Taming of Pict

Matej Košík[1]

Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
kosik@fiit.stuba.sk

Abstract. This article presents additional necessary measures that enable us to use Pict as an object-capability programing language. It is desirable to be able to assess the worst possible threat that we—users—risk if we run a given program. If we know the threat, we are able to decide whether or not we are willing to risk running the program. The cost of a security audit that reveals such an assessment will be non-zero but it need not to be directly dependent on the size of the whole original program. It is possible to write programs in such a way that this analysis can be reliably performed on a fraction of the original program—on the trusted computing base. This technique does not always give the most accurate assessment but it gives sound and interesting assessment relatively cheaply. It does not prevent usage of other techniques that can further refine the initial assessment.

1 Introduction

There are two different points of view of a computer system. We can view it from an administrator's point of view and from a user's point of view. Users should be regarded as primary because the purpose of computers is not to be administered but to be used. The goal of the administrator is to ensure that none of the users is given excess authority. The goal of the user is (should be) to ensure that each of the processes runs with appropriate authority. Security mechanisms provided by operating system are practical for administrator but they do not help users with their security goals. Microsoft "Immutable" Law #1 states:

> If a bad guy can persuade you to run his program on your computer, it's not your computer anymore.

The problem is, how to decide who is a good guy and who is a bad guy. More importantly, even good guys can make mistakes and their programs can cause damage. The purpose of the computer is that we—users—can run programs on it. This rule basically says that we are safe as long as we do not run any program on it. Let us stop here and think how ridiculous it is.

Noticeable progress has been made in the area of designing programming languages with respect to security. Outstanding example is the E programming language [1]. From the security point of view[1], it is interesting because it enables

[1] The E programming language addresses also other important problems.

programmers to follow the principle of the least authority (POLA). Multiple aspects of the language contribute to this fact:

- the authority to invoke methods of a particular object is an unforgeable capability
- when some subsystem decides to keep some capabilities as private, there are no language constructs that would enable other untrusted subsystems to "steal" them
- the reference graph can evolve only according to *rules of allowed reference graph dynamics* presented in Section 9.2 of [1]

The contribution of this paper is that it shows how, through a refactorization of the libraries of the Pict programming language [2], the "ambient authority" is reduced to a minimum, and Pict can provide many of the benefits of existing object-capability languages. Provided examples illustrate the technique for determining authority of untrusted subsystems without the need to analyze their code.

2 Related Work

While this article is mostly concerned with taming of Pict—turning Pict into an object-capability programming language—this is not the first work of this kind. See for example: Oz-E [3], Emily [4], Joe-E [5].

The Raw Metal occam Experiment (RMoX) [6] can be regarded as a source of inspiration that languages, based on process calculi, can be used for defining of behavior of various operating system's components and their mutual interaction. Using programming language constructs as a mechanism for isolation of various subsystems from each other instead of relying on awkward hardware support is also one of the points of the Singularity project [7].

3 The Pict Programming Language

The goal of the authors of the Pict programming language was to create a language that could play for the π-calculus a similar role as Haskell plays for the λ-calculus. It is defined in layers, see Figure 1. Syntax of the core language is formally described in the Pict Language Definition [8] in Chapter 3; see rules tagged as C (as *Core*). Some of the syntactically correct programs can be further rejected by the typing rules at compile time. Semantics of the core language is defined in Chapter 13 of that document. It defines:

- structural congruence relation
- reduction relation

These together define behavior of all Pict programs.

Programs written in the core Pict cannot break *rules of allowed reference graph dynamics*. Derived forms make functional and sequential programming in

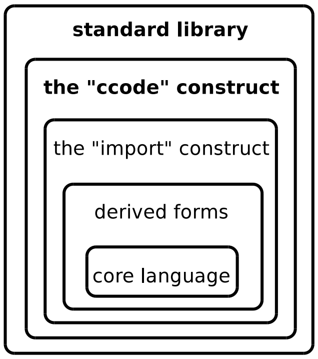

Fig. 1. Layers of the Pict programming language. Programs that are composed solely from core constructs, derived forms and `import` directives are completely harmless because they have minimal authority.

Pict more convenient. By definition, they do not add expressivity to the core Pict language and thus can be used without concerns that the *rules of allowed reference graph dynamics* could be broken.

The `import` directive is one of the two extralinguistic constructs of Pict. It enables us to split the whole program into multiple, separately compilable modules. These modules are related via `import` construct. This relation forms partial order with the biggest element—it is the main module of a complete program. There is no `export` directive via which the programmer could explicitly specify which bindings he wants to export from a given module. All the variables that are bound in the outer-most scope are automatically exported. The effect of the `import` directive is that all the variables exported by the imported module are visible in the importing module.

The `ccode` construct is the second of the two extralinguistic constructs of Pict. It enables the programmer to inline arbitrary C code into Pict programs. This is very useful and very dangerous at the same time. It is the sole mechanism that Pict programs can use to interact with their (non-Pict) environment such as the operating system. It is also used for implementation of certain operations in an efficient way. This is not absolutely essential[2] but it is pragmatic. Additional rules presented later in the text ensure that this construct cannot be directly or indirectly abused by untrusted modules to gain excess authority. These additional rules ensure that untrusted modules cannot break the *rules of allowed reference graph dynamics*.

The Standard Pict Library [9] provides several reusable components. Figure 2 shows some modules that are part of this library. Some aspects of this original organization are logical and some are not logical. The `import` directive binds them into a partially ordered set. Minimal elements (`Misc`, `Prim`) are shown on

[2] The core language can model integers, booleans, strings and other values of basic data types together with operations with them.

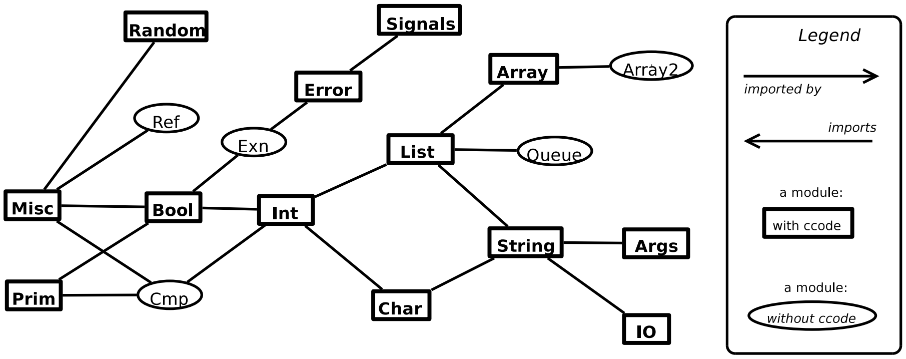

Fig. 2. Partial ordering of modules with respect to the `import` relation. These modules are described in detail elsewhere [9]. This figure is provided only for general impression concerning the structure of modules. Not all the aspects of this original version are completely logical.

the left. Maximal elements (`Random`, `Ref`, `Signals`, `Array2`, `Queue`, `Args`, `IO`) are shown on the right. The $A \prec B$ means that module A is imported by module B.

Those names that are bound in the outer-most scope of some module are also exported by that module. Let $en(A)$ denote a function that maps a given module A to the set of names that it exports. Then, by definition of the semantics of the `import` directive:

$$A \prec B \;\Rightarrow\; en(A) \subseteq en(B)$$

That is, all the names bound in the outer-most scope of module A are also bound in the outer-most scope of module B that imports A.

When $bn(A)$ denotes the set of all names bound in any scope within module A then for all modules A, by definition of the `import` construct, holds:

$$en(A) \subseteq bn(A)$$

All the names bound in A need not to be, and usually indeed are not, exported.

4 Refactorization of the Original Pict Library

The attempt to minimize the trusted computing base is inherently a good idea. In this light, the organization of the original Pict library is not optimal. Each module that employs the `ccode` construct must be considered as part of the trusted computing base. And, as you can see in Figure 2, there are many such modules. Additionally, the original set of primitives, expressed via `ccode` construct, is not orthogonal. Many of the existing primitives can be rewritten in terms of a pure Pict code. After we removed those superfluous primitives and we concentrated the originally scattered primitives in a few dedicated modules,

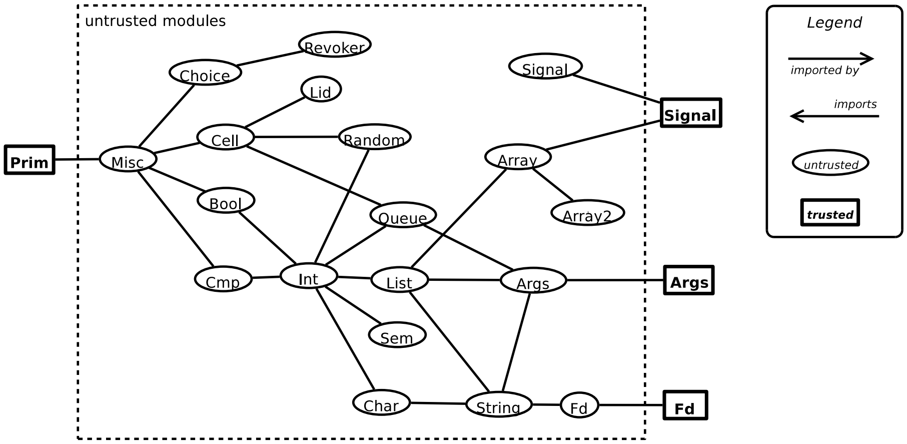

Fig. 3. Refactored standard library of Pict with respect to security.

the situation is different, see Figure 3. Now it has sense to discriminate among trusted and untrusted modules as follows:

- *trusted* modules can contain any (compilable) code
- *untrusted* modules:
  - cannot use the `ccode` construct
  - cannot import any trusted module except for the `Prim` module which provides harmless primitives

Due to the inherent properties of the Pict programming language, these measures are sufficient to ensure that rules of allowed reference graph dynamics hold for all our untrusted modules.

After all these measures, by *minimal authority* of untrusted modules written in Pict we mean:

- Trusted modules have no straightforward way to influence policy how much memory can various sub-components allocate from the common heap of free memory which in Pict is bounded—it's size is by default 1 MB. When untrusted components exhaust it, the Pict runtime prints out a relevant error message and terminates the whole system.
- Trusted modules have no straightforward way how influence the scheduling policy that would define rules for CPU utilization by untrusted modules. At present, there is a scheduler. It ensures fairness of the CPU utilization but this is not what we always want.
- Untrusted modules can make run-time errors (such as division by zero or they might attempt to access non-existent element of some array). These run-time errors are always detected and result in calling the `error` function that performs appropriate actions. At present, this means that some error message will be printed on the screen and the whole system will terminate.

So indirectly, untrusted modules have the authority to terminate the whole system.

This situation is not at all fully satisfactory. But this is still far better than ambient authority of trusted modules. To address these remaining issues the whole Pict runtime must be redesigned with these problems in mind.

5 Powerbox

Powerbox is a fundamental security pattern. Its origin can be traced to the DarpaBrowser described in [10] and in [11]. It enables us to dynamically raise the level of authority of untrusted subsystems to a sufficient and acceptable level.

If we are concerned with some single purpose program then we have to identify the authority this program needs. There is nothing inherently wrong that various programs require some authority. As long as it is explicitly declared, users or security auditors can efficiently judge whether it is acceptable for us to grant such authority to the actual program.

To be able to follow POLA, the whole program must be split in at least two modules. The first of them will be trusted and the other one will be untrusted. The purpose of the trusted module is to communicate part of its ambient authority to the untrusted module. The purpose of the untrusted module is to use the authority it is given and to do what we expect from it. It receives required capabilities "by introduction". What kind of capabilities are communicated and through which channel depends on the contract between the trusted and the untrusted part.

Let us show a very simple example. We keep the untrusted module in the `Untrusted/Guest.pi` file and the trusted module in the `Trusted/Host.pi` file. The `Untrusted/Guest` module might look as follows:

```
new contract : ^!String

run contract?logger = ( logger!"0123456789"
                      | logger!"0123456789"
                      | logger!"0123456789"
                      | logger!"0123456789"
                      | logger!"0123456789"
                      )
```

It creates a fresh channel `contract` that can be used for passing values of the `!String` type[3]. The process in the untrusted module blocks until it receives a

[3] Pict is a strongly typed programming language. Each channel has a type. This type determines what kind of values can be communicated over a given channel. An attempt to send a wrong type of value over some channel is detected at compile time. The `contract` capability has a type `^!String`. Our process holds this capability *by initial conditions*. The initial `^` character means that this capability can be used for sending as well as for receiving values of the `!String` type. Our process uses this

value from the `contract` channel. When such value arrives, it will be bound to the `logger` variable. The untrusted guest then has all the authority it needs to do its job. In this case it prints 50 characters on the screen. The above program is not very useful but it could very well perform various simulations and then print out the simulation report. The above code fragment is a mere illustration.

The `Trusted/Host` module is responsible for selecting parts of its ambient authority and communicating appropriate capabilities to the untrusted module. For example:

```
import "Untrusted/Guest"

run contract!print
```

It imports two modules. The first one is `Untrusted/Guest`. This means that it will see the `contract` capability exported by that module (because it is bound in the outer-most lexical scope). It also imports the `Trusted/Fd` module. It means that it will see the `print` capability exported by that module. It is up to this trusted module to select proper capabilities. In this case it selects the `print` capability and sends it over the `contract` channel. Of course, there may be situations where some guest needs more than one capability. In those cases the trusted host sends an n-tuple of capabilities.

Appropriate makefile for building executable out of these two modules can look as follows:

```
Host: Trusted/Host.pi Untrusted/Guest.px
        pict -o $@ $<

Untrusted/Guest.px: Untrusted/Guest.pi
        isUntrusted $< && pict -reset lib -set sep -o $@ $<

clean:
        @rm -f {Trusted,Untrusted}/*.{o,px} Host
```

Please notice two things:

- `Untrusted/Guest.pi` module is checked with the `isUntrusted` script whether it indeed can be regarded as untrusted[4]
- we compile the `Untrusted/Guest.pi` module with the `-reset lib` flag that inhibits inclusion of the standard prelude[5].

capability to receive a value from it and binds this value to the `logger` capability. This capability is of `!String` type. That means that the `logger` capability can be used for sending strings along it. It cannot be used for receiving strings from this channel.

[4] Untrusted modules cannot employ the `ccode` constructs. Untrusted modules cannot import trusted modules except for the `Trusted/Prim` module.

[5] Precise information concerning the “standard prelude” can be found in [12]. Basically, it is a default sequence of `import` directives that is desirable in case of trusted modules but it is undesirable in case of untrusted modules.

These two actions give us enough confidence to believe that the `Untrusted/Guest` module has initially *minimal authority.* Its authority is later raised to be able to print characters on the standard output. It is not given any other authority. It cannot tamper with files that can be accessed by the user that runs this program. The untrusted module cannot communicate with other processes on your local system. Neither it can communicate over network. It can only print as many characters on the standard output as it wishes. For some programs this kind of authority might be completely sufficient and as you can see it can be trivially implemented.

The same scheme has many variants. The derived forms make certain useful things such as functional programming as well as sequential programming easier. If we express our trusted host and our untrusted guest in so called "continuation passing style" then it would appear that the trusted host gives the untrusted guest the capability to call certain functions. In this case, it is completely up to the trusted host to choose the right set of function-capabilities. The chosen set determines the authority of the untrusted guest.

The Powerbox pattern can also be used in situations when our system consists of multiple untrusted subsystems. In that case, each subsystem will be placed in a separate powerbox. This way, each untrusted component can be given different capabilities and thus we can determine the authority of particular untrusted modules independently. The complexity of the trusted part is determined by the complexity of our security policy. It is independent from the complexity of the untrusted part that does the real job.

6 Experiments in the Kernel Space

Capability-secure languages are useful not only in user-space but they can have interesting applications in the kernel space, too. They can be a precursor to making progress in monolithic (in the traditional sense) kernels. We have a flexible alternative to the classical microkernel-based operating system architecture. In our preliminary experiment we use the Pict programming language because it was easier to adapt to run on a bare metal. From the security point of view Pict is in principle as good as E. From the concurrency point of view, the E programming language is much better. It provides more advanced synchronization mechanisms than Pict so E is a very good alternative for the future. Figure 4 shows the structure of modules with respect to the `import` relationship. This relationship determines the *connectivity by initial conditions* according to the semantics of the `import` directive. Capabilities that are exported from module A are also visible in module B if module A is imported by module B. Modules `Memory`, `IRQ` and `IO` provide various powerful primitives. For example the `Trusted/Memory` module exports a function

```
(memory.write.byte offset value)
```

that enables (those who see this function-capability) to write any `value` of byte size to any `offset` within current data segment (that spans through the whole physical memory). The `Trusted/IO` module exports two functions:

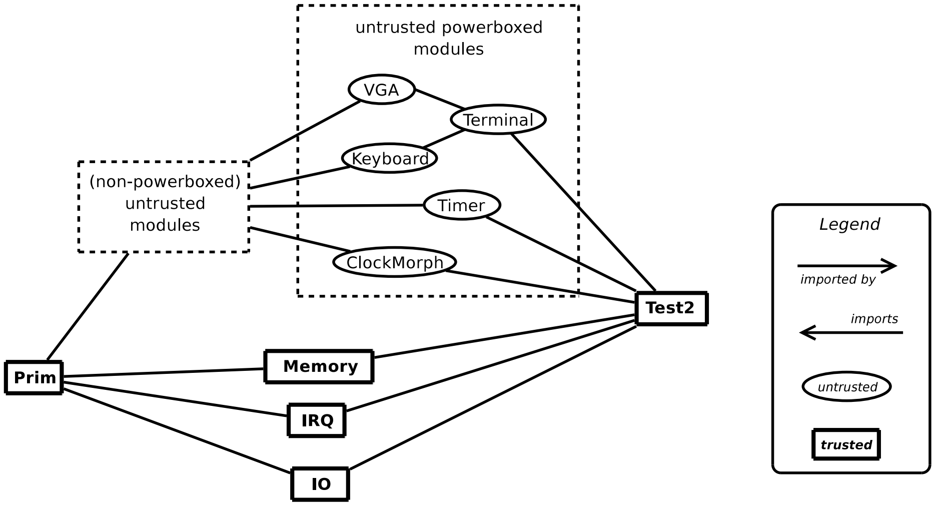

Fig. 4. Structure (with respect to the `import` relation) of modules that form our experimental kernels. `Test2` is the main module.

```
(io.write.byte port value)
(io.read.byte port)
```

Those who see the first function can write any `value` (byte) to any I/O `port`. Those who see the second function can read any I/O `port` of byte size.

The trusted `Test2` module has a single task, to disseminate proper capabilities to proper modules via the Powerbox pattern according to POLA. For example, one thing that the `VGA` module needs is the authority to write to the I/O port number 980 (`0x3D4` in hexadecimal system). The `Test2` module sees the `io.write.byte` procedure so it could give the `VGA` module this capability. However, with respect to POLA, it gives it a different capability:

```
\(value) = (io.write.byte 980 value)
```

This abstraction (unnamed function, lambda-expression) is given to the `VGA` module. It gives it the authority to write any `value` (of appropriate type) to the I/O port 980. The `Test2` module gives the `VGA` module few other similar capabilities. As a result, the `VGA` module has the authority:

- to write any byte to the I/O port 980
- to write any byte to the I/O port 981
- to read a byte from the I/O port 981
- to write any byte to the Video RAM (nowhere else)

This is enough for the `VGA` module to be able to provide expected services. Various abstractions created by the `Test2` module that act as proxies to more

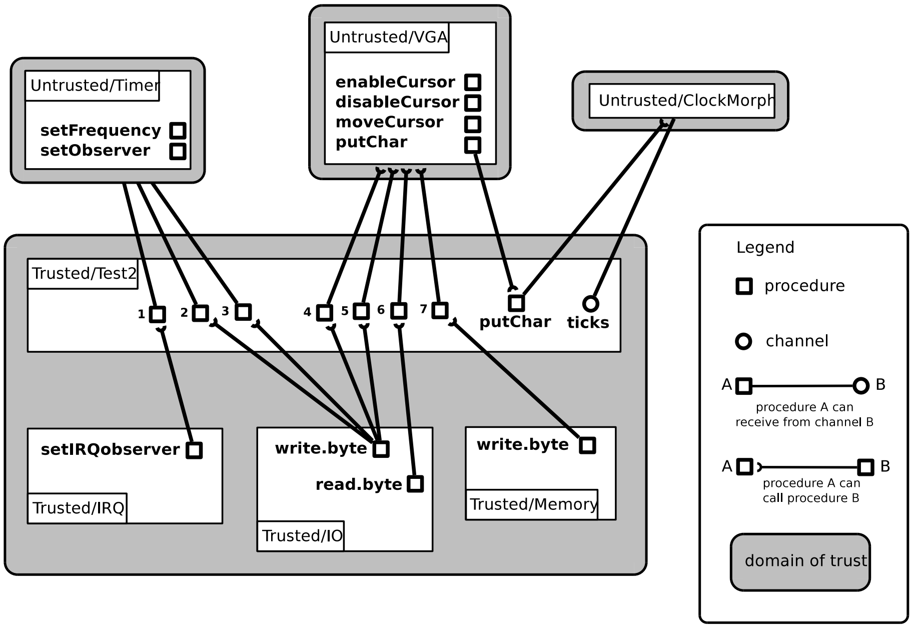

Fig. 5. The actual reference graph in the `Test2` kernel.

powerful capabilities are denoted as small numbered rectangles in the `Trusted/Test2` module in Figure 5.

If we have a complete and correct[6] knowledge concerning behavior of functions, procedures and processes in all the trusted modules (`Memory`, `IO`, `IRQ`, `Test2`), then we can safely assess the upper bound of authority of all untrusted powerboxed modules (`Timer`, `VGA`, `ClockMorph`) safely only with regard to the reference graph shown in Figure 5. It is possible because rules of allowed reference graph dynamics hold for our untrusted modules. In case of `VGA` and `Timer` modules the situation is trivial. We have complete knowledge about behavior of functions that we give to these two modules. Recall that we give the `VGA` module the following capability.

```
    \(value) = (io.write.byte 980 value)
```

Since we have a complete information concerning the `io.write.byte` function, we also have a complete information concerning the behavior of the above abstraction. So the authority of the `VGA` and the `Timer` can be determined precisely; regardless of their actual implementation. These drivers provide various functions. One of them is:

```
    (vga.putChar x y ch attribute)
```

[6] This is why we should try to keep the trusted computing base as small as possible.

When called, it puts any given character ch with any attribute anywhere on the screen. This is its assumed effect we believe it does.

In a similar way can we also give appropriate authority to the ClockMorph component. It is supposed to show the number of seconds from the boot-time in HH:MM:SS format. This kind of component obviously needs the authority to print eight consecutive characters somewhere on the screen. If we want to follow POLA also in this case, we have to implement a proxy function that will drop most of the vga.putChar authority and it will provide the ability to change eight consecutive characters on the screen, not more. We have defined the putChar function in the Test2 module that does exactly this. It relies on the vga.putChar function, see Figure 5. Regardless how perfectly our trusted proxy function implements additional restrictions, unless we verify the correct behavior of the original untrusted vga.putChar function, we cannot claim anything stronger than: "The ClockMorph component has as much authority as the VGA driver plus it can receive messages from the tick channel." But even with this simple technique, without studying the code of particular untrusted modules, we can see that the authority of the ClockMorph component is fairly limited.

7 Conclusion and Future Work

Our immediate goal is to address two immediate problems concerning minimal authority:

– there is no way how to give particular untrusted components only limited share of the CPU bandwidth
– there is no way how to give particular untrusted components only limited amount of memory

At present, when some of the untrusted components uses up the whole available memory, the runtime terminates the system. This is a show-stopper for using Pict for writing robust, from the traditional point of view, monolithic operating system kernels.

Sophisticated proof-techniques were developed for proving correctness of functional code, sequential (procedural) code. These can be very useful in analysis of procedures that are made visible to untrusted powerboxed modules. From the formally proved effects of these procedures (or processes) we can determine the authority of untrusted powerboxed modules that are part of the system.

Acknowledgments. This work was partially supported by the Slovak Research and Development Agency under the contract No. APVV-0391-06 and by the Scientific Grant Agency of Slovak Republic, grant No. VG1/3102/06. Thanks also go to our paper shepherd Frank Piessens whose insightful comments significantly helped to improve the paper.

References

1. Miller, M.S.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA (2006)
2. Pierce, B.C., Turner, D.N.: Pict: A programming language based on the pi-calculus. In Plotkin, G., Stirling, C., Tofte, M., eds.: Proof, Language and Interaction: Essays in Honour of Robin Milner, MIT Press (2000)
3. Spiessens, F., Roy, P.V.: A Practical Formal Model for Safety Analysis in Capability-Based Systems, Revised Selected Papers. In: Trustworthy Global Computing, Springer Berlin (2005) 248–278 Lecture Notes in Computer Science.
4. Stiegler, M., Miller, M.: How Emily Tamed the Caml. Technical Report HPL-2006-116, Advanced Architecture Program. HP Laboratories Palo Alto (2006)
5. Mettler, A.M., Wagner, D.: The Joe-E Language Specification (draft). Technical Report UCB/EECS-2006-26, EECS Department, University of California, Berkeley (2006)
6. Barnes, F., Jacobsen, C., Vinter, B.: RMoX: A raw-metal occam experiment. In Broenink, J., Hilderink, G., eds.: Communicating Process Architectures 2003. Volume 61 of Concurrent Systems Engineering Series., Amsterdam, The Netherlands, IOS Press (2003) 269–288
7. Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G., Larus, J.: Deconstructing process isolation. In: MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness, New York, NY, USA, ACM (2006) 1–10
8. Pierce, B.C., Turner, D.N.: Pict language definition. (1997)
9. Pierce, B.C., Turner, D.N.: Pict libraries manual. Available electronically (1997)
10. Wagner, D., Tribble, E.D.: A Security Analysis of the Combex DarpaBrowser Architecture (2002)
11. Stiegler, M., Miller, M.S.: A Capability Based Client: The DarpaBrowser. Technical Report Focused Research Topic 5 / BAA-00-06-SNK, Combex, Inc. (2002)
12. Pierce, B.C., Turner, D.N.: Pict libraries manual. Available electronically (1997)