority have effectively become separated again, just like with ACL's, and the same problems arise.

Since the language will represent capabilities as data abstractions (e.g. objects, procedures, abstract data types (ADT's)) it must make sure that these abstractions can handle delegation of authority appropriately. Unbundled abstractions (ADT's), that provide operations separately from values, can lead very easily to the creation of deputies that are confused by authority amplification.

To minimize the opportunities for deputies to be confused by authority amplification, a secure language must provide all access to system resources as bundled data abstractions that *completely encapsulate* their internal state. Authority amplification is defendable only in cases where normal capabilities would not suffice.

2.2 Pragmatic Principles: Promoting Secure Programming

With the basic principles in place, all essential control of authority distribution and propagation becomes available to programmers, and they can now – in principle – start building entities that will perform reliably in collaboration with untrusted ones. However, it is not enough that Oz-E *enables* secure programming, it should also make secure programming *feasible in practice* and consequently *favor* secure programming (Sect. 7) as the default.

Defensive Correctness. The dominant pattern of secure programming, which the language must make practical, is that clients may rely on the correctness of servers, but that servers should not rely on the correctness of clients. In other words, a server (any "callee" in general) should always check its preconditions. A client (any "caller" in general) may rely on the server, if it has the means to authenticate the server. The usefulness of this pattern has emerged from experience with E and its predecessors.

In traditional correctness arguments, each entity gets to rely on all the other entities in the program. If any are incorrect, all bets are off. Such reasoning provides insufficient guarantees for secure programming. To expect programmers to actually check all preconditions, postconditions, and invariants is not a realistic approach either. *Defensive correctness* is when every entity explicitly checks its input arguments when invoked. This is a realistic and effective middle way.

We require the language to make it practical to write most abstractions painlessly to this standard. We require the libraries to be mostly populated by abstractions that live up to this standard, and that the remaining members of the library explicitly state that they fall short of this standard.

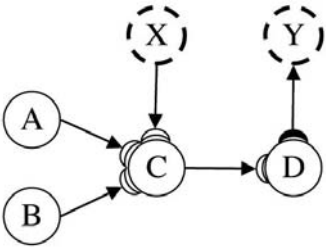

Fig. 3. Paths of vulnerability

Figure 3 shows an access graph. Dashed nodes are entities not relied upon in any way. White crescents indicate explicit checking of input arguments when invoked. A black crescent indicates explicitly checking all arguments when invoking. `A` and `B` are vulnerable to (rely upon) `C` and `C` is vulnerable to `D`, and since vulnerability is a transitive relation, `A` and `B` are also vulnerable to `D`. Because `C` checks its incoming arguments when invoked, it will protect itself and its clients from malicious arguments (e.g. provided by `X`). Paths of vulnerability are easy to follow and go one way only. Two clients vulnerable to the same server are not for that reason vulnerable to each other.

To support defensive correctness, Oz-E has to make it easy for the programmer to check incoming arguments. Guards, authentication primitives, and auditors, presented in the next sections, realize such support.

Guards. *E*'s *guards* [Sti00] form a soft typing system [CF91] that provides syntax support to make dynamic checking as easy as using static types. Guards are first class citizens and support arbitrary complex dynamic checking without cluttering the code with the actual tests. They can be user defined, and combined into more complex guards by logical operators.

Authentication. For an entity to defend its invariants in a mutually distrusting context, it can be important to know the origin of collaborating entities. The entity might want to authenticate a procedure before invoking it, and an argument before applying the procedure to it. Because capabilities unify designation and permission, and because the confused deputy problem can be naturally avoided, there is no need to authenticate the invoker.

We do not necessary want to know who wrote the code for that entity – since that knowledge is not very useful in general – but whether we want to rely upon the entity that loaded it and endowed it with initial authority. For example, if we rely upon bank B, we can authenticate an account-entity A by asking B if A is genuine, in other words if B recognizes A as one of the accounts B – or maybe an associated branch – created earlier.

Authentication by Invited Auditors. The above form of authentication is only useful to authenticate entities of which the alleged creator is a relied-upon third party. Moreover, this form of authentication cannot tell us anything further about the actual state of an entity at the time of authentication.

To reliably interact with entities of unknown origin, it must be possible to have them *inspected* by a relied-upon third party. Without breaking encapsulation – which would violate the principles in section 2.1 – that can be done as shown by E's *auditors* [YM00]. When an entity is created, a relied-upon third party *auditor* is *invited* by the creator, to inspect the entity's behavior and lexical scope. Later, when the auditor is asked to vouch for the relied-upon properties, it will reveal its conclusions, or if necessary re-inspect the state of the entity before answering yes or no. If inconclusive or uninvited, it will answer no.

Failing Safely. When an entity cannot guarantee its invariants in a certain condition, it should raise an exception. The default mechanism should not enclose any capabilities or potentially sensitive information with the exception that is raised. Part of this concern can be automated by the guards discussed earlier, who will throw an exception on behalf of the entity.

Preemptive Concurrency and Shared State. Preemptive concurrency enables an activation of an entity at some point in its progress to destroy the assumptions of another activation of the same entity at another point in its progress. This phenomenon is called *plan interference*.

Semaphores and locks give programmers control over the interaction between concurrently invoked behavior, but their use is error-prone and increases the overall complexity of a program. Good locking becomes a balancing exercise between the danger of race conditions and deadlocks. Preemptive concurrency with shared state makes defensive programming too hard because considering a single invocation of behavior is not enough to ensure preconditions and invariants.

For example, consider a simple "observer"-pattern [GHJV94]. With message-passing concurrency as explained in chapter 5 of [VH04] – all entities involved are *Active Objects*, subscription is done by providing a *Port*, and notification via a *Port.send* operation – all update notifications of an entity are guaranteed to arrive at all subscribers in the order of the update. With *threads* there is no guarantee whatsoever about the order of arrival and it becomes dauntingly hard to impose a proper order while at the same time avoiding deadlocks.

2.3 Additional Principles: Support for the Review Process

When the language is ready to provide all the necessary support for secure programming, one more important design concern remains. The programmers are now in the position to avoid security flaws while programming, but they also need to be able to quickly find any remaining vulnerabilities that might have got in. Oz-E must be designed to make security debugging easy. Its syntax should therefore allow programmers to quickly identify big parts in a program that *are* obviously safe, and concentrate on the remaining part.

A minimum set of tools to support debugging and analyzing the vulnerabilities is indispensable. These can range from support for syntax coloring to debuggers of distributed code and tools for security analysis. To this goal, we are currently researching formal models that allow us to analyze authority confinement amongst entities collaborating under mutual distrust [SMRS04]. A tool

based on this model would allow us to investigate the limits of the usability of patterns of safe collaboration that emerged from experience (e.g. the Powerbox [SM02] and the Caretaker[MS03]), and enable the discovery of new such patterns.

3 Proposed Structure of Oz-E

The Oz language has a three-layered design. We briefly introduce these layers here, and refer to chapter 2 and appendix D of [VH04] for a detailed explanation.

The lowest layer is a simple language, kernel Oz, that contains all the concepts of Oz in explicit form. The next layer, full Oz, adds linguistic abstractions to make the language practical for programmers.[1] The final layer, Mozart/Oz, adds libraries and their interfaces to the external environment that depends on the operating system functionality.

We realize that in an ideal world, the language and the operating system should be developed together. Pragmatically, we will provide as much of the operating system functionality as possible inside the third layer of the language. Any remaining functionality – not fitting the language without a complete rewrite of the operating system – will be accessible through a general system interface.

The importance of the layered architecture for security is stressed by a flaw in the current Mozart system that was found by Mark Miller. The module *Time*, currently available in the second layer as ambient authority, provides access to the system clock and should therefore be transferred to layer three, the functionality of which can only be available via explicitly granted capabilities.

Read access to the system time can be used to read covert channels regardless of the countermeasures (e.g. randomness in thread execution sequence and adding randomizing delays) the system could have taken to prevent this. Adversaries that are prevented from reading the system type might still be able to send out the secrets they can discover, but there are countermeasures that can make it arbitrary hard for them receive their instructions via covert channels.

We propose for Oz-E to keep as much of this layered structure as possible, while staying within the boundaries of the security requirements. We will start with very simple versions of these layers and grow them carefully into a full-featured language, maintaining the security properties throughout the process. The project will start by showing formally that the initial versions of kernel language and full language are secure. During the growth process, we will maintain at all times a formal semantics of the kernel language.

In the following three subsections, we present each of the three layers and we discuss some of the issues that need to be resolved for each layer. Of course, the early stage of the project does not allow us to attempt completeness in this respect.

[1] A *linguistic abstraction* is an abstraction with syntactic support. An *abstraction* is a way of organizing a data structure or a control flow such that the user is given a higher-level view and does not have to be concerned with its implementation.

3.1 Kernel Language

The kernel language should be complete enough so that there is no need to go lower, e.g., to a byte code level. As the kernel language is the lowest level seen by (normal) application developers and library designers, reasoning and program development will be simplified. Only the language designers themselves will go below that level. The implementation will guarantee that the kernel language satisfies its semantics despite malicious interference by programs written in it.

The initial kernel language will be as close as possible to the general kernel language of Oz, which has a complete and simple formal semantics as given in chapter 13 of [VH04]. This is the most complete formal semantics of Oz that exists currently. As far as we know, the relevant part of the Mozart system implements this semantics. It is straightforward to show that this kernel language satisfies basic security properties such as secure closures (encapsulation based on lexical scoping), absence of ambient authority, and unforgeable identity of kernel language entities.

In the rest of this subsection, we address two specific issues that are directly related to the kernel language, namely authentication and finalization. Authentication is an issue that is directly related to security. Finalization is an issue that is indirectly related to security: the current design has problems that would make building secure systems difficult.

We prefer the kernel language of Oz-E to be a subset of the full language. This results in semantic clarity, uniformity of syntax and simplicity, all important pedagogical assets when teaching Oz-E. Furthermore, the kernel language subset will allow us to experiment with language extensions while staying within the language.

Authentication via Token Equality. A basic requirement for building secure systems is authentication of authority-carrying entities. Entities that were created by relied-upon third parties should be recognizable with the help of the third party. This means that the entity needs an identity that is unforgeable and unspoofable, otherwise a creator could never be sure the entity is really the one it created earlier. Unforgeable means that it is impossible to create an identity out of thin air that matches with the identity of an existing entity. Unspoofable means that the authenticity check cannot be relayed (man in the middle attack).

The kernel language has to let us achieve these properties for its own authority-carrying entities and also for user-defined entities built using the kernel language. Both of these categories impose conditions on the kernel language semantics. Let us examine these conditions. In the following paragraphs we use the term “entity” to mean a language entity of a type that can carry authority (be a *capability*), as opposed to pure *data* (Sect. 7).

For kernel entities, authentication is achieved by the kernel language syntax and semantics. The kernel semantics ensures that each newly created entity has a new identity that does not exist elsewhere and that is unforgeable.

For user-defined entities, authentication has to be programmed. For example, say we have a user-defined entity called “object” that is implemented as a one-argument procedure. The object’s identity should not be confused with the

procedure's identity. This implies that the kernel language should have operations to build unforgeable and unspoofable identity into user-defined entities. One way to do this uses the concepts of chunk and name from the Oz kernel language. A *chunk* is a record with only one operation, field selection. A *name* is an unforgeable constant with an equality operation. With chunks and names, it is possible to build an operation that wraps an entity in a secure way, so that only the corresponding unwrap operation can extract the entity from the wrapped one [VH04]. This is similar to the sealer/unsealer pairs [Mor73] in the *E* language [Sti00].

Finalization. Finalization is the user-defined "clean-up" operation that is related to automatic memory management. When an entity is no longer reachable from an active part of the program, its memory can be reclaimed. Sometimes more than that has to be done to maintain the program invariants. For example, there might be a data structure whose value depends on the entity's existence (it counts the number of entities satisfying a particular property). Or the entity might hold a descriptor to an open file. Finalization handles cases such as these.

The current finalization in Oz does not guarantee that an entity that became unreachable is no longer used. The last operation performed on an entity before it becomes unreachable should truly be the last operation performed on the entity. To guarantee this, we propose to follow the "postmortem finalization" technique (executor of an estate). This was invented by Frank Jackson, Allan Schiffman, L. Peter Deutsch, and Dave Ungar.[2] When an entity becomes unreachable, the finalization algorithm invokes *another* entity, which plays the role of the executor of the first entity's estate. The executor will perform all the clean-up actions but has no reference to the original entity.

3.2 Full Language

The full language consists of linguistic abstractions built on top of the kernel language and (base) libraries written in the full language itself. Giving this linguistic support simply means that there is language syntax that is designed to support the abstraction. For example, a `for` loop can be given a concise syntax and implemented in terms of a `while` loop. We say that the `for` loop is a linguistic abstraction.

The full language has to be designed to support the writing of secure programs. This implies both building new abstractions for secure programming and verifying that the current language satisfies the properties of secure programming. The language should not provide ambient authority or leak potentially confidential information by default. For example, the current Mozart system has an exception handling mechanism that in some cases leaks too much information through the exceptions.

Modules and Functors. Like Oz, the full language will provide operations to create and manipulate software components. In Oz, these components are

[2] We searched for a publication to reference this work but found none.

values in the language called *functors*, which are defined through a linguistic abstraction. Functors are instantiated to become *modules*, which are executing entities. Modules are linked with other modules through a tool called the *module manager*. This linking operation gives authority to the instantiated module.

In Oz-E, the module manager has to be a tool for secure programming. For example, it should be easy to run an untrusted software component in an environment with limited authority, by linking it only to limited versions of running modules. Such modules can be constructed on the fly by the user's trusted shell or desktop program, to provide the right capabilities to host programs. This mechanism can also be used for coarse grained "sandboxing", e.g. to run a normal shell with a limited set of resources.

3.3 Environment Interaction

The security of Oz-E must be effective even though the environment is largely outside of the control of the Oz-E application developers and system developers. How can this be achieved? In the long term, we can hope that the environment will become more and more secure, similar to Oz-E itself. In the short term, we need libraries to provide controlled access to the operating system and to other applications.

Security of an application ultimately derives from the user of the application. An application is secure if it follows the user's wishes. The user should have the ability to express these wishes in a usable way through a graphical user interface. Recent work shows that this can be done [Yee02]. For example, selecting a file from a browser window gives a capability to the application: it both designates the file and gives authority to perform an operation (such as an edit) on the file. A prototype desktop environment, CapDesk, has been implemented using these ideas. CapDesk shows that both security and usability can be achieved on the desktop [SM02].

Oz has a high-level GUI tool called QTk. It combines the conciseness and manipulability of the declarative approach with the expressiveness of the procedural approach. QTk builds on the insecure module Tk and augments that functionality instead of restricting it. QTk has to be modified so that it satisfies the principles enunciated in [Yee02] and implemented in CapDesk.

4 Cross-Layer Concerns

The previous section presented a layered structure for the Oz-E language and system. In general however, security concerns cannot be limited to a single layer in such a structure. As explained by another paper in this book [MTS05], they are pervasive concerns. Some them will affect several layers. In this section we discuss three such concerns: pragmatic issues of how to make the system easy to program, execution on distributed systems, and the need for reflection and introspection.

4.1 Pragmatic Issues in Language Design

A secure language should not just make it *possible* to write secure programs, it must also make it *easy* and *natural*. Otherwise, one part of a program written with bad discipline will endanger the security of the whole program. The default way should always be the secure way. This is the security equivalent of fail-safe programming in fault-tolerant systems.

We propose to use this principle in the design of the Oz-E concurrency model. The two main concurrency models are message-passing concurrency (asynchronous messages sent to concurrent entities) and shared-state concurrency (concurrent entities sharing state through monitors). Experience shows that the default concurrency model should be message-passing concurrency. This is not a new idea; Carl Hewitt anticipated it long ago in the Actor model [Hew77, HBS73]. But now we have strong reasons for accepting it. For example, the Erlang language is used for building highly available systems [Arm03, AWWV96]. The *E* language is used for building secure distributed systems [MSC+01]. For fundamental reasons, both Erlang and *E* use message-passing concurrency. We therefore propose for Oz-E to have this default as well. One way to realize this is by the following semantic condition on the kernel language: *cells can only be used in one thread*. This simple semantic condition has as consequence that threads can communicate only through dataflow variables (declarative concurrency) and ports (message-passing concurrency).

4.2 Distributed Systems

The distribution model of Oz allows all language entities to be partitioned over a distributed system, while keeping the same semantics as if the entities were on different threads in a single system, at least when network or node failures are not taken into account. For every category of language entities (stateless, single-assignment, and stateful) a choice of distributed protocols is available that minimizes network communications and handles partial failure gracefully. Fault-tolerant abstractions can be built within the language, on top of this system.

We want to keep the Oz-E distribution system as close as possible to this model and put the same restrictions on communication with remote threads as with local threads (such restrictions were discussed in section 4.1).

We are in the process of replacing the current, monolithic implementation of distribution in Mozart by a modular implementation using the DSS (Distribution Subsystem) [KEB03]. The DSS is a language-independent library, developed primarily by Erik Klintskog, that provides a set of protocols for implementing network-transparent and network-aware distribution. We will briefly consider the opportunities offered by the DSS to add secure distribution to Oz-E.

Responsibility of the Language Runtime System. The division of labor between the DSS and the language system assigns the following responsibilities to the language runtime system:

1. Marshalling and unmarshalling of the language entities.
2. Differentiating between distributed and local entities.
3. Mapping of Oz-E entities and operations to their abstract DSS-specific types, which the DSS will distribute.
4. Choosing amongst the consistency protocols provided by the DSS, based on the abstract entity types, and adjustable for individual entities.

Secure marshalling should not break encapsulation, and every language entity should be allowed to specify and control its own distribution strategy and marshalling algorithm. *E* provides such marshalling support via "Miranda" methods that every object understands and that provide a safe default marshalling behavior which can be overridden. Oz-E could build a similar implementation for the language entities that can perform method dispatching (e.g. objects). For the other entities (e.g. zero-argument procedures), Oz-E could allow specialized marshalers to be invited into the lexical scope of an entity when it is created. Section 5.2 gives two examples of how invitation can be implemented in Oz-E. Alternatively, Oz-E's kernel language could use only object-style procedures that by default forward marshalling behavior to marshalers, and that can override this behavior.

Depending on these choices, marshalling might need support at the kernel language level. The other three responsibilities of the language system can be provided as part of an Oz-E system library.

Responsibility of the Distribution Subsystem. The DSS itself takes responsibility for:

1. Distributing abstract entities and abstract operations.
2. Providing consistency, using the consistency protocols that were chosen.
3. Properly encrypting all communication, making sure that external parties cannot get inside the connection.
4. Ensuring that it is unfeasibly hard to get (guess) access to an entity without having received a proper reference in the legal way.
5. Authenticating the distributed entities to ensure that no entity is able to pretend to be some other entity.

In [BKB04] the DSS is shown to have security requirements that are compatible with the requirements for safely distributing capabilities. Three attack scenarios have been investigated:

1. Outsider attacks. It should be impossible (infeasibly hard) for an attacker node that does not have legal access to any distributed entities, to access an entity at a remote site or to make such an entity unavailable for legal access.
2. Indirect attacks. It should be impossible for an attacker node that has legal access to a distributed entity but not the one being attacked, to perform this kind of intrusion or damage.
3. Insider attacks. It should be impossible for an attacker node that has legal access to a distributed entity, to render the entity unavailable for legal access.

This can only be guaranteed for protocols that do not distribute or relocate state such as protocols for asynchronous message sending or stationary objects (RPC), and only if the attacker node did not host the original entity, but only a remote reference to it.

Apart from the requirements of the second scenario, the current DSS implementation claims to follow all these requirements. DSS distribution protocols will be made robust to ensure that no DSS-node can be crashed – or forced to render entities unavailable for legal access – by using knowledge of the implementation. This is called "protocol robustification" and is still under development.

The fact that only asynchronous message sending and RPC-style protocols are protected from insider attacks is no objection for Oz-E. In section 4.1 such restriction was already put on the interaction between entities in different threads: normal threads on as single node will not be able to share cells.

4.3 Reflection and Introspection

To verify security properties at runtime, we propose to add the necessary primitive operations to the kernel language, so that it can be programmed in Oz-E itself. How much should a program be able to inspect itself, to verify security properties? The problem is that there is a tension between introspection and security. For example, a program might want to verify inside a lexically scoped closure. Done naively, this breaks the encapsulation that the closure provides. In general, introspection can break the encapsulation provided by lexical scoping.

To avoid breaking encapsulation the *E* language allows a user-defined entity to invite relied-upon third parties (auditors) to inspect an abstract syntax tree representation of itself, and report on properties that they find. Section 5.2 shows how this could work in Oz-E.

Safe Debugging. In a distributed environment, where collaborating entities spread over different sites have different interests, how can debugging be done? The principle is similar to safe introspection: entities are in control of what debugging information they provide, and the debugger is a third party that may or may not be "invited into the internals" of the entity.

Code Verification. Loaded code should not be able to bring about behavior which exceeds behavior that could be described within the kernel language. Since we plan to use the Oz VM to run Oz-E bytecode, and the Oz VM itself provides no such guarantee, we must verify all code before loading it. Such verification of byte code is a cumbersome and error-prone task. Oz-E should be restricted to load code from easily verifiable abstract syntax tree (AST) representations of kernel and full language statements instead of byte code.

5 Some Practical Scenarios

In this section we take a closer look at how some of these ideas could be implemented. We want to stress that the examples only present one of the many

possible design alternatives and do not express any preferences or recommendations from the authors. They are only provided as a clarification to the principles and as a sample of the problems that Oz-E designers will need to solve.

5.1 At What Level Should We Implement Guards?

In section 2.2 we explained briefly the benefits of guards and how they are supported in E. Let us now show in pseudocode how expressions could be guarded in Oz-E and how a linguistic abstraction for guards could look like.

```
fun {EnumGuard L}
   if {Not {List.is L}}
   then raise notAList(enumGuard) end
   end
   for X in L do {Wait X} end
   proc {$ X}
      try
         if {Member X L}
         then skip
         else raise guardFailed(enumGuard) end
         end
      catch _ then
         raise guardFailed(enumGuard) end
      end
   end
end
Trilogic = {EnumGuard [true false undefined]}
{Trilogic (x == y)} % will succeed
{Trilogic 23}       % will raise an exception
```

Fig. 4. A three valued logic type guard

The example in Figure 4 guards a three valued logic type consisting of **true**, **false**, or unknown. `EnumGuard` ensures that the set is provided as a list and that all its elements are bound. Then it creates a single parameter procedure that will do nothing if its argument is in the set, or raise an exception otherwise. A guard `Trilogic` is created from that, and tested in the two last lines. The first test will succeed, the second one will raise an exception.

What if we want to use this guard in a procedure declaration? Let's first assume we want to guard an input parameter, in this case `X`. Then:

```
proc {$ X:Trilogic ?Y} <S> end
```

can be translated into:

```
proc {$ X ?Y} {Trilogic X} <S> end
```

Guarding output parameters is more difficult. If `Y` is unbound then:

```
proc {P X ?Y:Trilogic} <S> end
```

```
proc {$ X ?Y}
   Y2
in
   thread
      try {Trilogic Y2} Y = Y2
      catch Ex
      then Y = {Value.failed Ex}
      end
   end
   <S>{Y->Y2}  %(1)
end
```

Fig. 5. Guarding output parameters

can be translated as shown in Figure 5. Note that in Figure 5 the expression marked `(1)` represents the statement `<S>` in which all free occurrences of the identifier `Y` are replaced by an identifier `Y2` which does not occur in `<S>` (see chapter 13 of [VH04]).

These examples work for atomic values that are either input or output parameters, but they cannot simply be extended for guarding partial values, because the latter can be used for both input and output at the same time. Another problem is the relational programming style where all parameters can be input, output or both depending on how the procedure is used. This definitely calls for more research, possibly revealing the need for a new primitive to support guards.

5.2 A Mechanism for Invitation and Safe Introspection

Let's assume we have a new construct `NewProc` that takes an abstract syntax tree (AST) and an environment record mapping the free identifiers in the AST to variables and values, and returns a procedure. Instead of creating a procedure like this:

```
P1 = proc {$} skip end
```

we could now also create a procedure like this:

```
P1 = {NewProc ast(stmt:´skip´) env()}
```

To create an *audited* procedure, an auditor is invoked with an AST and an environment. The client of the procedure can call the auditor to inquire about the properties that it audits. Let's build an auditor to check declarative behavior. We first present one that keeps track of the declarative procedures it creates.

Figure 6 builds an auditor procedure that takes a message as argument. If the message matches `createProc(...)` it will investigate the AST and environment provided, and create a procedure by calling `{NewProc ...}` with the same arguments. If the investigation returned **`true`**, it will store the resulting procedure in a list of all the created procedures that succeeded the `Investigate` test. If the message matches `approved(...)` it will check this list.

Rees [Ree96] gives strong arguments against the approach of Figure 6, as it easily leads to problems with memory management, performance, and to seman-

```
declare
local
   AuditedProcedures =  {NewCell nil}
   fun {Investigate AST Env}
      ... % return boolean indicating whether
          % {NewProc AST Env} returns a declarative procedure
   end
   proc {MarkOK P}   % remember that P is declarative
      AuditedProcedures := P | @AuditedProcedures
   end
   fun {IsOK P}      % checks if P is marked declarative
     {Member P @AuditedProcedures}
   end
in
   proc {DeclarativeAuditor Msg}
         case Msg
         of createProc(Ast Env ?P) then
            if {Investigate Ast Env}
            then
               NewP = {NewProc Ast Env}
            in
               {MarkOK NewP}
               P = NewP
            else P = {NewProc Ast Env}
            end
         [] approved(P ?B) then
            B = {IsOK P}
         end
      end
   end
end

P1 = proc {$} skip end
P2 = {DeclarativeAuditor createProc(ast(stmt:´skip´) env())}
P1OK = {DeclarativeAuditor approved(P1 $)} % P1OK will be false
P2OK = {DeclarativeAuditor approved(P2 $)} % P2OK will be true
```

Fig. 6. Stateful auditor that investigates declarativity

tic obscurity. For this reason W7 – like *E* – has chosen to provide a primitive function to create sealer-unsealer pairs. Figure 7 provides an alternative approach that avoids these drawbacks.

The auditor built in Figure 7 is stateless, and lets `MarkOK` wrap the created procedure in some kind of recognizable entity that can be invoked as a normal procedure. An invokable *chunk* would do for that purpose, as it could have a secret field accessible by the *name* `Secret` known only to the auditor. For this to work, Oz-E's kernel language has to provide either invokable chunks or a primitive function to create sealer-unsealer functions.

```
declare
local
   Secret = {NewName}
   fun {Investigate AST Env}
      ... % return boolean indicating whether
          % {NewProc AST Env} returns a declarative procedure
   end
   fun {MarkOK P}
      WrappedP in
      ... % wrap P in some sort of invokable chunk WrappedP
      ... % WrappedP when invoked, will transparently invoke P
      WrappedP.Secret = ok
      WrappedP
   end
   fun {IsOK P}  % checks if P is marked declarative
      try P.Secret == ok catch _ then false end
   end
in
   proc {DeclarativeAuditor Msg}
         case Msg
         of createProc(Ast Env ?P) then
            if {Investigate Ast Env}
            then P = {MarkOK {NewProc Ast Env $}}
            else P = {NewProc Ast Env}
            end
         [] approved(P ?B) then
            B = {IsOK P}
         end
      end
   end
end
```

Fig. 7. Stateless auditor that investigates declarativity

Instead of providing the environment directly for the auditor to investigate, [Rei04] suggests a mechanism to manipulate the values in the environment before giving them to the auditor (e.g. by sealing) to make sure that they cannot be used for anything else than auditing.

Instead of inviting an auditor, one could invite a relied-upon third party that offers general introspection and reflection. It would have roughly the same code-frame as the auditor, but provide more detailed – and generally non-monotonic – information about the internal state and the code of the procedure.

6 Conclusions and Future Work

A long-term solution to the problems of computer security depends critically on the programming language. If the language is poorly designed, then assuring security becomes complicated. If the language is well-designed, for example, by

thoroughly following the principle of least authority, then assuring security is much simplified. With such a language, problems that appear to be very difficult such as protection against computer viruses and the trade-off between security and usability become solvable [Sti].

A major goal of Oz language research is to design a language that is as expressive as possible, by combining programming concepts in a well-factored way. The current version of Oz covers many concepts, but it is not designed to be secure. This paper has given a rough outline of the work that has to be done to create Oz-E, a secure version of Oz that supports the principle of least authority and that makes it possible and practical to write secure programs. We have covered both language and implementation issues. We also explain what problems arise when a secure language lives in an insecure environment. Building Oz-E will be a major undertaking that will require the collaboration of many people. But the potential rewards are very great. We hope that this paper will be a starting point for people who want to participate in this vision.

7 Glossary

Data. A reference to an Oz-entity that has structural equality and consists only of atoms, numbers, and completely grounded records that contain only data.

Capability. An unforgeable reference that designates an entity of any type with token identity. A capability comes with a fixed set of permissions: the different kinds of interactions it supports.

Permission. A means for interacting with the entity designated by a capability. For example, a procedure comes with the permission to be applied to values.

Authority. Any directly or indirectly observable effect an entity can cause. The entity has to use a permission to achieve such an effect. Invoking a procedure for instance could result in the update of a file, or influence the state of an object that will eventually effect the screen.

Dynamic Authority and Revocation. While the permission to invoke a procedure cannot be revoked, the authority that is provided by such a permission can dynamically change and even reduce to zero. Authority depends on the behavior of the invoked entity, which is usually influenced by its state and by the arguments provided to it. Authority also depends on the behavior of the *invoker*, which can decide whether or not it will use the returned values, and to which extent. Authority is thus generated via collaboration during the exertion of a permission, and both collaborators – invoker and invoked entity – have certain means to dynamically influence the authority that is realized.

Endowment. When creating an entity, the creating entity can provide part of its authority to the created entity.

Parenthood. When creating an entity, the creating entity automatically gets the only initial capability to the created entity.

Secure Programming. Programming using components of which the reliability is unknown or uncertain, while still guaranteeing that a predefined level of vulnerability is not exceeded. Secure programming has to guarantee two conditions:

1. all relied-upon components are programmed reliably so that they
   (a) do not abuse their authority to inflict unacceptable damage, and
   (b) cannot be lured into doing so by their collaborators.
2. no authority that can be abused to inflict unacceptable damage can become available to not-relied-upon components.

Acknowledgments

This work was partially funded by the EVERGROW project in the sixth Framework Programme of the European Union under contract number 001935, and partly by the MILOS project of the Walloon Region of Belgium under convention 114856. We owe a great deal of our insights and ideas on Oz-E to the e-lang community. We thank Raphaël Collet, Boriss Mejias, Yves Jaradin and Kevin Glynn for their cooperation during the preparation of this paper. We especially want to thank Mark Miller for contributing ideas on capability-secure programming and defensive correctness, for pointing out some security flaws in the current Mozart implementation, and for reviewing this paper and suggesting corrections and reformulations. Any remaining errors and obscurities in the explanation are the sole responsibility of the authors.

References

[Arm03] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors.* PhD thesis, Royal Institute of Technology (KTH), Stockholm, December 2003.

[AWWV96] Joe Armstrong, Mike Williams, Claes Wikström, and Robert Virding. *Concurrent Programming in Erlang.* Prentice-Hall, Englewood Cliffs, NJ, 1996.

[BKB04] Zacharias El Banna, Erik Klintskog, and Per Brand. Report on security services in distribution subsystem. Technical Report PEPITO Project Deliverable D4.4 (EU contract IST-2001-33234), K.T.H., Stockholm, January 2004.

[CF91] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Massachusetts, 1994.

[Har89] Norm Hardy. The confused deputy. *ACM SIGOPS Oper. Syst. Rev*, 22(4):36–38, 1989.
`http://www.cap-lore.com/CapTheory/ConfusedDeputy.html`.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245, August 1973.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, June 1977.

[KEB03] Erik Klintskog, Zacharias El Banna, and Per Brand. A generic middleware for intra-language transparent distribution. Technical Report T2003:01, Swedish Institute of Computer Science, June 2003.

[MMF00] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Proceedings of the 4th International Conference on Financial Cryptography*, pages 349–378. Springer Verlag, 2000.

[Mor73] James H. Morris. Protection in programming languages. *Communications of the ACM*, 16(1):15–21, 1973.

[MS03] Mark S. Miller and Jonathan Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *8th Asian Computing Science Conference (ASIAN03)*, pages 224–242, December 2003.

[MSC+01] Mark Miller, Marc Stiegler, Tyler Close, Bill Frantz, Ka-Ping Yee, Chip Morningstar, Jonathan Shapiro, Norm Hardy, E. Dean Tribble, Doug Barnes, Dan Bornstien, Bryce Wilcox-O'Hearn, Terry Stanley, Kevin Reid, and Darius Bacon. E: Open source distributed capabilities, 2001. Available at `http://www.erights.org`.

[MTS05] Mark S. Miller, Bill Tulloh, and Jonathan S. Shapiro. The structure of authority: Why security is not a separable concern. In *Multiparadigm Programming in Mozart/Oz: Proceedings of MOZ 2004*, volume 3389 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2005.

[Ree96] Jonathan A. Rees. A security kernel based on the lambda-calculus. Technical report, MIT, 1996.

[Rei04] Kevin Reid. [e-lang] Proposal: Auditors without unshadowable names, August 2004. Mail posted at e-lang mailing list, available at `http://www.eros-os.org/pipermail/e-lang/2004-August/010029.html`.

[SM02] Marc Stiegler and Mark S. Miller. A capability based client: The darpabrowser. Technical Report Focused Research Topic 5 / BAA-00-06-SNK, Combex, Inc., June 2002. Avalalbe at `http://www.combex.com/papers/darpa-report/index.html`.

[SMRS04] Fred Spiessens, Mark Miller, Peter Van Roy, and Jonathan Shapiro. Authority Reduction in Protection Systems. Available at: `http://www.info.ucl.ac.be/people/fsp/ARS.pdf`, 2004.

[Sti] Marc Stiegler. The SkyNet virus: Why it is unstoppable; how to stop it. Talk available at `http://www.erights.org/talks/skynet/`.

[Sti00] Marc Stiegler. *The E Language in a Walnut*. 2000. Draft available at `http://www.erights.org`.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, 2004.

[Yee02] Ka-Ping Yee. User interaction design for secure systems. In *4th International Conference on Information and Communications Security (ICICS 2002)*, 2002. UC Berkeley Technical Report CSD-02-1184.

[YM00] Ka-Ping Yee and Mark S. Miller. Auditors: An extensible, dynamic code verification mechanism.
Available at `http://www.erights.org/elang/kernel/auditors/`, 2000.