EL: A Shell for the Ethos Operating System

BY

GIOVANNI GONZAGA NEBBIANTE
B.S., Politecnico di Milano, Milan, Italy, September 2011

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2014

Chicago, Illinois

Defense Committee:
Jon A. Solworth, Chair and Advisor
Chris Kanich
Stefano Zanero, Politecnico di Milano

To my family

To everyone that supported me in my studies

ACKNOWLEDGEMENTS

I'd like to express my gratitude to my advisor, Prof. Jon Solworth, who always supported me with his vast expertise and guided me along the way.

I'd like to thank the rest of my dissertation committee, especially Prof. Stefano Zanero, for taking time out of his busy schedule and providing valuable feedback.

I am thankful to have been given the opportunity to work with many skilled researchers in the Ethos laboratory, including Mike Petullo, Xu Zhang, Wenyuan Fei, Yaohua Li, Siming Chen. Their patience and always on point answers helped me a lot especially during the first months in the lab.

Special thanks go to fellow students and other friends I've been sharing this journey with. We made a great group for the last year and a half.

Another special thank you goes to our International Program Coordinator, Lynn Thomas. She's always been available with careful advice during the exchange program and followed me in the graduation process.

Finally, I would like to thank my whole family, for their patience and for always being supportive.

GGN

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In this thesis we present the design and implementation of El, a new shell and scripting language for the Ethos operating system.

The main goal of a shell is to bring operating system functionality and user space tools composition to the command line, and to provide a Programming Language (PL) for user space scripting. El aims to export an interface to the underlying Operating System (OS) that is as minimal as possible, making use of—and preserving—Ethos universal properties and abstractions. El is also intended to play a major role in Ethos user space programming, and thus is designed to overcome the issues of the major shells in use today in terms of PL abstractions.

The result is an inherently safer shell and scripting environment, where attack surfaces that are common for other shells are removed by design.

CHAPTER 1

INTRODUCTION

Shells are widely used by system administrators for managing their systems and application developers to compose functionality; in addition they are used for one-off tasks. A shell effectiveness depends on the smoothness of its integration with its underlying operating system and the simplicity of the interface exported by the operating system.

In this thesis we try to identify the weakest features of the shells in use today, and we propose a newly designed shell, El.

El is designed from scratch to be the Ethos shell and the preferred user-space language for scripting and small-to-medium size applications. In addition, given Ethos' security-first design, we focus in particular on security weaknesses in shells' design and implementation.

''*Although most users think of the shell as an interactive command interpreter, it is really a programming language in which each statement runs a command. Because it must satisfy both the interactive and programming aspects of command execution, it is a strange language, shaped as much by history as by design*''.

The above quote from Brian Kernighan and Rob Pike[1] summarizes a fundamental aspect of shells' design and evolution: shells were originally intended both for interactive usage at the command line and for scripting purposes; the reason why, historically, they have been particularly bad at the second class of tasks is due to the way the languages evolved, shaped by user requests and new implementations more than by design and evolving specification[1].

The question that comes to mind is thus: *Can a shell language be extensively used as a user-space/scripting programming language, without requiring the switch to more capable and structured PLs, and/or incurring in unavoidable limitations?* Various flavors of sh are successfully used for all sort of configuration and automation purposes, but as soon as the code base exceeds the few lines mark, programmers prefer (correctly) to use general purpose scripting languages. There is an undeniable tension among the two scopes (*scripting* and *shell*), and it's reasonable to say that shells have never been able to bridge this gap. With the newly designed El we (at least partially) accomplish this.

Hence, our discussion develops along three different main tracks:

G1: Ethos requires a different type of shell

Ethos *requires* a different type of shell in that the shell has to adhere to the OS

---

[1] The first *POSIX* standard for shells (IEEE Std 1003.2-1992) came in 1992, 20 years after shells first originated.

exported interfaces and must be integrated with them, especially with respect to composition of programs and access to file system objects. In the rest of this thesis, we describe how El preserves Ethos' properties by design.

We summarize goal **G1** as: *preserving Ethos universal security properties by language and interfaces design.*

G2: Ethos enables a different type of shell

Ethos *enables* a structured approach to Inter-Process Communication (IPC) and filesystem interaction with Ethos types. The well-defined object type semantics, applied system-wide and enforced by the OS itself, results in an inherently more secure (and smaller in code base) user space applications. This applies also to El code. In Chapter 2 we describe the Ethos security properties, with emphasis on the features El makes large use of.

Thus we summarize goal **G2** as: *exploiting Ethos security properties in order to obtain an inherently more secure shell.*

G3: General user-space programming

El has to enable the Ethos' user and user-space programmer to perform different tasks:

(a) interact with the system at the command line interface

(b) ordinary shell scripting

(c) small-to-medium size programming tasks that are too large or complex to be comfortably handled with ordinary scripts in other shells, due to the limited capabilities of code organization and lack of traditional PL features.

Item (c) goes beyond the features provided by the majority of shell languages in use today. Hence, our third goal **G3** is to *provide a language (and environment) better suited for general user space programming, with respect to other shells*.

As we'll discuss in detail in the rest of this thesis, goal **G1** and **G2** reflect in fundamental semantic differences with respect to usual shell interfaces and composition means. Goal **G3** affects El's design at different levels, including El's syntax, type system, composite types, error handling mechanisms and code insertion.

1.1 Related Work

We cite here previous related works in the category of shells and minimal scripting language implementations. The list is not comprehensive, given the breadth and history of the two research fields we are considering; nevertheless, we try to highlight the particular aspects that are more strictly related to El design.

UNIX was the first system to make the command interpreter an ordinary user process without special permissions. This led to various successive shell implementations, trying to improve the user-shell interaction. The first UNIX shell was Thompson shell[2], a primitive shell with only basic control structures and no variables. Thompson shell introduced the syntax for pipes and redirections, '|' '>' '<', adopted in syntax

and semantics by other shells until today. The Mashey shell[3] introduced simple text variables and the $ symbol to dereference them, and internalized some control flow constructs like if and goto that were previously implemented as external commands.

Two new shells emerged in the late 70s. The Bourne shell[4] introduced the Algol-inspired syntax that we still are confronted with today, and it became the default UNIX shell. The C shell[5] was far better as a scripting language than anything before, providing a more PL-oriented syntax resembling C's syntax, and can be considered the main ancestor of many following scripting languages. csh also introduced the concept of *builtins*, i.e. the idea of embedding the most commonly used utilities directly in the shell.

The Bourne shell and the C shell later developed into ksh[6] and tcsh[7] respectively. tcsh was a direct evolution of csh, introducing file name completion, command line editing, and other features that made better for the interactive, Command Language Interface (CLI) usage. The Korn shell (ksh) integrated many new concepts introduced by csh into the Bourne shell syntax.

rc[8] and its evolution, Inferno sh[9], two shells for the plan9 operating system, came with many innovative concepts. rc first introduced array variables, a cleaner separation between lexical and syntactical analysis, simplifying quoting and avoiding multiple scanning of the same input. Inferno sh is a more modular shell, where much of the functionality is loaded at runtime, including basic programming constructs. It also makes use of scoped exceptions for error handling.

The es[10] shell is an example of attempt at introducing cleaner PL semantics in the shell realm. es introduces functional language primitives into the shell. It allows code to be passed around as data. Traditional shells approximated this feature by passing commands as strings, but this resulted in unsafe and weird quoting rules. es has lexically scoped variables, first-class functions, and an exception mechanism. scsh[11] goes even further, trying to embed the shell into a functional language, Scheme.

The most innovative work on shells in recent years is Microsoft's PowerShell (PS)[12], currently in version 3.0. PS features a full-fledged scripting language, based on the .NET framework, and object manipulation capabilities, going beyond the UNIX model of text-based communication among entities involved in a computation. cmdlets (PS commands) are designed with a consistent interface, accepting objects in input (or as parameters) and producing objects in output. For instance, the pipeline is a programmatically-accessible entity, where well-formed, complex objects can be written to and read from. In Section 3.6 we contrast El design for typed object interaction with PS.

Despite some of the revolutionary concepts introduced for shells, the most widely used shells today (bash, zsh) are almost direct evolutions of ksh, with many improvements in terms of user-friendliness for interactive usage (powerful completions, history, customization), but still suffering from being unsuited as languages for scripting tasks in general.

1.2 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 introduces Ethos (and the kind of attacks it is designed to withstand to), with particular emphasis on the aspects more related to El. Chapter 3 examines some pitfalls in the design of shells and OS-/shell integration from the past, and provides rationale for the design choices we made for El. Chapter 4 is concerned with a more in-depth analysis of El implementation. In Chapter 5 we evaluate the results of our work. In Chapter 6 we conclude with possible future directions of development for El.

CHAPTER 2

ETHOS AND ETHOS TYPES

In this chapter we will introduce the Ethos OS, with an overview of the design rationale and enhanced security properties. Then, we will focus on a few particular aspects of Ethos' design and Application Programming Interface (API) that are most related to El's design and implementation: the type system abstraction and the Input/Output (I/O) and networking system calls.

2.1 Introduction

Ethos is a clean-slate OS, designed from the ground up to provide robustness and enhanced security guarantees. Ethos' abstractions and system calls interface are designed to ease secure application development and configuration. The API is designed from scratch, providing higher level abstractions compared to other OSs. This comes at the obvious cost of forgoing backward compatibility, in order to minimize complexity. Compatibility with existing standards otherwise consumes about 90-95% of code in a new OS[13].

Ethos targets a Virtual Machine (VM) instead of bare metal, and this has multiple advantages in terms of development and distribution. First, compatibility requirements, together with support for the majority of the device drivers, are delegated to

Dom0. Device drivers codebases are characterized by the highest bug density overall, up to three to seven times higher[14].

Second, running alongside another OS also means that Ethos does not need to support a full range of applications, as missing applications can be run on other OS. This eases adoption, as running Ethos does not preclude running other OSs—much like introducing a new PL.

Ethos aims to make applications more robust by providing high-strength security services (for authentication, authorization, isolation, and cryptography) and by minimizing complexity. Complexity arises in current systems both from the quantity of code needed to implement functionality (i.e., the attack surface) and from reasoning about the security properties of programs. Ethos reduces complexity by providing more abstract operations, with easier-to-reason-about failure modes and by providing "inescapable" protections—protections that applications cannot bypass. For example, Ethos provides encryption, authentication and authorization of all network connections without requiring specific per-application code.

The abstractions we consider in the following are the one most related to El design and implementation: the concept of types and the Ethos types infrastructure, including typed IPC.

2.2 Universal Properties

Here is an overview of the security properties guaranteed by Ethos, along with the security requirements they address.

Network Authentication

Every user is identified by an immutable Universally Unique Identifier (UUID). An UUID is assigned also in the case of a previously unknown or anonymous network user. This also addresses authentication needs normally left to application code.

Network Encryption

All application level network communications are encrypted on Ethos, in order to address data confidentiality and integrity requirements.

Type Checking

As discussed in more detail later, I/O for every application is subject to type checking performed by the Ethos kernel itself, in order to guarantee data integrity.

Key Isolation

User keys are never shared with applications, since encryption is handled directly by the OS.

Denial of Service (DOS)-resistance

DOS protection is built into Ethos network stack, increasing the chances that the system will continue to provide service to legitimate users even in case of abnormal consumption of resources by an attacker.

2.3 Types

Types and type checking have a central role in Ethos security-targeted design. Benefits of handling well defined types and formats are well known and can be applied to each system layer.

2.3.1 Rationale

Type systems are usually a matter of discussion in the context of PLs. Strongly typed PLs are intrinsically more secure, avoiding by design the possibility of type errors that can arise from unchecked usage of unsafe languages. Type safety prevents *untrapped errors*—errors that goes unnoticed and don't stop execution—and can reduce *trapped errors*—errors that are detected and causes execution to stop. Untrapped errors are especially a risk factor, since can result in unpredictable execution behavior, and can be exploited by attackers to produce arbitrary behavior in an application[15].

In order to exploit the same typing benefits in OS development, the OS itself can be built using safe/typed PLs for what possible. Examples are various OS/kernels written in Java[16] or Haskell[17]. Other ways of providing type safety in OS kernels use static type checking and formal verification of existing code bases[18].

The Ethos way is different: the OS is built in C, a statically typed but non-safe language; applications are written in Go, a strongly typed language. Typed entities are instead defined by system-wide, cross-language type definitions. Every filesystem

or network object on Ethos is of a specific, well defined type. A type hash is applied to directories, IPC streams and network streams. Type checking is centralized, applied at kernel level. Ethos is thus focused on extending this type of consistency (benefits of PL type systems that are usually in terms of internal consistency) to interaction across multiple applications, that are (by OS design) written themselves in type-safe languages[19].

2.3.2 Ethos Types

There are two kinds of representation for a typed object: one as a runtime instance—in a running program's memory—and the other as a serialized object, whether it persists somewhere in the filesystem or "in transit" in the case of network communication and IPC.

Ethos subjects all the network communications and filesystem I/O to the type checker. The allowed types for an IPC or filesystem object is specified at the directory level: each directory has a *type hash* associated with it, the type hash uniquely identifies the object type system-wide. Thus, a directory with type associated T can only contain files of type T. IPC types are specified again making use of a typed directory as the *service directory*. Only objects of the specific type are allowed to be read/written from/to such stream.

The Ethos Types infrastructure is made of different components:

1. the *type checker* is part of the kernel, and responsible for allowing or not each I/O operation;
2. the type checker makes use of the *type graph*, a special object that contains the hash and type description for every type known to the system;
3. system programmers can define new types at compile type using Ethos Type Notation (ETN) description files, or use primitive types handled out of the box. New types are first installed on the system's type graph.

In the following, before to delve into the Ethos Types infrastructure details and ETN, we first give an overview of the Ethos Types checker properties and of the scope of serialization.

2.3.3 Ethos Types Properties

A system call that writes an object will succeed if the object is of the correct type (w.r.t. the destination) and well-formed, or fail and return an error. Similarly, if a read system call succeeds, the returned object is guaranteed to be of the right type and well-formed.

The property assured by the Ethos' type checker, *object integrity*, is defined as:

1. objects are read or written as a whole;
2. an object—either in its external or memory representation—must always be consistent with its type;

3. an object exchanged between two programs must produce an *equivalent object* when is read by the receiving program.

2.3.4 Serialization

Serialization is the process of translating a given object or runtime state o in a format suited to be stored and/or transmitted, and transformed back into its original runtime representation—or an equivalent one—later on, yielding o'.

A (de)serialization process must guarantee the property of semantic equivalence between the original object o and the de-serialized one o'.

The resulting representation of o' will in general be identical in case of source and destination runtimes of the same nature, while it might differ substantially when the interacting systems are run in different environments. In any case, what has to be preserved is the semantic equivalence of o and o', a property defined by the both the serialization specification and the actual implementation(s).

Examples of serializers tied to specific languages are the Java Serializable[20] interface implementation or Python's pickle module[21].

The need for language independent serialization formats became more and more relevant during the years as the nature of systems evolved from centralized, same-environment, same-language systems to distributed systems involving possibly many different PLs and environments.

Examples of language-agnostic serialization libraries are Google's Protocol Buffers[22] and Apache Thrift[23]. Both provide type (and service) definition means, and generate code targeting multiple languages, that the programmer can use in order to create, encode, and exchange objects in distributed systems.

2.3.5 ETN

Ethos defines its own serialization format, and the Ethos type infrastructure makes large use of it for all sorts of I/O operation. ETN defines a syntax for description of types and Remote Procedure Call (RPC) services. A description file is typically application specific, and contains definitions for types and RPC services that the application or various interacting parts of it make use of. From the description file, Ethos:

1. builds a type node for every type, including types that are referenced by other types,
2. creates an unique hash for every type,
3. installs the newly defined types in the system,
4. generates code targeting Go or C to access objects of the created types,
5. generates code stubs for the defined RPC interfaces.

Installing the new types in the system's global type graph allows Ethos to always have available the type information (and hash) for every user defined type, thus being able to perform type checking on I/O operations involving every single type.

TABLE I: ETN BASE TYPES

int8	uint8	bool	int16
uint16	int32	uint32	int64
uint64	float32	float64	string
any	union	tuple	array
dictionary	struct	interface	method

ETN is language agnostic, as the Ethos types infrastructure is able to generate code in both C and Go (the two system programming languages supported on Ethos), that under the hood share the same binary encoding for transmitted objects.

User-defined types are built aliasing or mixing in composites of the predefined ETN types. A list of ETN base types is presented in Table I. In Figure 1 we provide an example ETN description file. The file defines two types, ‘Message’ and ‘User’, both are ETN structs. ‘Message’ describes a generic message, referencing the ‘User’ type for the ‘To’ and ‘From’ fields. ‘string’ is an ETN base type. From this ETN description file, two type nodes are thus created and added to the system’s type graph, ‘Message’ and ‘User’, the former referencing the latter.

2.3.6 Type Checking Objects

Ethos ensures that every file write must go through the kernel for type checking. Ill-formed objects are stopped during a write operation, which fails. Every object contained in a filesystem directory is of the same type, and every filesystem directory

```
1  Message struct {
2      From         User
3      To           []User
4      Subject      string
5      Message      string
6  }

8  User struct {
9      Username     string
10     Host         string
11 }
```

Figure 1: Example ETN description file

has an associated type hash, determining this type. The same mechanism also applies to IPC, since Ethos IPC services are named by filesystem paths.

The type for a directory is specified upon creation; the Ethos CreateDirectory system call accepts a type hash to be applied to the directory path. For situations where different types need to be mixed inside the same directory, ETN Union and Any types provide a solution, at the obvious cost of handling union tags or castings for the programmer.

2.3.7 Files and Directories

Ethos file objects are designed to be read or written entirely—Ethos doesn't support seeking for files. This follows naturally from the fact that file objects have well-defined types associated: supporting streaming for some specific types (e.g.: array-types could

TABLE II: ETHOS READ/WRITE SYSCALLS

	read	**write**
var	readVar(descriptor, name)	writeVar(descriptor, name, content)
streaming	read(descriptor)	write(descriptor, content)

naturally support streaming access) would compromise the general design, and complicate error recovery.

On UNIX, it is common to stream to a file, optionally appending, or redirecting output from a command execution. Files are streaming entities on UNIX. The equivalent notion of streaming entity on Ethos is instead the directory. Directories support both the *var* read/write—where the object read/written has an arbitrary name—and the *streaming* read/write—where the object read/written has no specified name, and Ethos itself takes care of naming as to preserve the write-order for subsequent reads. These two modes reflect in the system calls reported in Table II.

Directories also solve naturally the problem of representing large files, the kind of file types for which one would expect to be able to seek and access the content in chunks. These can be persisted as multiple objects inside the same directory and streamed in order (or accessed randomly by name).

2.3.8 IPC

IPC on Ethos is established making use of Ethos I/O (Table II) and Networking (Table III) syscalls:

1. a service is advertised by the “server” component, identified by a fd (the streaming directory used to establish the IPC) and a name
2. the “client” component can ipc through the advertised service, and is automatically authenticated, authorized, and the connection encrypted if IPC happens over a network; the host is left out in case of local IPC;
3. the server component can then import an incoming IPC, using the listeningFd returned by the advertise call.

   The ipc and import calls return a file descriptor each, representing respectively the write and read ends of the service. The client component can then write to this fd, and the server component read the stream of written objects in order at the other end.

TABLE III: ETHOS NETWORKING SYSCALLS

Call	**UNIX Equiv.**	**Semantics**
advertise(fd, name)	listen	Services have an associated fd and are named by strings instead of ports
import(fd)	accept	Ethos adds authentication, authorization, connection encryption and DOS protection.
ipc(fd, name, host)	connect	Ethos adds authentication, authorization, connection encryption and DOS protection.

CHAPTER 3

DESIGN

In the following we present the main design challenges and discuss how we tackled them with the design of El. This chapter is thus focused on multiple issues, listed here alongside the main goal to which they refer (as per the goals definition in Chapter 1).

1. the design of a shell and programming language, which in turn means the design of the integration with the underlying OS (**G2**, **G1**) as well as the supporting PL itself (**G3**);
2. combine the two intended usages for El (scripting applications and the CLI interaction) in a successful manner—something other shells/shell languages lack (**G3**);
3. integration with the security mechanisms provided by Ethos—especially the ones we described in detail in Chapter 2—in a way that doesn't compromise the assurances they offer (**G1**);
4. reduce complexity to the minimum possible, so as to equip the end user with easy-to-reason-about semantics, that in turn affect positively the exposure to security-related pitfalls of programs and systems built with El (**G2**).

First, we define a few terms we'll make use throughout the rest of this document.

3.1 Preliminary Definitions

In the following, we use the term sh to refer indifferently to the original Bourne shell or any of the modern descendants of the same family, as described in Section 1.1. We'll explicitly refer to some implementation when needed.

The term *scripting language* is used to refer to a general purpose, interpreted language. Mainstream examples of the family are Python and Ruby. We'll use the term *scripting language* in contrast to *command language* or *language for CLI*, the category of special purpose, shell languages. In this category fall for instance the original Bourne shell and bash.

3.2 Scripts vs. Command Line

There is a tension between a shell language and a scripting language. More generally, there are substantial differences between the scope and ways of interaction in a CLI vs. full-fledged applications made of (possibly multiple) script file(s). These include:

1. Shell languages have minimal syntax and are able to scale down to terse one-liners; scripting languages are instead able to scale up to bigger programming tasks, leaving aside compactness.
2. Scripting languages are typically dynamically typed, shell languages instead provide an (almost completely) untyped definition. The lack of type checking in shell languages can be source of security vulnerabilities. This difference also reflects

up to the syntax: scripting language types are not static and thus, for instance, a specific per-type literal is needed in order to differentiate primitive types.

3. Shells are good at so called *dataflow* programming, where a program is modeled as a sequence of connections among operations, applied as soon as the input becomes available. The dataflow model of UNIX shells is simply a linear sequence of piped operations. Scripting languages don't usually address the dataflow needs; they instead provide different forms of abstract concurrency models, in the classic Von Neumann architecture[24].

3.3 Design Rationale

El aims to be the shell language (and shell) for Ethos primarily, but also to be able to scale up to bigger programs as a scripting language would. In the following we discuss how these requirements reflect in different aspects of the design.

3.4 Syntax

As discussed in Section 3.2, El has to be able to scale down to terse one-liners, and at the same time scale up to small to medium complexity tasks. The ability to scale in the context of small code bases is mainly driven by two factors:

1. the power to split the code base and organize the code, export and require single pieces of encapsulated functionality (for the scale-up part), and
2. the terseness and brevity of the syntax (for what concerns the scale-down part).

Terseness and brevity have thus to be supported directly by El's syntax. Since Go is Ethos' primary user-space system programming language, El syntax borrows from Go syntax[25].

El syntax is, I believe, far less ugly than sh. The reasons why sh syntax is universally considered dirty are historical ones. Bourne shell syntax was modeled after Algol, and that's where many weirdnesses come from, like the use of reversed keywords to mark the end of a construct (e.g.: if ... fi). This rule in particular has exceptions (as in do ... done, since od was already taken as a keyword being the name of an executable) that make it more difficult for a beginner to grasp the syntax.

As an additional example of what is generally considered ugly syntax, consider the way if conditions are expressed in sh. As shown in Figure 2, the syntax for the AND operator (Figure 2-a)—other than ugly by itself—is completely different from the syntax for the OR operator (Figure 2-b). This breaks the fundamental rule of *predictability* that every API or syntax design should guarantee. Note that there are valid reasons why the example works this way[1],what we want to point out is the fact that sh isn't a general purpose scripting language, and constructions like this are a messy attempt at making it look like one.

---

[1] An additional remark of the fact that sh is in fact a special purpose language, meant for assembling commands in different ways: [and /bin/[are both synonyms for test, implemented as builtins. Thus what the if construct does in the end is just checking the return status of the "condition". The additional closing square bracket is an extension to the bash grammar in order to make it look more familiar to programmers from other languages.

```
1  if ["$foo" = "bar"
2    -a "$n" != "john"]
3  then
4    # ...
5  else
6    # ...
7  fi
```

(a) And “operator” in if condition

```
1  if ["$foo" = "bar"]
2    || ["$n" != "john"]
3  then
4    # ...
5  else
6    # ...
7  fi
```

(b) Or “operator” in if condition

Figure 2: sh syntax examples

Although the initial syntax design can be considered flawed, the Bourne shell was so innovative and good at performing its tasks that the syntax stayed almost unchanged in sh more recent incarnations. New shells of the family still support the majority of the original Bourne syntax, if for no other reason than for backward compatibility.

The fact that sh syntax is inappropriate for common scripting needs can thus be ascribed to the design choice of making it (very) good at CLI interaction mode—and CLI only.

El is instead designed with both objectives in mind, and thus provides both a brief syntax for CLI interaction and a more sane set of constructs for the scripting usage.

In Section 4.2.1 we describe how we handled this requirement at the parser and evaluation levels. The complete grammar for El is reported for reference in Appendix A.

3.5 Types

The most compelling advantage of typed variables is that they permit a system to trap errors. Type safety removes *untrapped errors*—errors which are not detected/reported and for which the execution continues as if no error occurred.

As Milner once stated, "*well typed programs can't go wrong*"[26]. Even if certainly an exaggeration, benefits of typed languages in security-sensitive contexts are plain for all to see. A strongly typed language at least reduces the effort required to build a safe system, where safe is intended with respect to untrapped errors.

sh variables are character strings, both in representation and semantics. In order to slightly lighten the burden of keeping track of the semantic type of a variable, bash offers the declare (aliased typeset) builtin.

declare applies special properties to bash variables, like "readonly", "integer", "array". For instance, declaring a variable as integer, bash will treat subsequent uses of the variable as a numeric integer value, allowing or not certain arithmetic operations on it. The extent of typing obtained is still far from what one would expect in other PLs: more than limited in number of available properties, the mechanism doesn't really provide any guard against untrapped errors. For example, assigning a string value to a previously integer-declared variable will result in no error reported (execution continues, making the (possible) error *untrapped*), moreover re-setting the variable integer value to 0 (arbitrary side effect).

El variables are instead typed. El type system is purely dynamic, in that the type of an El variable or expression is something well-defined and known at runtime only. Each operator is defined for a specific subset of types, and applying an operation on objects of non-matching type will result in *trapped* errors.

El's type system is small by design, resembling other "small" scripting languages like Lua ([27]) or JavaScript. El's available types are actually more than JavaScript ones, for example JS provides a single, floating point numeric representation (Number), and no integers. In addition, unlike JS, El's is lexically scoped (there's no way, accidentally or not, to pollute an outer scope).

References are defined at the first assignment, and have the type of the expression on the right hand side of the assignment. Thus we can say that a variable belongs to the scope where it is first assigned. Types are not declared statically, they're inferred from literals, and propagated throughout expression evaluation. Although practically there's no such thing as "variable declaration", given that every symbol is declared at first assignment, El has a form of variable declaration statement, in order to be able to bind a symbol to a specific scope.

References to undeclared variables (symbols never assigned), as well as references to variables declared without specifying any value, are nil values. Any operator applied to a nil value will result in an exception (trapped error).

3.6 Typed I/O

Pipes, redirections, and, in general, the idea of communicating through streams of text, are certainly one of the major UNIX contribution to the history of OS interfaces. Many UNIX commands take text-like input and/or produce text-like output, allowing the user to compose pipelines of made up of separated computing pieces, streaming between each other (and executing a parallel fashion).

These tools are incredibly powerful in the hands of users/sysadmins. Although, this power comes at a cost: the text interface has revealed extremely general but at the same time doesn't define any structure for the exchanged data, causing many different standards to arise. This in turn means the handling of the text input is delegated to each single application, for which programmers have rewritten countless different parsers. Many of these targets the same "data type", possibly causing the coexistence of different implementations at the same time in the same environment, often with subtle mismatches in the respective behavior.

The free form nature of the streamed data also aggravates the problem of trust of the source of the input[1]. Having no guarantees on the form of the input—no other entity enforces a structure or checks for ill-formed data prior than the destination program itself—makes heavy sanitization necessary *inside* every application, especially

[1] More specifically, the source and all the other entities (communication channels, transformations) involved between the source and the final destination.

when the final destination of this input is—or influences in some way—executed code. Consider the following example (this has been the only way to install the official package manager for one of the most widespread web server development environments nowadays[1])

```
curl http://npmjs.org/install.sh | sudo sh
```

The short one-liner contains many of the bad practices described before, like trusting executable code piped straight into a subshell (run with superuser privileges), moreover on a insecure connection. Piping to the shell also has another fundamental flaw, related to unexpected premature ending of the pipe stream[2].

El is designed to retain the flexibility of UNIX pipes, without give up on the advantages Ethos offers in terms of typed communication—instead making great use of Ethos' typed IPC in order to provide a simpler interface and additional security guarantees. Ethos' (and consequently El's) design impact on the fundamental flaws we just described in the following ways (which we detail more accurately in Section 5.1).

---

[1] Reference: https://npmjs.org/. Recently, the install process has been fixed, first switching to https and finally shipping the package manager along with the core package binaries, avoiding the problem altogether.

[2] If the connection closes mid stream, sh will execute the partial script in its buffer. In this (very unlikely, but still possible) case, what if the script is interrupted in between a critical operation composed of multiple commands, or if a truncated command results in a dangerous one?

Avoid in-application parsing

Applications (and scripts) don't need to rewrite their own parsers, as they make use of Ethos' parsers.

Single decoder definition system-wide

The parser definition is unique system-wide, and thus cuts on possible mismatches deriving from subtle semantics differences between one implementation and another.

Object integrity

Ethos takes care of checking object integrity. With a stream of well-formed objects, El is not subject to possible errors deriving from an unnoticed early connection end.

Support for typing and typed objects I/O in El shares some similarities with PowerShell's one. PS avoids custom application parsing of text based on consistency in naming rules among different cmdlets, so that the output from one cmdlet can be used as the input to another cmdlet without reformatting or text manipulation. PS objects are runtime .NET entities.

The two designs differ significantly in that El manipulates objects that are defined at the OS level. Their runtime representation is unique and defined by the Ethos type definition. On Ethos, the whole concept of types for filesystem and IPC objects is OS-defined, as opposed to defined by a particular runtime framework.

3.7 Functions and Scope

Here we list the main flaws in sh (particularly bash) function design and scoping rules. We already discussed improvements introduced in various implementations for what concerns both these aspects in Section 1.1.

Return values

Bash functions don't have return values; they only produce output streams. Every reasonable method of capturing that stream and either assigning it to a variable or passing it as an argument requires a subshell, which breaks all assignments to outer scopes. Strategies for returning values other than the status (success or failure) include: setting a global variable with the result; use command substitution; pass in the name of a variable to use as the result variable.

Function arguments

You can't pass arguments "by reference" either. Working with arrays is even worse— the best you can do, typically, is to pass each array element as a separate argument. This means libraries of nontrivial reusable functions are not feasible, except by performing eval back-flips.

Scope

Bash has a simple system of local scope which roughly resembles "dynamic scope". Functions see the locals of their callers, but can't access a caller's positional parameters. Reusable functions can't be guaranteed free of namespace

collisions unless you resort to weird naming rules to make conflicts sufficiently unlikely.

Closures

In bash, functions themselves are always global (they have “file scope”), so no closures. Functions are not “first-class”—you can’t assign functions to values, or pass them as parameter (except by resorting to ugly string-and-eval hacks)—and there are no anonymous functions—which would be useless anyway given the missing local scope. bash uses strictly call-by-value semantics.

El aims to simplify and enhance at the same time scoping and functions rules, as to resemble common scripting languages in behavior. El’s functions are first-class citizens in the language—that is, they can be assigned to identifiers or passed as parameters as any other value could—and can be anonymous too. This should at least encourage a more functional approach to programming, and simplify code reuse and separation “in-the-small”.

3.8 Error Handling

sh/bash error handling mechanisms integrate with UNIX status codes. An executable exiting with a non-0 status can for instance stop a list of commands from executing (using the ‘&&’ operator). By default, simple commands failing don’t cause the execution of the script to stop, although this can be obtained with ‘set -e’. In addition, bash offers traps, signal handlers defined on the “global” scope. traps can be

used to handle both error conditions in a bash scripts and OS signals. The mechanisms is not general in that a trap is defined at the top level, and there is no way to define a trap with limited scope. For example, defining a trap for a certain signal that is also handled in a “sourced” script, results in re-definition (overriding) of the same trap.

Ethos too provides a large set of status codes, including insufficient permission, resource limitations, or invalid operation. An ok status code indicates success.

Non-ok simple commands cause an error exception to be thrown. El’s equivalent of traps are exception handlers. An El exception handler creates a new scope and defines the handler for that scope only. An error exception can be caused either by non-ok return status of executables or by explicit calls to panic. Any not-ok status resulting from a command invocation can be explicitly handled, based on the exception value.

An exception is handled by a catch block, defined at any arbitrary scope depth. If, unwinding the stack, no user-defined exception handler is found, the exception is unhandled (catched by El’s global handler) and causes execution to stop. This is in contrast with bash where a non-ok result for an external command invocation won’t stop execution by default (without the ‘set -e’).

3.9 Packages

One of the main limitations that prevent sh to be used as a general purpose scripting language is the lack of a structured way to separate code modules, or export functionality as a library would usually do.

sh/bash has to main ways of including/executing external code, here we describe why they're unsuitable for any sufficiently advanced code separation need. For a thorough description refer to Section 5.1.2.

source (or '.')

source <filename> (or the equivalent '. <filename>' notation) evaluates filename line by line, in the current shell environment. In other words, it is equivalent to adding filename's content to the current script (or typing its lines one by one at the shell). Thus, the included code has full control on the including environment and, vice versa, the calling environment can modify in unexpected ways the execution environment of the included script.

sh/exec

'sh filename' (or simply 'filename.sh') executes filename in a subshell (new process). Thus filename executes in a different environment, and information flow back to the caller is awkward, except for return status or evaluating the stdout with command substitution.

For El code separation we designed a simple package system. Library users are able to 'require' packages (El script files), that are evaluated in their own separate environment and explicitly accessed for functions calls and variable references. A package can in turn decide what to export to the "outside world" making use of the 'export' keyword. We discuss how this reflects in terms of goals **G1-3** in Section 5.1.2.

3.10 Unifications

El abstraction power and integration with Ethos are tightened by the concept of *unifications*. In our design, unifications are interfaces that have more than one implementation, but exposing the same kind of interaction for the end user.

3.11 Directories and Maps

The first dual we describe here is the one involving the file contents of a directory and El maps. A map variable may in general refer to a directory file contents or be independent of the filesystem.

Let 'T' be the type of a directory's elements. A directory is accessed in El as a map, with a range of 'T'. Operations on maps are shown in Table 2. Directory-backed maps can be accessed and iterated like standard maps, although their implementation differs. For instance, each access on a directory-backed map is a file read, each assignment is a file write.

Treating file directories as maps facilitate common OS tasks. For example, Ethos does not have any built-in support for environment variables; instead the Ethos environment variable passing equivalent is a directory which contains key-value pairs, represented as string files.

3.12 Functions and Executables

We designed El functions, builtins and executable invocation as to provide a consistent interface shared across the three. Syntax for function invocation allows for

executable-like free form parameters; a function can locally override an external executable using the same name. As discussed , current support for this unification is limited by a few main factors: argument passing for executables is not typed yet (i.e. executable still receive string valued parameters), and functions in pipeline statements don’t have the ability to interact with the pipeline objects stream.

CHAPTER 4

IMPLEMENTATION

In this chapter we present the main implementation challenges and discuss the choices made in each case.

First, we provide an overview of the language in Section 4.1. In Section 4.2 we first give an high-level description of the codebase, and then discuss specific implementation challenges and relative solutions.

4.1 The Language

The complete El grammar can be found in Appendix A. Here we present an high level overview of literals, identifiers, assignments, value accessors, control flow constructs, functions and builtin functions.

We start off with a list of valid commands (presented in Figure 3) that should give an idea of the syntax.

Some of the lines shouldn't be surprising, others deserve more explanation. At line 2 we are redirecting a single string ('foo') to a directory ('/user/jon/strings') and executing in background. At line 4, the output of 'ls' for the current directory is piped to count. The resulting pipeline stream (composed of a single int object in this case) is then collected into a tuple, and its first element is accessed and assigned to the variable '$n'. In lines 6-8 we are iterating over a 'tuple' literal defining a tu-

```
1  echo Hello world
2  echo foo > /user/jon/strings &

4  $n = (ls . | count)[0]

6  for $k in [a, b, c] {
7      echo $k
8  }

10 $bob = /user/jon/contacts/bob
11 $msg = new Message{To: $bob, Message: Hello}
12 /user/jon/messages/msg = $msg
```

Figure 3: El syntax examples

ple containing three strings. In lines 10-12 we are accessing a file system object (‘/user/jon/contacts/bob’), creating a new runtime object of type ‘Message’ and storing the created message at a specific location in the filesystem. For this lines to work as expected, the objects involved need to be of the correct types, i.e.: ‘$bob”s type must match the type of the ‘Message.To’ field, and the type of the ‘/user/jon/messages/’ directory has to be ‘Message’.

In the rest of this section we describe El’s syntax in more detail. Syntax for primitive and composite types is presented in Table IV. The path literal is a special case: as we’ll discuss shortly, based on the context it can be evaluated either as a path node or as a plain string literal.

An identifier is a name matching the regular expression

TABLE IV: EL LITERALS

Type	**Literal**	**Example Usage**
all *Integer*s	n	42
all *Float*s	n.d	3.14
String	s or "s"	foo or "el"
Bool	true or false	true
Tuple	[e_1, . . . , e_n]	[1, 2, 3]
Map	⟨ k_1:e_1, . . . , k_n:e_n⟩	⟨ age: 30, height: 80⟩
Set	{e_1, . . . , e_n}	{1, 2, 3, 5, 7}
path	path	./a/valid/path

\$[a-zA-Z0-9_]+

i.e. starting with a $ and composed by one or more alphanumeric characters. Words starting with $ are always substituted as identifiers, except for when they appear quoted, as in "$$id$".

4.1.1 Types

El is dynamically typed. El's readily available types are of two kinds: *primitive* types (such as int, float, . . .) and *composite* types (such as tuple or map). Composite types may hold values of any other type In addition, El can handle any user defined type known to Ethos. The way objects of different Ethos types are mapped to language objects vary with respect to the type considered. El's primitive types are in a one to one mapping with primitive types. Composite user defined types are instead

mapped to a single El type—object—and type specific checking is handled by means of introspection.

El thus defines a number of primitive types, plus three composite types (tuple, set, map). These constitute the core type system, for which El's operators are defined. Some operators are then extended to work on arbitrary user-defined object types (this is the case of the '[ ]' access operator, extended to work on generic slice objects, and the '.' value access operator, extended to work on generic struct objects).

The list of operators on primitive types, along with the types they can handle, is shown in Table V. A simple set of upcasting (generalization) rules is defined for numeric types. Operators can thus work on same-type objects or on objects for which a valid generalization can be built, such that the obtained upcasted values are of the same type. Such type coercion mechanism is built walking a lattice on which the valid generalizations are encoded. The reference lattice is shown in Section 4.1.1.

The target type for the upcasting algorithm is the lowest possible (in terms of lattice representation) common ancestor. At the lattice top we can find the any type. Restrictive types cut down on errors, in exchange of reduced generality. An any-typed object instead can contain any value, and provides the right type interface for generic tools, as we describe in Chapter 6.

4.1.2 Assignments

An assignment has the form

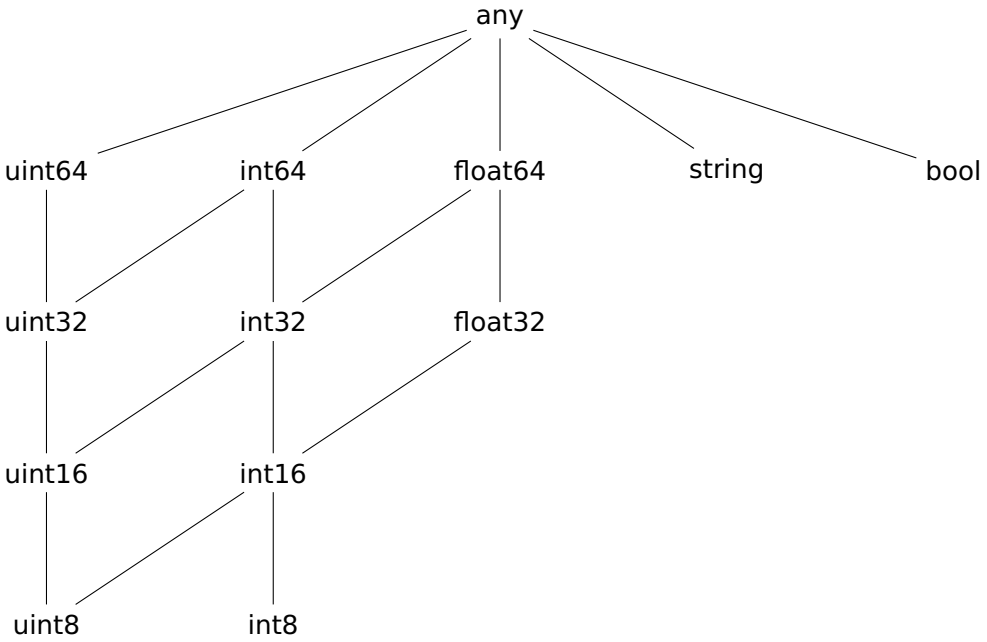


Figure 4: El primitive types lattice

a = expression

where expression is a generic expression, and a can be an identifier, an accessor (see below) or a path literal. In case of a path literal the assignment goes beyond the runtime boundaries, trying to write the expression value to the specified filesystem location. The conditional is particularly necessary as the actual write are successful only when the destination object type and the runtime object type match—being the destination object type the type applied to the containing folder. As detailed in section

Section 4.1.1 and Section 4.2.5, El can handle objects of primitive types (mapped to ETN primitive types) and objects of arbitrary, user-defined types.

A path literal is also a valid expression: evaluation of a path literal consists again in transcending El's runtime and reading the specified object from the filesystem.

Wrapping up this brief discussion of assignments and path literals, here is an example of how it is possible to copy a filesystem object to an alternative location

```
/path/b = /path/to/object
```

It is not recommended in general to use the assignment construct for a simple copy operation, as it involves superfluous (un)marshalling of the same object from and to the filesystem. The use case is for instance creating copies of modified objects, consisting in reading the object, partially modifying the value, and writing back to the desired location.

4.1.3 Accessors

We describe two different kinds of accessor: composite type accessor, using the square brackets syntax, and value accessor, used to access values exported by modules and fields of struct objects.

A square bracket accessor looks like

```
c[index]
```

where c is an El composite type (tuple, map), a string or a slice object (refer to Section 4.2.5 for details), and $index$ is a generic expression evaluating to an int value—in

order to access tuples, strings or slice objects—or to a string value—accessing a map element.

A value accessor is in the form

```
v.val
```

or

```
v.fun()
```

where v can be the reference to a required module, and thus both the forms ‘v.val’ and ‘v.fun()’ make sense in general, since modules can export values and methods. Alternatively, ‘v’ can be a struct object, and in this case only the ‘v.val’ form is valid, and serves the purpose of accessing a specific struct field. As already discussed in Section 3.9, the validity of an access to a package values or methods is controlled by the package itself, using the ‘export’ keyword. Evaluation of an package’s value or function call occurs in the package environment, isolated from the current executing environment. A new evaluation environment is created for every ‘require’. We conclude with an example usage of a value accessor, where we make use of an exported method of the math package.

```
$max = require(math).max($a, $b)
```

4.1.4 Control Flow

El’s control flow statements are similar in syntax and semantics to the Go ones, since that Go is the primary application programming language on Ethos. El has two

TABLE V: EL OPERATORS FOR PRIMITIVE TYPES

Operator	**Meaning**	***Int***	***Float***	***String***	***Bool***
+	add/concatenation	✓	✓	✓	
-	subtraction	✓	✓		
- (unary)	unary minus	✓	✓		
*	multiplication	✓	✓		
/	division	✓	✓		
=	equality	✓	✓	✓	✓
!=	inequality	✓	✓	✓	✓
<	lower than	✓	✓	✓	
>	greater than	✓	✓	✓	
<=	lower equal	✓	✓	✓	
>=	greater equal	✓	✓	✓	
||	logic disjunction				✓
&&	logic conjunction				✓
!	logic negation				✓

kinds of for loops: the usual C-like iteration (Figure 5-a), defined by an initialization, an iteration step and an halting condition; the iteration on items of a sequence (tuple, string) or contents of a map. In the example in Figure 5-b, we iterate over the contents of a map accessing the string keys (‘$k’) and the value each key is pointing to ‘$v’.

The ‘if’ statement (Figure 6-a) is composed by the a condition, the ‘then’ block, and the optional ‘else’ block.

The ‘switch’ keyword defines a switch/case statement, as shown in Figure 6-b.

```
for $i = 0; $i < $n; $i++ {
    print $k
}
```

(a) C-like loop

```
for $k, $v in $some_map {
    print $k
}
```

(b) Range loop

Figure 5: El loop constructs examples

```
if $a && $b {
  ...
} else {
  ...
}
```

(a) if statement

```
switch $name {
case "alice" : {}
...
else : {}
}
```

(b) switch statement

Figure 6: El if and switch constructs examples

Additional control flow keywords are ‘break’, ‘continue’, ‘return’, all carrying their common meaning, and ‘panic’, used to raise error conditions as discussed in Section 3.8.

4.1.5 Functions

El’s function definitions create new scopes, that along with the function signature and body themselves, constitute the function closure: a function with associated referencing environment. Thus, a value of type function (as discussed in Section 3.7, functions are first class objects in El), is represented by the definition—in form of a

reference to the definition Abstract Syntax Tree (AST) node—and by the referencing environment—a reference to a copy of the environment at definition time.

Functions are defined using the keyword func, in one of the following equivalent forms

```
func name($a, . . . )
$name = func($a, . . . )
```

The first form can be considered syntactic sugar for the second, since functions are values and thus a function definition is a valid expression. In both cases, the $name identifier will reference a closure, that can be invoked equivalently as

```
$name($v, . . . )
```

or

```
name($v, . . . )
```

or

```
name $v . . .
```

The third form is equivalent to the command execution syntax, and thus may come in handy to locally override an executable, or wrap a long command execution exposing variable arguments only.

4.1.6 Builtins

El builtin functions are a small but easily extensible set of predefined functions. Builtins can be overridden by user function definitions, that is: the function lookup searches for function definitions in scope first, and for builtin functions only if there are no functions defined with the given name. In case of command-like style of invocation, El performs an additional lookup step for commands with the given name. For example, for the following call

```
echo something
```

the lookup process looks like:

1. search for functions in scope named ‘echo’;
2. if not found, search for builtin functions named ‘echo’;
3. if not found, search for programs named ‘echo’;
4. if not found, raise an exception.

Here follows a (non comprehensive) list of El builtin functions.

print(format, params...)

Prints to (untyped) stdout given the format string and a variable number of parameters.

scan(format, params...)

Reads from (untyped) stdin a series of parameters, based on the provided format strings, and yields a typed value for each parameters.

sprint(format, params...)

Formatted print to a string. Returns the printed string.

require(pkgname)

Requires a package and returns the package reference, if found. If a relative path is given, require looks up for packages in the current working directory or in /user/scripts.

type(value)

Type introspection feature. Returns the string representation of the type of value.

len(value)

Return the length of value. value must be of type string, tuple or a slice object of user defined type.

append(t,v)

Appends value v to the tuple (or slice object) t, provided that v's type matches to t's content type. Returns a new tuple (or slice) of len equal to len(t) + 1.

concat(t1,t2)

Concats the two tuples or slices t1 and t2, provided that t1's content type t2's

content type match. Returns a tuple (or slice object) of len() equal to len(t1) + len(t2).

panic(err)

Throws an exception. The err parameter is an optional value that can be used to identify the exception nature in a catch block.

assert(val, message)

Tests the given bool value val. If not true, throws an exception with the given message.

4.1.7 Command Line

Although the radically different in semantics, command line execution of programs and redirections in El look the same as other UNIX shells. Pipes use the usual '|' operator, redirections '>' and '<', and '&' causes background execution of a pipeline. Thus the following pipeline execution

```
ls . | count > ./ints
```

behaves as expected, provided the translation to Ethos semantics: the pipe between ls and count accepts objects of type 'any' (the type accepted by count's stdin); the redirection to ./ints has a directory as its target, and it must be typed as int (type produced by count on stdout); the whole execution is carried out in background, i.e. El continues execution (and presents a new prompt if in interactive mode) without waiting for count to exit.

Multiple pipeline operations may be queued on the same line using on of the ‘;’, ‘&&’ and ‘||’ connectors. ‘’ continues execution of the line only in case of no exception thrown (i.e.: if the previous operation completed without failing); ‘||’ continues execution only if the previous operation failed. For example, the following redirection operation is executed only if the ‘./strings’ directory exists, since otherwise ls raises an error condition. ‘print done’ is instead executed in any event.

```
ls ./strings && echo foo > ./strings; print done
```

4.2 Implementation

In Table VI are listed the main sub-packages composing the El codebase, along with an approximative size in Lines of Code (LOC) and an overview of the accomplished tasks. The LOC measure refers to Go lines of code except for package el/parser which is mainly composed of goyacc code.

4.2.1 Parsing

The initial design of the parser didn’t made it to the current version. The parser was written from scratch as a parser combinator, using a small library written in Go and heavily inspired by (while much simpler than) *Parsec*[28]. Although fully functional, the obtained parser was too slow to be used in real world without an heavy performance tuning.

Luckily, among the great variety of tools available for Go there is *goyacc*[29], with which the current parser implementation is generated. *goyacc* is a version of *yacc*[30]

TABLE VI: EL CODEBASE ORGANIZATION

Package	**LOC**	**Description**
el	250	main access point, handles command line arguments and execution modes for El (Read-Eval-Print Loop (REPL)/interactive vs. script)
el/types	300	primitive types definitions, composite types representation, internal type checking facilities
el/environment	300	execution state, symbol table, control flow flags
el/eval	1800	per-nodetype evaluation methods, builtin functions
el/io	1000	typed access to filesystem objects, access to Ethos' *typeGraph*
el/operators	900	operators for primitive types
el/parser	1200	goyacc-generated parser, AST structure and walk interface definition

written in Go, generating parsers written in Go. The output of the parsing phase is an AST that is directly used in the evaluation phase.

As described in Section 3.4, the major requirements for what concerns the syntax of El is to be able to support both CLI and scripting interaction modes naturally. Most of the perceived terseness of a sh one-liner boils down to two main factors:

1. the ability to pipe commands, perform redirection, conditionally execute the next operation based on the status of the previously executed one, and to execute pipelines in background, are all accessible with minimal syntax; and
2. the fact that every token is by default interpreted as a string, except for keywords and special tokens; this dramatically reduces the amount of non-alphabetic char-

> acters that one has to type, since the only way to perform I/O between commands and/or filesystem is exactly through (untyped) streams of characters, that is: there is no need (nor there are means) to provide command line arguments or values that are not strings.

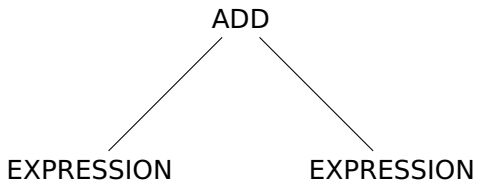
As of (1), El simply borrows the same syntax. For what concerns (2), El adopts different expedients. *Bare words* are parsed as strings. Another form of string literal is specified instead between double quotes, and can thus accommodate strings otherwise not expressible, e.g: starting with '$' or containing spaces. El also differentiates between two parsing contexts: command line arguments and "everything else". In the first case, tokens are always parsed as strings, except in the case of variable substitution, since this is what command arguments are going to be in the end—strings. We discuss how we plan to augment command line arguments semantics in Chapter 6.

The complete grammar for El (a stripped out and formatted version of the input file for *goyacc*) is reported in Appendix A.

4.2.2 Evaluation

Evaluation is performed directly on the AST, result of the parsing phase, and is structured as an *attribute grammar* where attributes are passed decorating the tree nodes.

We need both synthesized and inherited attributes. Two disjoint sets of evaluators are defined: *downEvaluators* are executed on the nodes during the descending phase




```
1  Add(node, env)
2      left ← node.Children[0]
3      right ← node.Children[1]
4      node.Value, node.Type ← operators.Sum(left, right)
5      return false
```

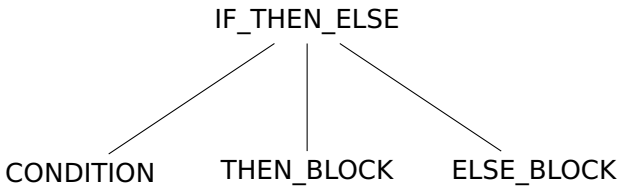
Figure 7: Example evaluator ‘add’ and relative AST portion

(i.e.: they mostly produce inherited attributes), *upEvaluators* during the ascending phase (i.e.: synthesized).

Evaluators have the ability to control the traversing of the tree, i.e: varying with their behavior, *downEvaluators* can either stop or not the recursive evaluation of a subtree; *upEvaluators* instead are executed on the ascending phase, and thus the relative sub-tree rooted at the current AST node will always be already evaluated.

Each evaluator has access to the AST node it is operating on (and thus its subtrees, if any) and to the current environment (including scope).

An example of *upEvaluator* is Add, which we reproduce in simplified form, along with the relative subtree it operates on, in Section 4.2.2. Add simply applies the Sum operator to its two children, and propagates the resulting value and type to the parent




```
1  IfThenElse(node, env)
2      condition ← node.Children[0]
3      thenBlock ← node.Children[1]

5      Eval(condition, env)

7      if isTruthy(condition.Value)
8          Eval(thenBlock, env)
9      else if length(node.Children) = 3
10         Eval(node.Children[2], env)

12     return true
```

Figure 8: Example evaluator ‘ifThenElse’ and relative AST portion

node. An example of *downEvaluator* is reported in Figure 4.2.2. IfThenElse stops the recursive evaluation of its children trees (by returning true) and takes care of explicitly evaluating only one of the two branches invoking Eval on such one (lines 7-10), after having established the truth value of the condition (line 5).

The first way different evaluators exchange information and state is thus directly through AST nodes’ values. The second way is interacting with the current environment—mostly assimilable with the current execution scope. Evaluators have the ability to set control flow flags for the scope, and query and/or clear them later. Examples of such

flags are: *break*, *continue* (used for control flow in loops) or *panic* (used for exception handling).

Summing up, evaluators define the core behavior of the interpreter, implementing control flow and expression propagation. Expression values and, in general, state of the execution, are propagated as attributes in form of values decorating the original AST or as special flags applied to the current scope.

As a final remark, we highlight the fact that, given the separation in code and tasks between the parsing and evaluation components of the codebase, additional steps before evaluation (think of some form of static checking or optimizations) can be added using the same AST-walking interface, before the actual evaluation step.

4.2.3 Environments

The Environment represents the state during evaluation, that is: the SymbolTable, the current working directory, and a few additional flags that the evaluators use for control flow (return, break, continue, panic, error handler). In addition, each environment has a meaningful name (typically the name of the function causing the new scope creation), that is used to present stack traces in case of unhandled errors.

An Environment can be copied, creating a new scope, that has access to the original scope but cannot pollute it with symbols defined in the inner scope.

The SymbolTable is organized as a simple stack, and correctly implements lexical scoping for El.

The generating scope chain is not copied when creating a new scope, instead it is just linked. This means an inner scope has a view of the modifications in the outer scope.

New environments are created in one of the following situations:

'require' of a package

A new top level environment is created for the package in order to evaluate it.

function definition

The generating environment is copied and attached to the function signature (in this way defining a *closure*). Every following function call will use this environment as its execution scope.

loop

Loops create new block scopes, mainly needed for the index / iterators definitions.

try

A try/catch construct creates a new environment and defines an error handler for the scope.

4.2.4 Execution Modes

El supports both interactive (REPL) and non-interactive execution modes. The Ethos shell is an El shell executed in interactive mode. A non-interactive execution

can be obtained by providing an El script as argument to the El executable—executing in a subshell—or sourcing an El file with ‘load file’.

In the next section we describe how generic system types are handled by El.

4.2.5 Typed Objects Manipulation

As previously described, El internal types system is composed of primitive types and the three composite types, set, tuple, map. Other than this default types, we want to be able to interact with objects of all system-known types, regardless of whether they are ETN primitive types, ETN composite types or user-defined types.

The typical usage we want to enable comprises: instantiation of new runtime objects (i.e.: not backed by physical filesystem entities); read of an object from a specific filesystem location, and write back to arbitrary location; read multiple objects of type ‘T’ from a directory into an El map (of type ‘[string]T’); access, modify, apply operators on these objects regardless of their origin.

There are two main possible strategies to achieve this result, that depends on the language with which the interpreter itself is implemented.

(a) If the implementation language has a runtime and with the ability to load code modules in an online fashion, one could follow a generate-and-use on demand approach, taking advantage of the tools already available on Ethos to generate encoders/decoders from an ETN type description. Every time a new—“new” to El—type hash is encountered reading a filesystem object, an encoder/decoder for

that type would be generated and dynamically loaded (or simply loaded if already created in the past), ready to be used to create/access objects of the given type (probably with the aid of some reflection features, that the same runtime has to support).

(b) Alternatively, it would be possible to implement a generic encoder/decoder for El, and provide access to objects of arbitrary types making use of Go's reflection features. This new encoder/decoder could be built partially on top of the existing Go packages, but would require dynamic type checking instead of pre-compiled interfaces to access objects of a specific type.

Solution (a) would be limited performance-wise. Moreover, it wouldn't be feasible with pure Go, that by design produces a single statically linked executable and doesn't provide runtime library loading facilities. Solution (b) would lead to at least partial code duplication (given that we want to access a generic type dynamically we can't just make use of Go-generated decoders). Except for this issue, this is the best way to go.

The current implementation lies somewhere in between, in that it makes use of generated encoders/decoders, but the libraries are linked at compile time. This is obviously a temporary solution, presenting a few issues: it is not really dynamic—to make use of a newly introduced type, El has to be re-compiled—and also requires linking many (possibly unused) Go packages, one for each type definition.

```
1 serviceFd ← OpenDirectory(serviceDir)
2 listenFd ← Advertise(serviceFd, name)
3 readFd ← Import(listenFd)
4 writeFd ← Ipc(serviceFd, name)
```

Figure 9: El pipe setup

4.2.6 Typed Pipelines

Here we detail the implementation of El's typed pipeline and redirections. Pipeline redirections in El are towards directories (as opposed to files in UNIX). Directories are Ethos' streaming entities, as detailed in Section 2.3.7.

The semantic should be straightforward: the pipe (or redirection) can be set up only if the stdout type of the producer and the stdin type of the consumer match. In the case of redirection to a directory, the "consumer" type is directory's type itself; in case of a pipe to another program, the consumer type is the type accepted on stdin by the consumer program. Similarly for the producer type (either the type of the directory streamed to stdin or the type of the producer).

In brief, El created pipes are Ethos services, and as such inherit all their properties of streaming typed channels. In addition, El can type check the pipe operation in advance, in order to provide meaningful error messages and avoid the waste of time and resources that building the pipe would be.

Producer and consumer programs exchange objects through Ethos' IPC. The main difference with an usual IPC is that the channel is set up by the shell instead of being set up by the two processes exchanging objects. El acts in this phase both as server and client component for the IPC, thus performing, in the order, the syscalls reported in Section 4.2.6. Lines 2-3 would be part of the server code; line 4 would be part of the client code; line 1 is normally part of both.

Note that the code in Section 4.2.6 is a simplification: particular care must be taken since the Import and Ipc are blocking, and thus there is no relative ordering of the two calls that would lead to a a non-stuck execution. In order to overcome the blocking behavior, we make use of the Ethos *Events system*. Briefly, each syscall has a non-blocking version[1] , returning an eventId. Thus it is possible to issue multiple syscalls, allowing the process to have multiple outstanding operations, and then obtain/handle the actual results in an asynchronous way.

In Section 4.2.6.1 we present a more complete version of the pseudocode. When we BlockAndRetire on evtImport and evtIpc (lines 10-11), the events can actually be satisfied, since we already issued both the Import and Ipc syscalls in their non-blocking fashion (lines 7-8).

---

[1] The blocking version of each syscall is actually a combination of a non-blocking call and a BlockAndRetire wait on the returned event identifier.

Once the IPC channel is established, each command in a pipeline is executed taking care of providing the correct file descriptors for stdin and stdout, as per the usual UNIX pipeline semantics.

4.2.6.1 Typing stdio

In order to set up the pipe service, we need to determine its the type. The service type hash is determined by the type of the producer's stdout and the consumer's stdin, when they match. If they don't, the pipe operation fails. Programs declare their stdio types by applying the desired types to predictably named directories during installation. These directories are /program/name-in and /program/name-out respectively, where name is the executable name.

Programs can always be generic in declaring stdio types of any (or union), and handling explicitly the actual type at runtime. Programs can also declare no type for stdin (or stdout), in which case they will not accept typed input (or produce typed output). Making stdio typing optional also makes the shell backward compatible with programs written prior to stdio typing introduction.

4.2.6.2 Named Pipes

Similarly to UNIX named pipes, Ethos' services are backed by a filesystem entity (serviceDir in the examples). El is responsible for management of such directories when executing commands in pipelines.

More specifically, El takes care of creating each pipe directory and clean it up once the reading process exits. Thus, unlike UNIX named pipes, Ethos pipes are not

```
1  serviceDir ← randomName()
2  CreateDirectory(serviceDir, hash)

4  serviceFd ← OpenDirectory(serviceDir)
5  listenFd ← Advertise(serviceFd, name)

7  evtImport ← async_Import(listenFd)
8  evtIpc ← async_Ipc(serviceFd, name)

10 readFd ← BlockAndRetire(evtImport)
11 writeFd ← BlockAndRetire(evtIpc)
```

Figure 10: El non-blocking pipe setup

persistent; at the same time, unlike UNIX traditional pipes, Ethos pipes are backed by a filesystem object.

Directories names are generated randomly, as to prevent a third party from being able to guess them. The CreateDirectory Ethos syscall requires a type hash to be applied to the newly created directory. This will determine the type of the service, and thus the type of objects allowed to flow through the pipe.

4.2.6.3 Redirections

For the case of redirections, the process is simpler, since there are no services to set up. The streamed objects are persisted in the destination directory (or streamed from the source directory in case of input redirection). Again, redirection is possible only if the types involved match.

CHAPTER 5

EVALUATION

In the this chapter we present the results obtained with El's development, showing how the design choices reflect in terms of goals **G1-3** as described in Chapter 1.

The evaluation is organized in two parts: in Section 5.1 we discuss El integration with Ethos and highlight obtained results with respect to goals **G1** and **G2**; in Section 5.2 we present the improvements introduced by El in terms of PL features for general user space programming, and we compare with several different shells.

5.1 El and Ethos

As per **G1** and **G2** definitions in Chapter 1, El both provides a secure shell interaction and composition (**G2**), enabled by Ethos interfaces, and preserves Ethos universal properties (**G1**), building on the OS semantics.

Table 5.1 highlights the major El's features that reduce exposure to attackers. The classes of vulnerabilities we are considering are:

parser vulnerabilities

Parsing code is complex. Applications shouldn't re-implement parsers for every data type, even if simple, to avoid increasing the code base size—and increasing it introducing some of the most error-prone code. This holds particularly for shell scripting, that in usual OS/shell setups (e.g.: UNIX/bash) often relies on regular

expressions for parsing rows, columns in a file, or similarly from a command output. Parsing vulnerabilities are common at various layers[31,32].

Applications on Ethos don't have to write their own parsers, they make use of parsers generated by Ethos type infrastructure. Same goes for the shell: El's typed object creation and filesystem access enables type-checked access to structured object types, and removes the need for custom parsing of text streams.

parser mismatches

Different implementations, even if in principle following a single specification, can lead to possible mismatches in accepted input one to each other, and thus possible issues for systems involving different components exchanging data parsed by different implementations. Similarly, an antivirus could not scan a certain file because it doesn't match any known type to the software, while another application actually making use of the file could accept it as valid, even if not scanned and thus possibly infected.

On Ethos, the parser implementation is one for each data type, hence there's no possibility of mismatches. El itself makes use of the same unique encoder/decoder for each data type when accessing a file system object or creating a new one. Different scripts, as well as a script and an application, will thus always have the same "view" for a given object.

injection attacks

Various type of injection vulnerabilities are still the preferred attack surface[33]. In an injection attack, a malicious user manipulates a free-form input in order to arbitrarily modify the behavior of the software that later manipulates that input.

Ethos applications are less prone by design to injections, given the use of structured types through encoders/decoders. What in usual architectures is typically represented as a single string, on Ethos is split in non-reducible input components and encoded as a specific composite type.

Moreover, El avoids by design command substitution and eval, that are often used to parse free form, possibly dangerous, input in other setups[34].

Having described the main vulnerabilities we are concerned with, we now list how these issues are tackled by El design. The list is split in terms of goals **G1** and **G2**, i.e. how El *preserves* Ethos properties and abstractions and how it enhances structured interaction with the OS environment compared to other shells and environments. El complies with Ethos semantics (**G1**) in:

providing coherent pipeline operations

Pipes and redirections, as described in Section 4.2.6, are streams composed of typed objects. This adheres to Ethos typing semantics. There is no way for programs to exchange data that doesn't conform to a specific type definition (as

TABLE VII: SHELL+OS FEATURES COMPARISON

	Pipes	**Command Substitution**	**fs Objects Access**
tcsh	bytes	yes	bytes/text
ksh	bytes	yes	bytes/text
scsh	text	no	bytes/text
bash	bytes	yes	bytes/text
rc/Inferno	text	yes	bytes/text
PS	objects	yes	Get-Item
El	objects	avoided*	path literal path constructor

* Command substitution is avoided by design: programs should exchange information in form of typed objects, not parsing textual output(**G1**).

per Ethos' type checker properties), and El doesn't provide any way of piping streams of characters—for instance El doesn't redirect output for programs that write text to stdout other than to the shell terminal.

imposing structured/typed access to data

As per Ethos' design, El imposes structured representation for tools and in general for all interactions. For instance, El has no "command substitution" feature; output from a command execution should instead be collected in a well-formed object(s) representation.

no untrapped errors

Ethos primary user space language is Go, a strongly typed language. El is intended to be used for many user space tasks and simple applications, and is designed as a dynamically typed language—in contrast to the untyped experience offered by commons shells. Hence, El doesn't introduce untrapped errors in Ethos user space development.

This is reinforced by the fact that failing operations by default causes El to raise exceptions (this applies both to El's runtime and to external commands failing with a non-ok status), and execution to stop in case the exception is not explicitly handled. sh will instead continue execution allowing "external" untrapped errors (failures in external programs invocations).

exporting simple and high level interfaces

El complies with Ethos design for simplicity in providing the user space programmer with high level data structures, error handling with exceptions and simplified filesystem access.

El exploits Ethos properties and abstractions (**G2**) by:

enabling typed composition of programs in shell scripts

UNIX/sh compose programs with pipes and redirections of text streams. Ethos/El equivalent are streams of typed objects. Accessing an object is transparent to application developers, in that parsers are provided by the development tool set,

and type checking is incorporated in the kernel. El provides typed pipes and redirections relying completely on Ethos abstractions of types and IPC.

enabling typed access to filesystem in shell scripts

El gives clean interfaces to files system access. Thanks to Ethos types and El filesystem integration, a whole class of error-prone parsing constructs often found in UNIX scripts are not needed altogether. sed, awk constructions to access specific values in files are replaced by path literals and type checked field access. Filesystem object can be accessed, created, modified using typed object constructors and path assignments.

reporting pipeline type checking errors in advance

Programs declare their composition interface (types accepted and produced on stdio), thus El can typecheck pipe operations and redirections in advance—prior than instantiating the pipe and involving the kernel type checker.

5.1.1 Code Readability

Code readability is a often omitted aspect in code quality analysis, and has particular implications during the code maintenance phase and bug fixing, which constitute the largest part of the software development life-cycle[35]. Even considering the usual small size of shell scripting projects, code readability is one of the key goals of El's syntax design; at the same time, given the need to support interactive usage, readability is in sharp contrast with brevity.

```
1    ls /some/dir
```

(a) Executed script foo.sh

```
1    export
         PATH=/path/to/my/ls:$PATH
2    /path/to/foo
```

(b) Executing script

Figure 11: sh environment attack

With El, we diminish the readability issues that various flavors of sh present by simplifying quoting rules (e.g.: no need to represent complex structures as strings, when more handy composite types are available) and being minimal and consistent with syntax (clear distinction between control flow and external programs, control flow constructs are consistent in syntax one to each other).

5.1.2 Packages

Here we compare El's and sh code insertion features. In particular, we evaluate the require feature highlighting the class of attack vectors it mitigates.

5.1.2.1 Environment Attacks

In general, by manipulating the environment of a script—i.e.: the environment in which it is executed—it is possible to change the script behavior. Consider an sh script that doesn't specify an absolute or relative path to execute a command (as in ls instead of /bin/ls). The behavior of this script can be arbitrarily modified when it is executed with a crafted environment $PATH variable, as shown in Figure 11-b. The executed script (Figure 11-a) might then be executing an arbitrary command instead of /bin/ls.

TABLE VIII: EL AND SH CODE EXECUTION FEATURES

	sh	**el**
subshell	"sh filename" (or "./filename.sh") executes the script in a subshell (different environment). Information flow is easy from caller to called script (command line arguments, exported variables). Information flow from called to caller is harder (except for return status).	"el filename" has the same semantics. Forks a new El process. Information flow is hard in both directions, except for return status value.
source	"source filename" or ". filename" execute **filename** in current shell context (as if the file's lines were typed one by one at the terminal).	El's "load filename" has the same semantics. Information can be exchanged in current environment/scope.
require	*not available*	included script explicitly exports functions and/or values, with fine-grained control.

The environment modification can also occur in the opposite direction, that is: an included script could modify variables used by the including one, based on the way the code insertion is done.

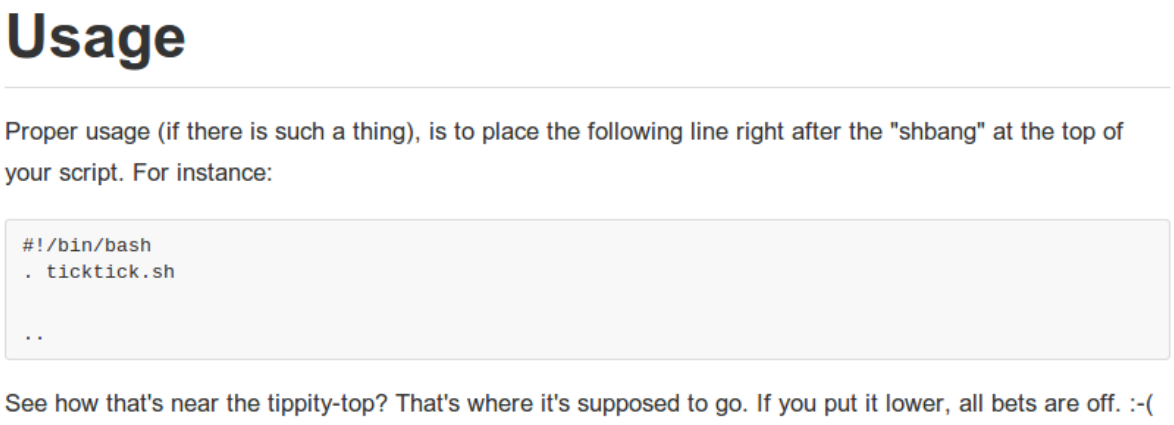
Environment attacks are a wide attack surface and affect many different layers. Consider for example the classic exploits involving $LD_LIBRARY_PATH modification (and a subsequent use of it)[36], or the PHP register_globals setting[37].

5.1.2.2 Code Insertion

In sh there are mainly two ways of executing code from another script file, source and subshell execution. In Table VIII we summarize the characteristics of the two, and we compare them to the El's counterparts.

"sh filename" and "./filename.sh"execute in a subshell. The two are slightly different ("./" can run non-sh scripts by looking at the first line, optionally specifying the interpreter program). Thus, they execute in a different environment, and information flow from executed script to executing is awkward, except for return status; for instance, it can be through the file system. Information flow from caller to called is easy, using "export VAR" and/or command line arguments. El's counterpart ("el filename") has the same semantics.

"source filename" and its alias ". filename" are sh builtins that read filename and execute the content in the current shell context—exactly as if the file's lines were typed one by one at the terminal. The included script has full control on the caller's environment—it's for instance possible to override variables and functions. Information flow is thus easy in both directions. The El equivalent is "load".



Usage

Proper usage (if there is such a thing), is to place the following line right after the "shbang" at the top of your script. For instance:

```
#!/bin/bash
. ticktick.sh

..
```

See how that's near the tippity-top? That's where it's supposed to go. If you put it lower, all bets are off. :-(

Figure 12: Typical sh usage warnings

For lack of better code inclusion mechanisms, sh's user libraries are usually imported with a source command. We analyzed the most popular bash libraries[1] , and found that *all* of them offer this single inclusion mean: source the entire library in your own script/shell. Developers releasing sh libraries are well aware of the limitations and possible security implications (see Figure 12).

User-library interaction can't be guaranteed free of namespace collisions, unless the library resorts to weird naming rules to make conflicts sufficiently unlikely.

---

[1]We obtained the most popular libraries from two sources: (a) advanced repository search on Github, specifying "bash" as the language, and ordering based on #stars,#forks. We explicitly excluded all results that are not strictly a set of library functions (e.g.: git extensions, build tools, . . .); and (b) manual Google search, again excluding non strictly-library results. The obtained sample counts 10 bash libraries.

Note that the global namespace and the possibility to override the including environment, other than constitute a possible attack surface, can be dangerous (and require time-consuming bug fixes) even if not exploited on purpose.

A single library among the analyzed ones [1] offers a possible mitigation, providing an ad-hoc command arguments interface. It is possible to invoke a single function with

```
sysfunc <command> [args]
```

forking and executing the specified function (“<command>”) in a new environment. This method of inclusion is better in terms of controlled information flow and avoided global variables clash, but requires a full subshell execution for every function invocation.

5.1.2.3 Require

require is El’s alternative for libraries and code reuse. It offers a simple way to provide library functionalities in El. A value is accessible only if explicitly exported. Information flow is thus controlled using language constructs by both the library programmer—by choosing what to export—and the library user—accessing functionalities and values as they’re needed.

A required script is evaluated only once, at first access. Thus, a reference to required package is used to access an *instance* of the package, that is: the package

---

[1] https://github.com/flaupretre/sysfunc

maintains its state in a closed environment, and requiring the package multiple times results in multiple instances, each one associated with its state/environment.

The require feature hence solves two main issues: the need for a clean way of structuring code in El applications, and the security-related issues discussed in Section 5.1.2.1.

5.2 Language generality

In Table 5.2 we summarize El features additions as a shell PL, comparing with several other shells. Other aspects not considered are mainly related to user interaction, including: history, command line editing, filename expansion, globbing. These are the main features not taken into account in the development of El and El's REPL due to the poor research challenges they pose.

5.2.1 Functions

El's functions are more versatile than bash ones in many ways: they are values and thus can be passed as arguments or stored in composite data types members; they are closures and thus able to encapsulate state; they can be defined inline as anonymous. In addition, both bash and Inferno sh functions are defined globally (this being a result of missing scoping rules), implying possibly many issues with name overriding.

PowerShell functions are instead more powerful artifacts, although they are missing the closure behavior by default (it is indeed possible to obtain a closure explicitly, invoking the 'GetNewClosure' method on a block). In PowerShell, a function can be

TABLE IX: PL FEATURES COMPARISON

	Functions	**Exceptions**	**Packages**	**Typing**	**Composite Types**
tcsh	no	no	no	no	string array
ksh	yes	‘trap’	no	no	string array
scsh	closure, first-class	yes	from scheme	dynamic	from scheme
bash	yes	‘trap’	no	‘declare’	(associative) array
rc/Inferno	yes + scoped blocks	yes	loadable modules	no	string array
PS	first-class, variadic	yes	yes	dynamic	.NET
El	closure*, first-class*	yes*	yes†*	dynamic*	tuple, map, set*

* General PL features that contribute to **G3** achievement: first-class, scope-creating functions, exception handling (compared to the problematic ‘trap’ discussed in Section 3.8), typed variables and composite types

† As discussed in Section 5.1.2, El’s ‘require’ also mitigates possibility of environment attacks (**G2**).

part of a pipeline statement and moreover is able control how the pipeline should handle the function call itself, whether as a single invocation on the whole list of pipeline objects or as a multiple 'filtering' invocation on each object flowing. Integration of El functions in pipeline statements is currently being designed, as discussed in Chapter 6.

5.2.2 Exceptions

We discussed bash error handling mechanism in Section 3.8, and highlighted its limitations.PowerShell distinguish between two main classes of errors: *terminating* and *non-terminating* ones. It is possible to configure PS to stop on non-terminating errors too, but normally only terminating errors cause PS to raise the exception (and terminate if it is left unhandled). Errors are handled with 'try/catch' statements, as usual in many PLs.

El errors are always *terminating*, i.e unhandled exceptions cause execution to stop. The error handling mechanism is a simplified 'try/catch' construct where arbitrary values can passed to a panic invocation in order to represent the error. Even if limited in features with respect to other languages' exception mechanisms (e.g. Java and other Object Oriented (OO) PLs benefit from exception classes and subclassing), El's error handling significantly improves bash alternatives and can enhance overall correctness and resiliency to corner-case conditions for a script.

5.2.3 Packages

We already discussed in Section 5.1.2 El's export/require functionality in terms of error avoidance. The same functionality is also relevant to improve El's usability as a general purpose scripting language, and thus it contributes to goal **G3** as well.

bash comes with no support for export/require. bash programmers are forced to source entire script libraries, accepting the security implications, or to make use of external executables. Inferno sh provides the *loadable modules* abstractions, and makes large use of it also for basic functionality like control flow. The main drawback of Inferno sh's loadable modules is that a module unique "export" ability is to define new *builtins*. Builtins are again globally defined and identified by their name only, thus not solving name collisions and separation. El's packages instead are encapsulated by a value returned by the require builtin, enabling, other than name collisions avoidance, multiple "instances" of the same package to be referenced, each one with its one specific environment.

PowerShell provides full-fledged package semantics. PS script modules are similar to El's packages in functionalities, except for the fact that a single import for every session has effect on the same package (there is no way to obtain references to two "instances" of the same module). In addition, PS provides binary modules, i.e. compiled modules with limited capabilities (e.g.: they can provide commands but not functions) but considerably improved performances.

The ability to organize code in isolated packages and reusable libraries is crucial in supporting structured scripting and utilities development. As an example of libraries available for El, we have been developing and using the “tuples” and “math” packages. The tuples packages provides useful (higher-order and not) functions for manipulation of tuples, like ‘sort’, ‘map’, ‘filter’, ‘fold’, ‘contains’.

5.2.4 Typing

Types play a fundamental role in the whole Ethos environment. El’s integration with Ethos types is thus fundamental in order to preserve Ethos semantics in user space scripting. Other than access to typed operation for IPC and filesystem, El internal types system guarantees an entire new level of abstraction and reliability for shell programming. Compared to bash, where all operations are on practically untyped operands (strings), types enable the shell to detect typing errors and ‘trap’ them, instead of carry on execution and allowing the *untrapped* error to possibly cause unexpected behavior for the program subsequent computations[15].

PowerShell type system is built upon .NET, thus enabling dynamic typing for primitive and composite .NET object types. Support for OO is somewhat limited, although it is possible for instance to declare a new type or to instantiate an object of an arbitrary .NET type from a PS script. In contrast, El has no support for new types definition, in line with Ethos semantics of compile-time definition of types a program makes use of.

5.2.5 Composite Types

Shell programming has suffered for long time of lack of structured ways to represent and handle structured data types. bash scripts make commonly use of all sort of quotation tricks to threat specific space-separated input either as a unique string or as an array of strings. This is prone to error, especially due to the many quotation rules and corner-cases.

El's composite types offer instead a clean interface to represent structured state and data. El maps are natural containers for references to an Ethos directory (string valued file names mapped to files of the same type 'T'); El tuples can be used to collect the output of a program execution (a stream of same-type objects), and possibly later iterate on the results. For instance, a common source of bugs in bash scripts is trying to iterate on the output of an 'ls' execution to obtain filenames and related information[1]. The primary issue is that a bash for loop iterates on the 'ls' output using Internal Field Separator (IFS) characters as delimiters; filenames are instead allowed to contain pretty much any characters, including spaces and newlines, easily breaking this kind of loops. Using El, the output of 'ls'—a stream of objects—can easily be collected in a tuple and/or filtered.

---

[1] As suggested by http://mywiki.wooledge.org/BashPitfalls, the correct way to iterate on directories content is instead making use of globbing.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this thesis we presented El, and shown how it fits in the context of the Ethos OS, preserving its security-first design as well as enhancing scripting for the platform. We also pointed out the areas where current shells come short, and how it is possible to improve the overall usability of a shell language as a generic user-space scripting language.

We highlighted some of the classes of attack El can remove or help to mitigate. In the long term, we will need to evaluate the susceptibility to errors of El programs, especially those related to security. The exception mechanism, type use, and enhanced readability are aimed at reducing these errors. Despite the early development stage, and considering the successful tools we've written using the language, El can already be considered a good foundation for Ethos shell and scripting. However, only time and widespread usage in Ethos user space will tell.

In the following we list the major limitations of the current implementation, together with directions of future improvement for the project.

6.1 Grammar Refactoring

The current grammar for El is the result of endless iterations, meant to rapidly testing new features or introducing new constructs for which the grammar was not

designed from the beginning. As such, it lost many of its nice properties and modularity along the way. A grammar redesign is in order, and also feasible now that the core syntax is well defined.

6.2 Typed Command-line Arguments

The process of transition to fully typed interfaces for Ethos is just missing a small piece: program arguments. Switching to Ethos' types for arguments means changes to the kernel, system calls, and Go' runtime at least. In the typed arguments design, programs declare their argument types (as well as whether they're optional, their default values), similarly as it happens already with stdio types. Once this is done, El will be able to exploit the type information on arguments to provide arguments type checking, completion and, eventually, access to man-pages generated from argument descriptions providing comments.

6.3 'any' operators

The way El scripts and Ethos programs have to accept or produce generic types is using the 'any' or 'union' types. An example of a type independent program is for instance 'count', producing an 'int' an accepting a stream of objects of 'any' type in input. A set of standard tools operating on 'any' type objects is in the works, including a generic 'obj-to-string' tool that comes in handy to print object gerarchies to the terminal.

APPENDIX

GOYACC EL GRAMMAR

⟨*identifier*⟩ ::= IDENTIFIER

⟨*int_literal*⟩ ::= NUMBER_LITERAL

⟨*float_literal*⟩ ::= NUMBER_LITERAL DOT NUMBER_LITERAL

⟨*string_literal*⟩ ::= STRING_LITERAL

| QUOTED_STRING_LITERAL

⟨*bool_literal*⟩ ::= BOOL_LITERAL

⟨*tuple_literal*⟩ ::= L_SQ ⟨*params_list*⟩ R_SQ

⟨*map_literal*⟩ ::= L_PAR ⟨*colon_params_list*⟩ R_PAR

⟨*path_literal*⟩ ::= PATH

⟨*set_literal*⟩ ::= L_CUR ⟨*params_list*⟩ R_CUR

⟨*literal*⟩ ::= ⟨*int_literal*⟩

| ⟨*float_literal*⟩

| ⟨*string_literal*⟩

| ⟨*bool_literal*⟩

| ⟨*tuple_literal*⟩

| ⟨*map_literal*⟩

| ⟨*path_literal*⟩

| ⟨*set_literal*⟩

APPENDIX (Continued)

⟨*assignment*⟩ ::= ⟨*identifier*⟩ EQ ⟨*expression*⟩

| ⟨*path_literal*⟩ EQ ⟨*expression*⟩

| ⟨*square_access*⟩ EQ ⟨*expression*⟩

| ⟨*path_constructor*⟩ EQ ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*bool_expression*⟩

⟨*bool_expression*⟩ ::= ⟨*or_expression*⟩

⟨*or_expression*⟩ ::= ⟨*and_expression*⟩ OR ⟨*or_expression*⟩

| ⟨*and_expression*⟩

⟨*and_expression*⟩ ::= ⟨*comparison*⟩ AND ⟨*and_expression*⟩

| ⟨*comparison*⟩

⟨*comparison*⟩ ::= ⟨*e_comparison*⟩

| ⟨*n_e_comparison*⟩

| ⟨*l_comparison*⟩

| ⟨*l_e_comparison*⟩

| ⟨*g_comparison*⟩

| ⟨*g_e_comparison*⟩

| ⟨*not*⟩

| ⟨*sum*⟩

⟨*l_comparison*⟩ ::= ⟨*sum*⟩ L_COMP ⟨*sum*⟩

⟨*l_e_comparison*⟩ ::= ⟨*sum*⟩ L_COMP EQ ⟨*sum*⟩

⟨*g_comparison*⟩ ::= ⟨*sum*⟩ G_COMP ⟨*sum*⟩

APPENDIX (Continued)

⟨*g_e_comparison*⟩ ::= ⟨*sum*⟩ G_COMP EQ ⟨*sum*⟩

⟨*e_comparison*⟩ ::= ⟨*sum*⟩ EQ EQ ⟨*sum*⟩

⟨*n_e_comparison*⟩ ::= ⟨*sum*⟩ NOT EQ ⟨*sum*⟩

⟨*not*⟩ ::= NOT ⟨*value*⟩

⟨*sum*⟩ ::= ⟨*add*⟩

| ⟨*sub*⟩

| ⟨*product*⟩

⟨*add*⟩ ::= ⟨*product*⟩ PLUS ⟨*sum*⟩

⟨*sub*⟩ ::= ⟨*product*⟩ MINUS ⟨*sum*⟩

⟨*product*⟩ ::= ⟨*mul*⟩

| ⟨*div*⟩

| ⟨*value*⟩

⟨*mul*⟩ ::= ⟨*product*⟩ MUL ⟨*value*⟩

⟨*div*⟩ ::= ⟨*product*⟩ SLASH ⟨*value*⟩

⟨*value_access*⟩ ::= ⟨*value*⟩ DOT ⟨*function_call*⟩

| ⟨*value*⟩ DOT STRING_LITERAL

⟨*square_access*⟩ ::= ⟨*value*⟩ L_SQ ⟨*sum*⟩ R_SQ

⟨*export*⟩ ::= EXPORT ⟨*function_definition*⟩

| EXPORT ⟨*var*⟩

⟨*value*⟩ ::= ⟨*value_access*⟩

| ⟨*square_access*⟩

| ⟨*function_call*⟩

APPENDIX (Continued)

| ⟨*literal*⟩

| ⟨*function_definition*⟩

| ⟨*identifier*⟩

| L_PAR ⟨*expression*⟩ R_PAR

| MINUS ⟨*value*⟩

| ⟨*constructor*⟩

| ⟨*path_constructor*⟩

⟨*statement*⟩ ::= ⟨*var*⟩

| ⟨*function_definition*⟩

| ⟨*value_access*⟩

| ⟨*control_statement*⟩

| ⟨*function_call*⟩

| ⟨*assignment*⟩

| ⟨*export*⟩

| ⟨*pipelines*⟩

⟨*var*⟩ ::= VAR ⟨*identifier*⟩ EQ ⟨*expression*⟩

| VAR ⟨*identifier*⟩

⟨*control_statement*⟩ ::= BREAK

| CONTINUE

| ⟨*return*⟩

| ⟨*loop*⟩

APPENDIX (Continued)

	|	⟨*if*⟩
	|	⟨*switch*⟩
⟨*loop*⟩	::=	⟨*foreach*⟩
	|	⟨*for*⟩
⟨*foreach*⟩	::=	FOR ⟨*identifiers_couple*⟩ IN ⟨*value*⟩ ⟨*block*⟩
⟨*identifiers_couple*⟩	::=	⟨*identifier*⟩
	|	⟨*identifier*⟩ COMMA ⟨*identifier*⟩
⟨*for*⟩	::=	FOR ⟨*for_init*⟩ SEMI ⟨*for_condition*⟩ SEMI ⟨*for_step*⟩ ⟨*block*⟩
⟨*for_init*⟩	::=	⟨*assignment_list*⟩
⟨*for_condition*⟩	::=	⟨*expression*⟩
⟨*for_step*⟩	::=	⟨*assignment_list*⟩
⟨*assignment_list*⟩	::=	ϵ
	|	⟨*assignment_list_1*⟩
⟨*assignment_list_1*⟩	::=	⟨*assignment*⟩ COMMA ⟨*assignment_list_1*⟩
	|	⟨*assignment*⟩
⟨*if*⟩	::=	⟨*if_then_else*⟩
	|	⟨*if_then*⟩
⟨*if_then*⟩	::=	IF ⟨*expression*⟩ ⟨*block*⟩
⟨*if_then_else*⟩	::=	IF ⟨*expression*⟩ ⟨*block*⟩ ELSE ⟨*block*⟩
⟨*return*⟩	::=	RETURN ⟨*expression*⟩
	|	RETURN SEMI
⟨*switch*⟩	::=	SWITCH ⟨*expression*⟩ ⟨*switch_block*⟩

APPENDIX (Continued)

⟨*block*⟩ ::= L_CUR ⟨*statement_list*⟩ R_CUR

⟨*statement_list*⟩ ::= ϵ

| ⟨*statement_list_1*⟩

⟨*statement_list_1*⟩ ::= ⟨*statement*⟩ ⟨*statement_list_1*⟩

| ⟨*statement*⟩

⟨*pipelines*⟩ ::= ⟨*pipeline*⟩ SEMI ⟨*pipelines*⟩

| ⟨*pipeline*⟩ AND ⟨*pipelines*⟩

| ⟨*pipeline*⟩ OR ⟨*pipelines*⟩

| ⟨*pipeline*⟩

⟨*pipeline*⟩ ::= ⟨*pipeline_1*⟩ AMPERSAND

| ⟨*pipeline_1*⟩

⟨*pipeline_1*⟩ ::= ⟨*pipeline_statement*⟩ PIPE ⟨*pipeline_1*⟩

| ⟨*pipeline_statement*⟩

⟨*pipeline_statement*⟩ ::= ⟨*command*⟩ ⟨*arguments_list_1*⟩ ⟨*redirections*⟩

| ⟨*command*⟩ ⟨*redirections*⟩

⟨*command*⟩ ::= STRING_LITERAL

| PATH

⟨*redirections*⟩ ::= ⟨*redirection*⟩ ⟨*redirection*⟩

| ⟨*redirection*⟩

| ϵ

⟨*redirection*⟩ ::= L_COMP PATH

| G_COMP PATH

APPENDIX (Continued)

⟨*arguments_list_1*⟩ ::= ⟨*argument*⟩ ⟨*arguments_list_1*⟩

| ⟨*argument*⟩

⟨*argument*⟩ ::= ⟨*identifier*⟩

| ⟨*literal*⟩

| DOT

| SLASH

| TOKEN

| MINUS ⟨*argument*⟩

⟨*switch_block*⟩ ::= L_CUR ⟨*case_list*⟩ R_CUR

| L_CUR ⟨*case_list*⟩ ⟨*case_else*⟩ R_CUR

⟨*case_list*⟩ ::= ϵ

| ⟨*case_list_1*⟩

⟨*case_list_1*⟩ ::= ⟨*case*⟩ ⟨*case_list_1*⟩

| ⟨*case*⟩

⟨*case*⟩ ::= CASE ⟨*expression*⟩ COLON ⟨*block*⟩

⟨*case_else*⟩ ::= ELSE COLON ⟨*block*⟩

⟨*function_call*⟩ ::= ⟨*expression*⟩ L_PAR ⟨*params_list*⟩ R_PAR

| STRING_LITERAL L_PAR ⟨*params_list*⟩ R_PAR

⟨*params_list*⟩ ::= ϵ

| ⟨*params_list_1*⟩

⟨*params_list_1*⟩ ::= ⟨*expression*⟩ COMMA ⟨*params_list_1*⟩

| ⟨*expression*⟩

APPENDIX (Continued)

⟨*colon_params_list*⟩ ::= ϵ

| ⟨*colon_params_list_1*⟩

⟨*colon_params_list_1*⟩ ::= ⟨*colon_param*⟩ COMMA ⟨*colon_params_list_1*⟩

| ⟨*colon_param*⟩

⟨*colon_param*⟩ ::= STRING_LITERAL COLON ⟨*expression*⟩

⟨*function_definition*⟩ ::= FUNC STRING_LITERAL L_PAR ⟨*named_params_list*⟩ R_PAR

⟨*block*⟩

| FUNC L_PAR ⟨*named_params_list*⟩ R_PAR ⟨*block*⟩

⟨*named_params_list*⟩ ::= ϵ

| ⟨*named_params_list_1*⟩

⟨*named_params_list_1*⟩ ::= ⟨*identifier*⟩ COMMA ⟨*named_params_list_1*⟩

| ⟨*identifier*⟩

⟨*constructor*⟩ ::= NEW STRING_LITERAL L_CUR ⟨*colon_params_list*⟩ R_CUR

| NEW STRING_LITERAL L_CUR ⟨*params_list*⟩ R_CUR

⟨*path_constructor*⟩ ::= PATH_CONSTRUCTOR COLON ⟨*expression*⟩ COLON

⟨*program*⟩ ::= ⟨*statement_list*⟩

CITED LITERATURE

1. Kernighan, B. W. and Pike, R.: The Unix programming environment. Prentice Hall Professional Technical Reference, 1983.
2. Unix 1st edition man - sh. http://man.cat-v.org/unix-1st/1/sh (date accessed 2/15/2014).
3. Mashey, J. R.: Using a command language as a high-level programming language. In ICSE, pages 169–176. Citeseer, 1976.
4. Bourne, S.: The unix shell. The Bell System Technical Journal, 57(6 Part 2):1971–1990, 1978.
5. Joy, W.: An Introduction to the C shell. University of California, Berkeley, 1980.
6. The new korn shell. http://www.linuxjournal.com/article/1273 (date accessed 2/15/2014).
7. Ellis, M., Greer, K., Placeway, P., and Zochariassen, R.: Tcsh-cshell with filename completions and command line editing. Department of Computer Science, Toronto, Canada, 1987.
8. Rc — the plan 9 shell. http://doc.cat-v.org/plan_9/4th_edition/papers/rc (date accessed 2/15/2014).
9. The inferno shell. http://www.vitanuova.com/inferno/papers/sh.html (date accessed 2/15/2014).
10. Es: A shell with higher-order functions. http://wryun.github.io/es-shell/paper.html (date accessed 2/15/2014).
11. scsh – a scheme shell. http://www.scsh.net/docu/scsh-paper/scsh-paper.html (date accessed 2/15/2014).
12. Microsoft windows powershell. http://technet.microsoft.com/en-us/library/ms714418.aspx (date accessed 2/15/2014).
13. Pike, R.: System software research is irrelevant, August 2000.
14. Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R.: An empirical study of operating system errors. pages 73–88, 2001.
15. Cardelli, L.: Type systems. ACM Computing Surveys, 28(1):263–264, 1996.
16. Golm, M., Felser, M., Wawersich, C., and Kleinöder, J.: The jx operating system. In USENIX Annual Technical Conference, General Track, pages 45–58, 2002.

17. Hallgren, T., Jones, M. P., Leslie, R., and Tolmach, A.: A principled approach to operating system construction in haskell. In ACM SIGPLAN Notices, volume 40, pages 116–128. ACM, 2005.

18. Klein, G., Derrin, P., and Elphinstone, K.: Experience report: sel4: formally verifying a high-performance microkernel. In ACM Sigplan Notices, volume 44, pages 91–96. ACM, 2009.

19. Petullo, W. M., Fei, W., Gavlin, P., and Solworth, J. A.: Ethos' distributed types.

20. Java serializable interface. `http://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html` (date accessed 2/15/2014).

21. Python pickle module. `http://docs.python.org/2/library/pickle.html` (date accessed 2/15/2014).

22. Protocol buffers. `https://code.google.com/p/protobuf/` (date accessed 2/15/2014).

23. Agarwal, A., Slee, M., and Kwiatkowski, M.: Thrift: Scalable cross-language services implementation. Technical report, Facebook, 4 2007.

24. Johnston, W. M., Hanna, J. R. P., and Millar, R. J.: Advances in dataflow programming languages. ACM Comput. Surv., 36(1):1–34, 2004.

25. The Go programming language. `http://www.golang.org` (date accessed 2/15/2014).

26. Milner, R.: A theory of type polymorphism in programming. Journal of computer and system sciences, 17(3):348–375, 1978.

27. Ierusalimschy, R., De Figueiredo, L. H., and Celes Filho, W.: Lua-an extensible extension language. Softw., Pract. Exper., 26(6):635–652, 1996.

28. Parsec. `http://www.haskell.org/haskellwiki/Parsec` (date accessed 2/15/2014).

29. goyacc. `http://golang.org/cmd/yacc/` (date accessed 2/15/2014).

30. yacc. `http://plan9.bell-labs.com/magic/man2html/1/yacc` (date accessed 2/15/2014).

31. National vulnerability database: Cve-2013-4623. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4623` (date accessed 2/15/2014).

32. National vulnerability database: Cve-2013-0156. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0156` (date accessed 2/15/2014).

33. Owasp 2013 top 10. `https://www.owasp.org/index.php/Top_10_2013-Top_10` (date accessed 2/15/2014).

34. Richards, G., Hammer, C., Burg, B., and Vitek, J.: The eval that men do. In ECOOP 2011–Object-Oriented Programming, pages 52–78. Springer, 2011.

35. Buse, R. P. and Weimer, W. R.: A metric for software readability. In Proceedings of the 2008 international symposium on Software testing and analysis, pages 121–130. ACM, 2008.

36. Capec-13: Subverting environment variable values. `http://capec.mitre.org/data/definitions/13.html` (date accessed 2/15/2014).

37. Capec-77: Manipulating user-controlled variables. `http://capec.mitre.org/data/definitions/77.html` (date accessed 2/15/2014).

38. Bash—typing variables: declare or typeset. `http://www.tldp.org/LDP/abs/html/declareref.html` (date accessed 2/15/2014).

39. Petullo, W. M., Zhang, X., Bernstein, J. A. S. D. J., and Lange, T.: Minimalt: Minimal-latency networking through better security.

40. Petullo, W. M. and Solworth, J. A.: Simple-to-use, secure-by-design networking in ethos. In Proceedings of the Sixth European Workshop on System Security, New York, NY, USA, 2013.

VITA

Giovanni Gonzaga Nebbiante

Education

B.S., Engineering of Computing Systems
Politecnico di Milano
2011

M.S., Computer Science (*current*)
University of Illinois at Chicago, Chicago, IL
2014

Working experience

Mobile Developer *Industree S.p.A.* 2008-2011
Reggio Emilia (RE) - Italy
Developed several Nokia mobile applications for two nationwide italian newspapers and other communication agencies.

Research Assistant *University of Illinois at Chicago* Jan 2013-Dec 2013
Chicago, IL