Ethos' Deeply Integrated Distributed Types

W. Michael Petullo
Department of Electrical Engineering and Computer Science
United States Military Academy
West Point, New York 10996
Email: mike@flyn.org

Wenyuan Fei
Department of Computer Science
University of Illinois at Chicago
Chicago, Illinois 60607
Email: wfei2@uic.edu

Jon A. Solworth
Department of Computer Science
University of Illinois at Chicago
Chicago, Illinois 60607
Email: solworth@rites.uic.edu

Pat Gavlin
Microsoft
Redmond, Washington 98052
Email: pgavlin@gmail.com

Abstract—**Programming languages have long incorporated type safety, increasing their level of abstraction and thus aiding programmers. Type safety eliminates whole classes of security-sensitive bugs, replacing the tedious and error-prone search for such bugs in each application with verifying the correctness of the type system. Despite their benefits, these protections often end at the process boundary, that is, type safety holds within a program but usually not to the filesystem or communication with other programs. Existing operating system approaches to bridge this gap require the use of a single programming language or common language runtime.**

We describe the deep integration of type safety in Ethos, a clean-slate operating system which requires that all program input and output satisfy a recognizer before applications are permitted to further process it. Ethos types are multilingual and runtime-agnostic, and each has an automatically generated unique type identifier. Ethos bridges the type-safety gap between programs by (1) providing a convenient mechanism for specifying the types each program may produce or consume, (2) ensuring that each type has a single, distributed-system-wide recognizer implementation, and (3) inescapably enforcing these type constraints.

I. Introduction

LangSec posits that trustworthy software which accepts untrusted inputs must define a context-free language L of acceptable inputs and must recognize any input against L before processing it [49]. This formal approach prevents (1) exposure to unsanitized inputs due to an incorrect recognizer, and (2) inconsistent recognizers. Recognizers can be small, simple, and formally verified; many tools exist that can help implement such recognizers in individual applications.

One way of providing LangSec is through a type system. In general, applications can contain *untrapped errors*—where an error goes unnoticed and computation continues—and *trapped errors*—where an error is detected and computation stops [16]. Untrapped errors are particularly pernicious since they result in arbitrary behavior. Sassaman et al. identified consequences that arise from poor input handling, including X.509 certificate ambiguity, IDS evasion, stack-smashing attacks, injection attacks, and format string attacks [49], [48]. *Type safety* prevents untrapped errors and can reduce trapped errors.

We describe here Ethos' distributed type system, *Etypes*. Ethos is an experimental clean-slate Operating System (OS) which is designed to make it easier to write robust applications—that is, applications which withstand attack. Because of the large number of attack vectors against applications in traditional software systems, it is a daunting task to create robust applications on traditional systems. Ethos eliminates many application attack vectors in the OS layer, so that application programmers do not need to deal with them. This is in keeping with Ethos' design goal of providing security over compatibility, simplifying application development and resulting in programs which are less prone to attack.

Processes on Ethos send data to other processes—either directly through Inter-Process Communication (IPC)[1] or indirectly through the filesystem—in the form of typed objects. Ethos subjects *all* I/O to recognizers which trap any ill-formed message. In general, inputs to the Ethos OS are either the result of user-space writes or arriving network packets.

The filesystem is the most important namespace for an OS, and Ethos both continues and extends the use of the filesystem namespace. In Ethos, type definitions exist in the filesystem, the filesystem names IPC channels in addition to files, and the filesystem bears a type label for each file/IPC channel. This bears some resemblance to the design of access controls in traditional OSs. OSs that subsume application-level access controls increase application programmer processing fluency because programmers no longer needed to build access controls into every program. Our experience is that programmers similarly benefit from Etypes.

The use of types throughout Ethos is extreme. Rather than using existing, low-level, and often ad hoc protocols, Ethos protocols are always generated from type descriptions. This

Permanent ID of this document: 04b3786f81a338a26d7b7935b420c019.
Date: April 1, 2014.

[1]Both local IPC and networking share one API on Ethos (§IV-C).

reduces application code, and thus chances for error. Our type descriptions are high-level and processed in a way that makes it easy to keep applications mutually intelligible. When it is necessary to use legacy protocols, we use *protocol proxies* to translate between Etypes and legacy protocols.

Etypes' contributions include:

C1 Each Ethos process' input and output channel has a declared type. Ethos guarantees that all input and output is recognized with respect to its declared type (§IV-C).

C2 Ethos tightly integrates PL/system support (§IV-B, §IV-C), which reduces application code and eliminates type errors.

C3 Etypes has had an unexpected impact the design of Ethos' scripting language.

C4 Ethos couples recognition with authorization to restrict at runtime which types a given application can produce or consume.

C5 Etypes provides a refined, naming-authority-free Universally Unique ID (UUID) generator for types (§IV-D). Types are independently defined and guaranteed not to conflict with types created elsewhere, yet equivalent types will always receive the same UUID.

Semantics implemented in the OS have significant requirements beyond those implemented in user space. An OS mechanism must be efficient, it must compose well with other OS mechanisms, it must be secure, it must be robust even if incorrectly used, and it must have a well-defined interface. Typically, an OS supports multiple languages to support a variety of uses. Despite these challenges, OSs provide complete mediation over interaction with the outside world, and so they are an ideal place to put security-sensitive abstractions.

We shall show in §V that Etypes provides stronger security services, removes many pitfalls which cause security holes, and reduces application lines-of-code (and therefore attack surface). We first survey types in general, including a series of definitions (§II). In the remainder of the paper, we discuss an overview of Etypes (§III), our design (§IV), evaluation (§V), and related work (§VI).

II. General discussion of types and OSs

Programming Languages (PLs) can be *untyped*, in which variables can hold values of any type; or they can be *typed*, in which each variable is restricted by a type [16]. Type-safe PLs have no untrapped errors. This is ensured either by an untyped language runtime performing *dynamic checking* or by *statically checking* a typed program prior to execution. (Some languages have features that require dynamic checking, even though most of the language is statically checked.) Go is a typed language, LISP is an untyped language, and assembly is a language that is both untyped and unsafe.

Traditional OSs are untyped and have untrapped errors both internally and in their interaction with applications. As examples, OSs themselves often contain buffer overflows, and the read and write system calls found in POSIX process untyped byte sequences. Due to the latter, these OSs have little opportunity to sanitize program input, which in turn allows applications to receive ill-formed data. All too often, this leads to silent and unpredictable application failure.

Before adding a type system to an OS, one has to decide the properties that such a type system should provide. There are a number of specialized type systems in use in PLs, including linear types [53], which ensure at most one reference to each object; dependent types [41], where values influence types; and union types [33], which allow performing only operations valid for two or more types. These type systems provide increased strength, yet for the reasons that follow, they presently appear to be too restrictive for inclusion universally at the system level.

It is best if an OS is multilingual. There are different ways of composing things in an OS, some of which are quick and dirty, others of which need meticulous construction, and yet others of which benefit from different programming paradigms. More exotic type systems either (1) rule out certain PLs or (2) cause an impedance mismatch when a PL is shoehorned into a system with an ill-fitting type system. We don't know what PL will be the most important for Ethos; thus we designed a simple type system for Ethos which is intended to provide a better match for popular type-safe PLs. We discuss language needs further in §IV-B.

It is a key thesis of the Ethos project that today's systems do not compose well, resulting in overall code bases which are at least an order of magnitude larger than is needed. These bloated code bases arise from OSs that are implemented as a series of low-level mechanisms to maximize the degree of implementation freedom. In contrast, although Ethos' higher-level mechanisms can provide any functionality, Ethos virtuously restricts implementation freedom. Comparing Ethos to traditional OSs is akin to comparing higher-level and lower-level PLs.

A final consideration is versioning, where types change across software revisions. In an OS, objects are longer-lived than in a PL, and thus versioning is more important. We wanted the type not to be too specific, since this would lead to more versions. It is desirable to automatically convert between versions; Ethos supports this by maintaining sufficient type information in the filesystem. But when automated conversion is not possible, we wanted to eliminate errors resulting from type-version mismatch, especially for large-scale distributed use. In §IV we discuss Etypes' UUIDs which satisfy both conversion and detection requirements.

III. Etypes

On Ethos, applications only read and write whole objects, thus preventing short or long reads/writes [55]. A system call that writes object o will either succeed if o is well-typed or return an error without application side effects. Reads are also guaranteed not to return an ill-typed object. Managing side effects in this way simplifies application development, and is a strategy shared with other systems [37].

Here we present a high-level introduction to Etypes. We later describe in §IV the particulars of how application programmers make use of Etypes, as well as the tight integra-

Type	eNotation	eCoding
Integer	b byte	little-endian signed or
	i int$X$	unsigned X-bit integers,
	u uint$X$	where X is 8, 16, 32, or 64
Boolean	b bool	unsigned 8-bit integer
Floats	f float32	little-endian IEEE-754
	f float64	
Pointer	p $*$T	enc. method ‖ value
Array	a [n]T	values
Tuple	t []T	length ‖ values
String	s string	length ‖ Unicode values
Dictionary	d [T]S	length ‖ key/value pairs
Structure	N struct {...}	field values
Union	M union {...}	uint64 union tag ‖ value
Any	a Any	type's UUID ‖ value
RPC	F(T_0,T_1,...,T_n)	uint64 func. ID ‖ args.
Annotation	['text']	n/a: contributes to UUID
	[see 'filename']	

TABLE I: Primitive, vector, composite, and RPC type eNotation and eCoding; UUIDs are encoded as arrays of bytes; lengths are encoded as uint32; T is an arbitrary type; ‖ is concatenation.

tion between Etypes and Ethos. With Etypes, programmers specify types and Remote Procedure Call (RPC) interfaces[2] using Etypes Notation (eNotation) which, like External Data Representation (XDR), is a data description language. Etypes provides three fundamental operations: *encode*, which takes a PL type and serializes it to our wire format, called Etypes enCoding (eCoding); *decode*, which takes an eCoding and sets an appropriately-typed PL variable to its value; and *recognize*, which Ethos uses to implement recognition.

A. Etypes Notation

eNotation describes types using a syntax based on the Go programming language. Table I lists the eNotation types, a syntax example for each, and their corresponding eCoding. Primitive types include integers (both signed and unsigned), booleans, and floats. Composite types include pointers, arrays, tuples, strings, dictionaries, structures, discriminated unions, any, and RPC interfaces. Unions, any types, and RPCs warrant further description.

Unions and any types: eCoding is implicitly typed, with two exceptions: unions and the any type. A union may be instantiated to one of a specified set of types and an any may be instantiated to one of the set of all types. When a program encodes an object as either a union or any type, Etypes includes an indication of the object's actual type in the encoding.

RPCs: An Etypes *RPC interface* specifies a collection of RPC functions. Etypes RPCs are built from stub and skeleton routines, similar to Open Network Computing (ONC) RPC or CORBA. Although eNotation specifies RPC functions in a traditional way, the Etypes implementations are in fact one-way—they do not have direct return values. Instead, a callee returns a value by invoking a reply RPC. This supports both asynchronous and synchronous communication, and it also enables multiple RPC return values, even in languages whose native procedures support only a single return value. One-way RPCs are a very simple, yet entirely general, mechanism.

[2]We consider an RPC interface a type for generality; RPCs are recognized, encoded, and decoded as with other types.

```
1  Certificate struct {
2     header CertificateHeader
3     ['ABA transit number in MICR form']
4     bankId       uint32
5     ['From account; bank's num. std.']
6     fromAccount  []byte
7     ['To account; bank's num. std.']
8     toAccount    []byte
9     ['Transfer amount in US dollars']
10    amount       uint64
11 }
```

Fig. 1: An eNotation structure representing a bank transfer certificate

B. Binding semantics to types

Ethos separates recognition from processing. However, there may be semantic restrictions beyond Ethos' type system that an application could violate. For example, an integer might represent time, which should increase monotonically. Or, a type might contain pointers but prohibit cycles. In Etypes, these considerations are left to applications—there is an inherent trade off between generality and safety in a system-wide type system. These higher-level semantics are called *processing invariants*.

eNotation's *annotations* informally describe the semantics associated with types. Annotations (1) contribute to a type's UUID, binding syntax and semantics together, (2) differentiate structurally identical types, and (3) express integrity requirements outside the scope of the type system (i.e., processing invariants). Application programmers, administrators, and users refer to a type's annotations to determine the meaning of an object of that type, and thus annotations minimize the *semantic-level difference* [6] among Ethos users. Etypes annotations go beyond traditional comments because they contribute to a type's UUID. We describe how these UUIDs integrate with Ethos' recognition in §IV-C.

Consider the cryptographic certificate type described in Figure 1, a digitally signed bank transfer order. This type contains a certificate header and four certificate-specific fields: a bank ID, two account numbers, and the transfer amount. The annotations, for example the one that describes the bankId field as an American Bankers Association transit number (at line 3), narrow the semantic-level difference by carefully describing the meaning of these values. More complex annotations can be included from an external file referenced by an eNotation specification. (Field names such as bankId contribute to type UUIDs too, but they are insufficient for detailed descriptions.)

IV. Design

Etypes is best understood in the context of Ethos. Ethos' design follows three principals:

(1) robust security services,
(2) higher-level abstractions, and
(3) abstractions that are designed to compose.

The first aims at providing the protections commensurate with the contemporary threat environment, while the latter two aim at reducing the pitfalls which cause security holes.

Ethos' robust security services include encryption, authentication, and authorization. Ethos encrypts all network communication for confidentiality and integrity, authenticates all services and users, and subjects all communication to authorization. We discuss Ethos network encryption and authorization elsewhere [44], [45]; here it suffices to say that these protections are complementary to Etypes.

We discussed the impact of higher-level OS abstractions and composibility in §II. The abstraction we focus on here is, of course, types. We discuss the composition of types with the filesystem, I/O system calls, and PLs. We will show that higher-level mechanisms can be implemented in the same number of lines of code as lower-level mechanisms, yet the higher-level mechanisms provide more complete semantics.

We implemented Ethos on a paravirtualized machine provided by Xen 4.1 [7]. Ethos currently provides memory paging, processes, encrypted networking, and a filesystem. We have implemented 39 system calls in Ethos; ported the Go and Python PLs to it; and built a shell, basic command-line utilities, a remote shell utility, and a networked messaging system.

The Ethos kernel (and PL runtimes) is presently written in C, and applications are written primarily in Go. We have thus far implemented Etypes support for the C and Go PLs, and we have built an Etypes scripting language called eL. Using authorization, Ethos prohibits applications in C[3]. We do this because it is far too difficult to write secure programs in C. Longer term, we plan to reimplement much of the Ethos kernel in a type-safe language, but this is an implementation detail orthogonal to our focus on Ethos abstractions here.

Targeting different PLs presents both a challenge and opportunity. In §IV-A, we describe how Etypes remains multilingual and runtime-agnostic through the use of type graphs, and we explain why multiple PLs are needed to meet the various requirements of Ethos in §IV-B. We discuss issues related to the deep integration of Etypes into Ethos in §IV-C.

Intra-program type checking requires care to ensure type identifiers remain unique [10], [31], especially in loosely-coupled distributed systems [47]. Thus Etypes identifies its types using a *type hash*—a UUID based on a cryptographic hash of a type's description. The type hash is a fully distributed mechanism: it names each type in a unique but predictable manner, yet it does not require *any* naming authorities. Type hashes also support the requirement of versioning, because changing a type will change its type hash. We describe the algorithm to compute type hashes in §IV-D. Finally, we present sample Ethos application code in §IV-E.

[3]Ethos provides mandatory and discretionary access controls, with the mandatory controls limiting the discretionary controls. Ethos authorization is based on executable and user. In the standard configuration, the C compiler cannot create executables. Even in a system which supports system development, C can only be used by system developers, and not by application developers.

Field	Description
Hash	Type hash, the UUID for the type
Name	Mnemonic for the type
Kind	Integer representing the type's kind (e.g., uint32, struct) from a fixed enumeration
Annotation	Annotation for a type, struct field, or func.
Size	Size, if the class is a fixed-length array
Elems	Tuple of type hashes representing: the type of a typedef, fields in a struct, type of elements in a vector, parameters/return values for an RPC function, RPC functions in an interface, or target of a pointer
Fields	Tuple of strings naming: the fields in a struct, parameters/return values for a function, or RPC interface functions

TABLE II: The contents of a type graph node; one node exists for every type specified

A. Type graphs

An Etypes *type graph* describes a collection of types (having themselves been specified using eNotation) in the form of a directed graph that contains type descriptions as nodes and type references as edges. Type graphs are self-contained; for all types T in graph G, any type which T references is also in G. For example, if a struct appears in a type graph, so do the types present in the struct's fields. Each type in a type graph bears a type hash rather than the (local) names used in its eNotation specification. Given the eNotation specification of a series of types, a utility named et2g generates such a PL-independent type graph along with its type hashes.

Ethos accepts only objects whose types exists in the system's type graph (§IV-C). User-space utilities such as eg2source (IV-B) also make use of the type graph. Type graphs themselves are stored as eCoding. Table II shows the contents of a type graph node.

B. PL integration

Our type system is multilingual, but it nevertheless deeply affects the PLs Ethos supports. Here we discuss its impact in terms of three different types of PLs: unsafe, statically-checked PLs such as C; type-safe, statically-checked PLs such as Go; and type-safe, dynamically-checked PLs.

For performance reasons, Etypes minimally relies on introspection. Instead, it uses eg2source to generate code targeted to a specific PL; given a type graph and output language, the tool generates code used to encode and decode types. For RPCs, eg2source creates stubs and skeletons, similar to ONC RPC or CORBA.

Unsafe PLs: In the case of C, encode and decode require compile-time type definitions, because C uses static typing. On the other hand, types can be added to a running system, so the kernel's use of recognize requires that recognize be table-driven and thus able to recognize types added after the kernel has been compiled. Currently, encode and decode are also table-driven, but unlike with recognize, this is not a requirement. (Note that there is a distinction between types encoded/decoded within the kernel during the course of its execution—these must be known to the kernel at kernel-compile time—and types encoded and decoded by applications but merely recognized by the kernel—these need not be known by the kernel at kernel-compile time.)

For C, eg2source generates tables which describe types, and a library called libetn walks these tables to encode, decode, or recognize the types. Since libetn depends only on external malloc- and free-like functions, it easily integrates into both the OS kernel and PL runtimes. Etypes simplifies kernel code by subsuming tedious manual encode, decode, and recognition routines.

Since C is not type-safe, C can have untrapped errors even with Etypes, such as a mismatch between an eNotation type and a C variable. However, C use in Ethos is limited to system software, where it can be subjected to more rigorous code inspection.

Type-safe, statically-checked PLs: Go is type-safe and statically-checked, and we designed Etypes to fit well with such languages. Each Go program contains only a fixed number of compile-time types, and likewise, the eNotation types associated with the external objects read or written by each program are declared in advance and restricted by Ethos. Such restrictions increase application security by reducing an attacker's ability to create weird input with which an application must contend.

For Go, eg2source directly generates individual encode and decode routines for each type, unlike C's libetn- and table-based implementation. Code which uses these routines to transmit data of some arbitrary type T from one process to another is shown in Figure 2. Figure 2a creates an IPC channel and sends the value t of type T on it. Figure 2b accepts the IPC channel and receives an object of type T from it. eg2source generates the procedures Ipc, WriteT, Import, and ReadT; thus the system calls necessary to implement these operations are hidden behind typed APIs for convenience (but even direct use of the system calls will be checked by Ethos' recognizer). In keeping with Ethos' goal of minimizing application complexity, Etypes' calls require no more application code than untyped I/O calls in other OSs. However, Ethos calls do more, reducing the total amount of application code.

We believe minimizing compositional code (in this case between types and filesystem operations) simplifies programming, reduces the chances for errors which can be exploited by attackers, and makes programs more readable.

Type-safe, dynamically-checked PLs: Traditional applications benefit from statically-checked languages because such languages provide higher integrity. However, utilities often

```
enc,dec := en.Ipc(hostname, serviceName)
enc.WriteT(&t)
```

(a) Create a connection and send a typed value t

```
enc,dec,user := en.Import(serviceName)
t := dec.ReadT()
```

(b) Accept a connection and receive a typed value t

Fig. 2: Go code to create/accept an IPC and read/write a value. T is an arbitrary type.

```
seen = {}

isATree(x)
   forall f in fields(x)
      if isPtr(f) then
         if x.f in seen then
            return false
         seen = seen union { x.f }
         if ! isATree(x.f) then
            return false
   return true
```

Fig. 3: isATree in eL

benefit from dynamic types. Consider traversing a filesystem recursively and displaying the contents of files; here each type may not be known at compile time, because additional types can be added to the system after compilation. For such requirements, we have built eL [42], a dynamically-checked language that integrates tightly with Etypes. It is dynamically typed, because eL's typing is in the filesystem (all variables exist in the filesystem).

Specifying types is unnecessary within an eL program, because eL uses the type labels from the filesystem in conjunction with the type graph to type its objects. Pipes and files support typed objects, and network programming, of course, uses Etypes-based protocols. Thus there is no need to write eL code for network protocols, or other boilerplate code which is orthogonal to the application semantics. As a result, programmers can use eL to quickly and easily produce distributed programs.

Like the Bourne shell, eL supports terse command lines for interactive use, but it also supports more traditional programming constructions. Composite types simplify quoting in eL when compared to the Bourne shell; this phenomena bears some resemblance to LISP, where quoted lists cleanly nest. We discuss this further in §V-B.

eL allows printing, summarizing, extracting information, and creating composites over the types in an Ethos system. It uses a combination of generic operators, introspection, and type-specific extraction to do this. For example, we have written a program isATree (Figure 3) which will walk all the pointers of a given object recursively to see if any node is reachable by multiple paths. isATree makes no stipulation about the type of objects it checks.

One of the great successes of UNIX is its text-based utilities.

Since UNIX was designed, these utilities have been diminished as types have become richer. Generic utilities, which process types dynamically, are essential to making Ethos accessible and general. Etypes enables UNIX-like utilities that manipulate richer data. This bears some resemblance to PowerShell [3], but Ethos provides deeper integration of types with its system call abstractions.

C. OS integration

Here we we describe how Ethos associates a type with every IPC channel and file and how Ethos uses these types to regulate system calls.

Recognition overview: Ethos verifies all network reads and all (file/IPC/network) writes using recognize. Ethos ensures that only the kernel may write to local files by design,[4] so recognizers need not run on user-space filesystem reads, a significant savings over application-based recognizers. Ethos traps ill-formed writes, as an aid to correctness and to provide problem diagnosis. Thus Ethos recognition behaves analogously to an authorization reference monitor, except that recognition restricts which data an application may read or write rather than merely which objects the application may read or write. We describe in §V-C how Ethos' recognizer and reference monitor interact to provide further protections.

Associating types with objects: Every object within a given Ethos filesystem directory has the same type, that is, a directory may contain only objects of a single type. As Ethos uses filesystem paths to name IPC services, directories determine the types of both files and IPC connections. While their homogeneity increases the number of directories, it also allows Ethos to enforce type safety transparently—a write need not explicitly specify a type. We say that a directory *declares* the type of its files. We provide an example Ethos program that accesses a file in §IV-E.

In Ethos, types need only be specified when creating a directory, a relatively infrequent operation compared to file operations. Applications create directories with the createDirectory system call, which accepts as parameters the parent directory file descriptor $dirFd$, a $name$, an authorization $label$, and a type hash $tHash$:

```
createDirectory(dirFd, name, label, tHash)
```

We expect directory creation to primarily be handled by administrative tools. If the directories are set up outside of a particular application, then the application and type policy are completely independent. In cases where a directory naturally contains various types, declaring the any or union type allows for heterogeneity in the style of traditional directories.

Files: In Ethos, the contents of a file can be any Etypes object, from primitives (e.g., a 32-bit integer) to complex entities made up of multiple objects (e.g., a tree). (Of course, a particular file's type must match its parent directory as described above.) Ethos provides a writeVar system call to write an object to a file *in its entirety* (an Ethos file is not a streaming object):

```
writeVar(dirFd, fileName, contents)
```

The OS traps any attempt to write an object that is ill-formed or not of the expected type.

The inverse of writeVar is readVar:

```
object = readVar(dirFd, fileName)
```

Similar to writeVar, the read object must be well-formed to be delivered to the application. To simplify further, rather than directly use the above system calls, application programmers use encode/decode interfaces which wrap the various read and write system calls (§IV-E).

Seek: Ethos does not support the seek operation within files. It may seem odd not to have file seeking; after all, video files can span many gigabytes. But this is necessary to simplify the handling of failures—an object is either of the correct type and the readVar/writeVar system call succeeds or it is not and the call fails without application side effects.

In designing Ethos, we had originally sought to provide a file seek. But its semantics became complex when considering typed objects because (1) objects have variable size and hence computing an offset to a well-formed sub-object is not straightforward; (2) the encoding is not normally visible to applications which only see decoded values; and (3) errors would not be detectable until the whole file was read, complicating error recovery. (3) is particularly troublesome, as application effects could be introduced into the system as it reads well-formed offsets, only to encounter an ill-formed offset later. At this point, error recovery becomes unnecessarily difficult.

As a consequence of reading files only in their entirety, Ethos bounds file sizes to conserve memory. While limiting file sizes might at first seem odd, large files on traditional OSs can exceed virtual memory. Thus even in traditional OSs, user-space programmers must explicitly manage objects which do not fit in (virtual) memory.

Anecdotal evidence and research indicates that large files are increasingly important [23]. Ethos does support very large aggregates—such as video files—as a directory of files. In Ethos, traversal of files is replaced by traversal of directories.

Streaming IPC and directories: Both Ethos IPC and Ethos directories are streaming entities, and Ethos enforces the type of each write to these constructs (i.e., both IPC and directories stream objects, not bytes). The write system call sends well-formed and appropriately typed data out on a streaming descriptor:

```
write(descriptor, contents)
```

Conversely, the read system call reads from a streaming descriptor. Again, the data must conform to the expected type.

```
object = read(descriptor)
```

Streaming directories present a special challenge, because their files must be named to preserve their order. The write system call, when applied to a directory, creates a file named

[4]This is due to filesystem encryption, the use of Trusted Platform Module (TPM), and memory protections.

with the current time; the read system call, when applied to a directory, reads the next file in lexical order by filename. Thus Ethos provides a mechanism to stream over a directory of files, where each file is non-streaming (i.e., read in its entirety).

Seeking within an Ethos directory takes place at the granularity of objects instead of bytes. For example, an Ethos video format might encode each frame as a file. A program could then implement video-fast-forward functionality that displays only every nth frame by skipping over $n-1$ frames (files) and then displaying the nth one. Similarly, rewind, fast rewind, and go to a frame m minutes into the video can be easily (and efficiently) implemented.

In its implementation of streaming directories, Ethos associates a current file name with each directory descriptor. In the video format described above, each file name is the frame's number in the form of a text string. Seeking on a directory consists of tracking the current frame number, adding or subtracting the difference in frames, and then converting the result to a string in order to read the file.

We provide program fragments making use of Ethos IPC and directory seeking in §IV-E.

Networking: In Ethos, both IPC channels and network connections are created by the ipc system call. (The IPC case merely results from an empty hostname parameter.) Etypes performs many networking chores at the OS-level, including encryption, cryptographic user authentication, endianess, alignment, recognition, and data value encoding, so that networking just works. No Ethos application can receive ill-typed data from the network, because first the data must satisfy Ethos' recognizer. Higher-level interfaces also allow Etypes optimizations independent of application code.

Type graph: Ethos stores the system type graph in its filesystem at /types. Both internal kernel types and types specified in the course of application development are organized as collections, and both kernel and application-specific collections are stored in /types/spec/c/, where c is a collection name. Each file in collection directory c is a graph node, and each file's name is the type's hash. The directory /types/all/ contains copies of all of the types described by the collections in aggregate. The directory /types/all/ is loaded by Ethos at boot time and reloaded for types created while the system is running.

We envision that application type hashes will be installed by Ethos' packaging system and will remain until the installing package and all of its nodes are removed. A type with hash h can be removed from /types/all/ only when it is not present in any directory /types/spec/$*$/ and it is not used as the type hash for any directory.

Protocol proxies: Ethos cannot directly support legacy protocols such as HTTP. We note that a second OS is always present on the same host as Ethos, since Ethos runs on top of a Xen Virtual Machine (VM). Thus there are two ways to interact with a legacy protocol:

(1) run the legacy protocol entirely on the second OS, or
(2) run a protocol proxy on the second OS.

A protocol proxy speaks both eCoding and a legacy protocol.

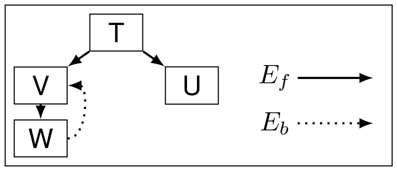

Fig. 4: A graph partitioned into E_f and E_b

A domain-specific language would be useful to deal with legacy protocols, and we have begun to build such a protocol proxy for HTTP.

Interestingly, the design of such eNotation protocols can provide security properties. This is because Etypes will limit the data that can be seen by the application and thus rule out issues such a mismatch between a length field and the actual size of data.

D. Type hash algorithm

eNotation identifies types based on the hash of their syntactic and semantic specification, ensuring each type has a UUID. The possibility of cyclic types complicates this process somewhat. For example, the eNotation in Figure 5a contains the cycle $V \to W \to V$. Here we describe the algorithm typeHash which calculates a type hash for T when given a type graph that describes T. (Of course, the type graph is not yet complete—it lacks each type's hash, which we will now compute).

Let t be the type graph node corresponding to T and let $G = (N, E)$ be a directed graph. N is the set of non-primitive eNotation definitions reachable from t, and E consists of the edges $[n, n']$ where node n directly references node n'.

First, typeHash's partition computes $G' = (N, E_f)$, a Directed Acyclic Graph (DAG) rooted at t and spanning G. (Given G, partition deterministically computes the same G'.) The remaining edges, $E_b = E - E_f$, are the back edges. Figure 4 shows the partitioning of our sample type.

We next describe how typeHash propagates hashes in E_b and E_f. Every node which references another node will contain indirectly—through a series of intermediate hashes—or directly the referenced node's hash.

First, typeHash deals with the back edges using intermediary. intermediary visits back edges $e \in E_b$, calculating the hash of the parent node of e and substituting this hash in e's child node. Included in each hash is the parent node's eNotation definition, including the annotations which precede it or are contained within it. These substitutions are done in an order such that the hash is always on a node which has not yet been re-written; this is ensured by noOut(n, E_f) which means that n has no outgoing edges in E_f. Thus intermediary computes hashes for all dependencies in E_b.

Next, typeHash uses collapse to compute type hashes for each $e \in E_f$, starting at the nodes which have no out edges and chaining back toward t. After computing this hash, collapse substitutes it for references to its type in other eNotation definitions, removes e from E_f, and repeats until E_f is empty.

Finally, typeHash computes t's hash.

```
T struct {
      aRef *U
      anotherRef *V
}

U struct { anInteger uint32 }
V struct { leadsToACyclicRef *W }

W struct { aCyclicRef *V }
```

(a) eNotation

h_5 = hash(T struct { aRef $*h_2$ anotherRef $*h_4$ })

h_2 = hash(U struct { anInteger uint32 })
h_1 = hash(V struct { leadsToACyclicRef *W })
h_4 = hash(V struct { leadsToACyclicRef $*h_3$ })
h_3 = hash(W struct { aCyclicRef $*h_1$ })

(b) Hash computation (subscript indicates order of computation)

Fig. 5: Sample eNotation structure containing the cycle $V \rightarrow W \rightarrow V$, along with its hash computation sequence

Algorithm 1 typeHash(t)

```
1: $[E_f, E_b] \leftarrow$ partition(t)
2: intermediary($E_f, E_b$)
3: collapse($E_f$)
4: return hash(t)
```

Algorithm 2 intermediary(E_f, E_b)

```
1: while $\exists[n'', n] \in E_f \mid$ noOut($n, E_f$) do
2:     for all $[n, n'] \in E_b$ do
3:         $h \leftarrow$ hash($n'$)
4:         replace references to $n'$ in $n$ with $h$
5:         $E_b \leftarrow E_b - \{[n, n']\}$
6:     end for
7:     $E_f \leftarrow E_f - \{[n'', n]\}$
8: end while
```

Algorithm 3 collapse(E_f)

```
1: while $\exists[n', n] \in E_f \mid$ noOut($n, E_f$) do
2:     $h \leftarrow$ hash($n$)
3:     replace references to $n$ in $n'$ with $h$
4:     $E_f \leftarrow E_f - \{[n', n]\}$
5: end while
```

```
1  TypeA struct {
2     W uint32
3     V Any
4  }
```

(a) Example eNotation

```
5  value := en.TypeA {uint32(0), uint64(1)}
6  dir := syscall.OpenDirectory (someDir)
7  dir.WriteVarTypeA(fileName, value)
```

(b) Example code: encode an any type to a file

```
8  dir := syscall.OpenDirectory (someDir)
9  value := dir.ReadVarTypeA(fileName)
10 switch value.V.(type) {
11    case uint64: // Of actual type uint64.
12 }
```

(c) Example code: decode an any type from a file

Fig. 6: Encoding/decoding an any type to/from a file

We now provide an example of how typeHash calculates the type hash for T. Initially, $E_f = \{[T, U], [T, V], [V, W]\}$. $[W, V]$ is a back edge and thus the only member of E_b.

The hash calculations are shown in Figure 5b. First, typeHash calls intermediary(E_f, E_b). The only edges that satisfy Line 1 are $[V, W]$ and $[T, U]$, and the only edge that satisfies Line 2 is $[W, V]$, so intermediary calculates the intermediate hash h_1 and replaces V in W's eNotation with h_1.

Next, typeHash runs collapse(E_f). This calculates the hash for U, labeled h_2 and propagates this hash to T, replacing its reference to U with h_2. Likewise, collapse hashes the definitions of W and V to compute h_3 and h_4. At each step, typeHash replaces the child's reference in the parent's eNotation with a hash. Finally, typeHash computes T's type hash, h_5.

E. Sample code

Filesystem access: Figure 6 provides an example of encoding to and decoding from a file in Go. Figure 6a defines TypeA, a struct containing an integer and an any. We assume the presence of a directory $someDir$ bearing the type hash of TypeA.

Figure 6b demonstrates how to write (encode) TypeA to this directory. Here Line 5 initializes $value$ to a TypeA structure, Line 6 opens $someDir$, and Line 7 writes $value$ to a file named $fileName$ in the directory. The OS would trap any attempt to write an object that is ill-formed relative to the type associated with $someDir$ (i.e., anything but a valid TypeA) as a runtime error.

Figure 6c provides the read. Line 9 decodes a file to a native Go struct using ReadVarTypeA. Trying to decode from a file using the wrong decode function (e.g., ReadVarTypeB) would result in a runtime error.

Streaming directories: §IV-C described an application that made use of a *video frame* type to store a video as a series of files within an Ethos directory, along with the system call semantics that make this convenient. We provide a portion of such a program in Figure 7. Each iteration of this listing's loop checks the current operation and either displays the next frame or performs a video seek operation. Seeking within streaming directories instead of within files allows Ethos to manipulate whole objects without complex failure semantics.

Any types: We now describe the details of using the any type as we depict in Figure 6's Lines 10–12. The any type in

```
current := 0
dir := syscall.OpenDirectory (someVideoDir)
for {
   op := getCurrentOp ()
   switch op {
      case PLAY:
         current++
      case BACK: // Back 30 objects.
         current −= 30
      case FORWARD: // Forward 30 objects.
         current += 30
   }
   name  := stringify(current)
   frame := dir.ReadVarFrameType(name)
   display (frame)
}
```

Fig. 7: Playing or skipping video frames; each frame is a file of type FrameType, and ReadVarFrameType reads the named file in the directory

```
I interface {
   Add(i uint32, j uint32) (r uint32)
}
```

(a) Example eNotation: an RPC interface

```
enc, dec := en.Ipc (hostname, serviceName)
enc.IAdd(0, 1)
dec.IHandle(enc)
```

(b) Example code: invoke RPC

Fig. 8: Invoking an RPC and handling the response, which will be passed to a user-defined function by IHandle

TypeA must be of some actual type specified using eNotation and present in Ethos' system type graph. (Not shown is the error handling for Lines 7 and 9 should the type be unknown.) Encoding an any type encodes the type hash of the actual type followed by the encoding of the actual type. Decoding an any type uses introspection to identify the actual type (Line 10). Once the actual type is determined, the application can act on it appropriately.

RPC: Figure 8 provides an example of invoking an RPC. Figure 8a defines an example RPC interface containing a single function, Add. Not depicted is a detailed annotation describing Add.

Figure 8b provides the body of an application. It opens a network connection using Ipc and initializes enc and dec to the returned encoder and decoder objects, respectively. These, in turn, are wrappers for Ethos' read and write system calls, and also provide access to the generated RPC stub/skeleton routines (recall that these too are abstractions of read and write). The program next invokes enc's IAdd function, thereby making an RPC request. Calling dec's IHandle function causes the program to wait for an incoming RPC reply to IAdd. The programmer must also implement iAdd and iAddReply, but this is not depicted (the generated skeleton routine IHandle will dispatch to these functions). Attempting to write an ill-formed request relative to the type associated with serviceName will cause a runtime error, and Ethos will not deliver ill-formed responses to the application.

V. Evaluation

Ad hoc input sanitization has been problematic, often due to incomplete or inconsistent recognizers. Traditional OSs do little to ensure data is recognized correctly or consistently, as each application is left to include their own recognizers. Tools such as bison, PADS [21], and Hammer [2] greatly aid in writing recognizers and parsers, but their use is discretionary and so it remains difficult to assess the safety of a large system as a whole. These issues give rise to subtle vulnerabilities.

Here we describe how Ethos addresses many of these vulnerabilities, before describing some performance results. We ran performance experiments on computers with 4.2 GHz AMD FX-4170 quad-core processors and 16 GB of memory.

A. The context-free equivalence problem

Jana et al. showed that because virus detectors and applications often parse data differently, detectors might miss infected files [34]. For example, a detector might not scan an infected file contained in an ill-crafted archive file, yet an extraction program might unwittingly extract it. Such difficulties are also encountered in defenses against Cross-Site Scripting (XSS) attacks, because a server's input validation must anticipate how a client browser will parse HTML [13]. Similar patterns result from multiple X.509 parsers [39], Flash parsers, and so on.

Ethos addresses the context-free equivalence problem by (1) explicitly typing data, (2) removing ad-hoc recognizers from applications and centralizing them at the system level and (3) generating both system-level recognizers and application-level encoders and decoders from a single eNotation description per type.

B. Injection attacks

In SQL- and OS-command injection attacks, an attacker influences an unstructured string that is later parsed and acted upon by another system component [29], [50]. For example, a naïve program might receive a file path from the network and add this path to a command line sent as input to a UNIX shell. An attacker could craft a path that contains the ';' character, which could cause the shell to execute arbitrary commands that the attacker embeds in his path string. Such injection attack vulnerabilities remain prevalent [40].

By forbidding unstructured communication. Ethos makes it unnecessary to combine data (i.e., the path input described above) with control strings (i.e., the ';' character). On Ethos, distinct types represent shell commands and database queries. An application communicates such requests in the form of an eCoding, and these encodings are equivalent to an abstract syntax tree. Avoiding the need to parse unstructured text removes the need for fragile sanitization routines and complicated character escape sequences. These Ethos facilities are more general than, but resemble, prepared SQL statements [12].

C. Semantic-gap attacks

Buccafurri et al. presented the Dalí attack in the context of digital signatures [15], and Jana et al. presented chameleon attacks in the context of anti-virus software [34]. These attacks arise because the types of the objects stored in filesystems or communicated between processes are left ambiguous. Applications must guess object types, and, of course, sometimes do so erroneously.

In a Dalí attack, a signatory is fooled into producing a certificate that can have two meanings, depending on which application views it (Buccafurri showed that one file could simultaneously be both a valid PDF and TIFF). Chameleon attacks are similar; here a file's type is inferred one way by a virus checker but another by an application, resulting in a falsely negative virus scan. In contrast, Ethos is type safe, it inescapably labels each object with its type (§IV-C), every application and utility interprets an object consistently by this type, and each type includes a universal meaning as defined by its annotations.

The depth of Ethos' type enforcement warrants further discussion. Consider two types t_1 and t_2 with identical structure; they nonetheless have different hashes in Ethos due to their annotations (§III-B). An Ethos application commits to a type when it reads or writes data as an Ethos object, and Ethos thereafter enforces this chosen type. Of course, an application could erroneously swap data of type t_1 for t_2 (i.e., read an object of type t_1 and write it as a structurally-identical object of t_2). This error is unavoidable through Ethos' structural recognition alone, but Ethos applications are written in type-safe programming languages, which assume responsibility for protecting against such mistakes within a process. A user might also inadvertently execute a maliciously written program, and that program could arbitrarily transform data but nonetheless write it as a well-formed Ethos object of the expected type; here Ethos' authorization system can restrict the application so that it can only write Ethos objects of type t_1. Ethos' certificate system provides further countermeasures against such swaps (see below). Even in these cases, there is no possibility of encoding errors since t_1 and t_2 are identically coded.

Another attack comes from short or long reads, in which the object is partially read or data beyond the object is read [55]. In these attacks, it is possible to get confused as to the sources of input, mistaking untrusted for trusted information. The interfaces provided by Ethos forbid short or long reads.

D. Certificates

A certificate is a signed statement, and the meaning of the statement depends on the type of the certificate. For example, Alice might want a certificate which requests a payment from her bank account to a utility company for \$100. Certificates provide very strong protections, as anyone can determine whose key signed it, but the signature prevents tampering since the signature depends on the certificate's content.

Ethos provides a sign system call:

```
sign(dirFd, fileName)
```

to sign certificates, and because sign is a system call, Ethos can isolate keys from applications to forbid any other means of signing [43]. For this system call to succeed, the process must have signing permission on objects in the directory $dirFd$. Moreover, the directory has only objects of a single type, so that an application—such as a less-trusted program that needs to generate signatures, but not on payment requests—cannot accidentally (or maliciously) sign a forbidden certificate type (i.e., meaning). Thus in Ethos permissions and types go hand-in-hand to strengthen protections, in addition to the semantic-gap protections described above.

System	Component	C	Go	Template	YACC
Etypes	libetn	1,278			
	et2g		826		329
	eg2source		1,407	2,320	
ONC	libtirpc	15,105			
	rpcbind	5,264			
	rpcgen	5,479			

TABLE III: Lines of code in Etypes (total 6,160) and ONC RPC (total 25,848)

Component	Purpose	LoC
MIME	Parse MIME	13,381
SMTP	Interact using SMTP	1,487
POP3	Interact using POP3	2,958
libxml2	Parse XML/HTML	136,362

TABLE IV: Lines of code in selected libcamel components

E. Reducing lines of code

We describe next how eg2source reduces the Lines of Code (LoC) in both the kernel and applications. Such a reduction almost always benefits security, as it removes opportunities for programmer error.

Kernel code: As previously discussed, we use libetn in the Ethos kernel. Replacing our original hand-written RPC code with eNotation machine-generated code added 1,124 while removing 1,778 LoC, a net reduction of 654 LoC. More importantly, this use of machine-generated code within Ethos ensures all of the system's RPCs implement formally specified grammars.

This brings libetn, et2g, and eg2source into the Trusted Computing Base (TCB) of Ethos. We note that the TCB typically contains far more than just the kernel—for example, encryption and many other code bases. The lines of code associated with each Etypes component are listed in Table III. Etypes' language support is less than 25% the size of ONC RPC, even though it supports both C and Go.

Application code: Applications also save many lines of code through the use of eNotation. For example, existing mail user, transfer, and delivery agents require much code to read and write various protocols and formats, including Simple

Mail Transfer Protocol (SMTP) message envelopes, Internet Message Format (IMF) message headers, and Multipurpose Internet Mail Extensions (MIME). Each of these is described by a series of Requests for Comments (RFCs). Furthermore, configuration files also require parsing.

These requirements increase application size. We reviewed libcamel [1], a library that implements many mail-related encoders (SMTP, POP3, MIME) and parsers (XML/HTML), and summarize its over 150,000 lines of code in Table IV. Its encoders/decoders support communication and storage, and its parsers are for languages which are used in networking for both clients and servers. In each case, substantial code is needed to handle input and output that can be ill-formed in arbitrary ways.

To illustrate the use of Etypes, we wrote eMsg, a messaging system for Ethos. eMsg is able to send and receive a message whose type is defined in eNotation and invoke RPC functions generated by eg2source. eMsg specifies the equivalent of SMTP and IMF using only dozens of lines of eNotation.

F. Performance

Microbenchmarks: We first analyzed performance by measuring the speed at which our implementation can recognize, encode to, and decode from a memory buffer (Figure 9). We wrote a series of microbenchmarks based on a collection of types, including primitive, vector, and composite types. We tested the speed of encoding and decoding using Etypes, XDR, and JavaScript Object Notation (JSON). We also measured the speed of recognition. An average for each test of encoding, decoding, or recognition is provided.

Figure 9 depicts our results. For scalar types, XDR is the fastest as it benefits from mandatory 32-bit alignment. XDR also encodes pointers faster than Etypes, but this is because Etypes supports cyclic and shared objects. JSON performs the slowest due to its use of runtime type introspection. Etypes' encoding of vectors containing scalar types benefits in the common case of little-endian architectures. C Etypes encoding is 0.136–22.823 (geometric mean: 0.985) times faster than XDR, and its decoding is 0.180–33.084 (geometric mean: 0.995) times faster than XDR. Verification is a common operation in the Ethos kernel, and its speed is 1.375–9.568 times faster than C Etypes encoding.

eMsg performance: Here we compare the performance of eMsg to Postfix. Since Ethos encrypts and cryptographically authenticates all network connections (More details about Ethos networking may be found in parallel publications [45], [44]), we configured Postfix with Transport Layer Security (TLS) encryption and client certificate authentication. We wrote a client program that connects to Postfix using TLS-protected SMTP over a UNIX socket and sends 2,500 emails to the server.

eMsg provides a client/server architecture roughly similar to Postfix. Both the client and server perform type-related work: the client-side kernel must check the well-formedness of messages sent to the network and the server-side kernel must do the same for received messages, as well as check the data that the receiver writes to a user's spool directory.

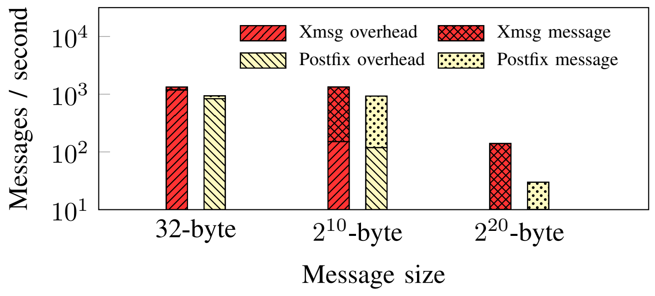

Fig. 10: eMsg performance

Figure 10 shows the performance of eMsg and Postfix, over three message sizes. Each bar consists of two parts: the message itself and its overhead. We modified eMsg so that it incurred an overhead roughly equal to Postfix, 1,229 bytes. (eMsg's natural overhead is smaller because its equivalent to IMF headers is more terse.) For each message size, eMsg was faster than Postfix.

G. Encoding density

eCoding's use of implicit encoding reduces the size of serialized data. There are a few exceptions, where we made space-time tradeoffs. For an any, a type hash identifies the actual type, requiring 64-bytes to precede the eCoding that follows. eCoding encodes nil pointers with a single byte, and the overhead for other pointers is also a single byte. Fixed lengths, as for arrays, need not be encoded, but tuples, strings, and dictionaries encode their length as a 32-bit value. (Choosing a 32-bit length is possible due to the integration of Etypes and Ethos—the maximum Ethos object size is $2^{32}-1$; larger constructions can exist as a collection of objects as discussed in §IV-C.) RPC calls contain a procedure ID as overhead, and discriminated unions contain a 32-bit tag. eCoding is not 32-bit aligned; thus it can encode values of less than 32-bits more efficiently than XDR.

VI. Related work

A. Types, distributed programming, and operating systems

Our design of the eNotation type hash builds on the ideas of network objects' fingerprint [10], [31]. The important additions provided by the eNotation type hash are the use of a modern, cryptographic hash; support for semantic binding through the use of annotations (§III-B); and a deep integration into Ethos (§IV-C),

Early languages designed for creating distributed systems include Argus and Emerald [36], [14]. Both were designed to run on UNIX. Argus provides an object encapsulation called a guardian and atomic actions. Emerald provides a single mechanism to invoke both local and remote objects, and its objects can migrate between nodes.

Java's type system provides isolation within the JX OS [24]. JX OS does not require a Memory Management Unit (MMU)

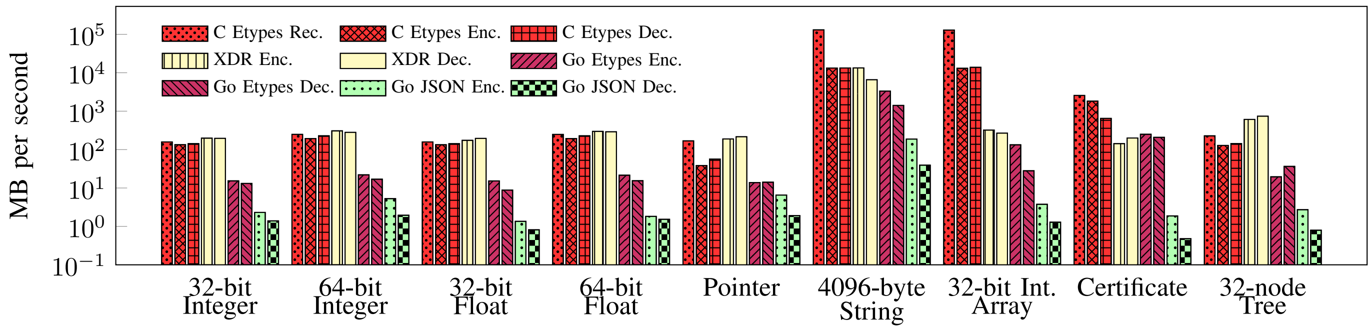

Fig. 9: Microbenchmarks: encode/decode to/from memory buffer

and instead implements software-based stack overflow and nil reference detection. Singularity is written in Sing♯, provides Software Isolated Processes (SIPs), and uses linear types to prevent shared-memory-related race conditions [32]. LISP and Haskell have also been used to construct an OS kernel [26], [28]. Ethos has a different goal than these OSs, namely to remain intentionally multi-lingual.

SPIN allows applications to specialize the OS kernel using extensions [8], [27]. Type safety allows SPIN to dynamically link extensions into the kernel while isolating other kernel data structures. Exokernels likewise maximally embody the end-to-end argument [20]. This has been shown to provide performance benefits, for example by allowing applications to interact more directly with network interfaces [22].

Fabric labels objects with their security requirements and provides distributed-system-wide, language-based access and flow controls [38]. HiStar provides a simplified, low-level interface and implements mandatory information-flow constraints without a language dependency [57]; DStar builds on HiStar to provide information flow across hosts on a network [58]. Developments in HiStar/DStar and Ethos are complementary: flow control will benefit from a higher system-level semantic understanding of the types in a system.

B. Serialization and RPCs

Many mechanisms exist for object serialization. These mechanisms range from the basic (e.g., htons) to the sophisticated. Care must be taken when serializing an object containing pointers which might produce *shared objects*—where two or more objects each reference a given object—or *cyclic objects*—where an object contains a direct or indirect reference to itself. Standard techniques exist to address these requirements [30]. In an *implicitly typed* encoding, only data is encoded, not its type. This reduces encoding size and eliminates type field interpretation while decoding. In *explicit typing*, encodings contain both type identifiers and data.

RPCs are procedure calls that are executed in another address space [11]. Some RPC systems allow communication with remote computers and others—often for performance reasons—allow communication only within a single computer [9], [56]. RPC systems are used both in applications and microkernels. Microkernel RPC is particularly performance-sensitive; its performance is often highly optimized [35].

PL-specific serialization and RPC systems do not need a multilingual Interface Description Language (IDL), and provide conveniences such as the ability to pass previously unknown objects such as subtypes to remote methods. PL-specific systems include Python' pickle [5], Java serialization [25], C++'s Boost [4], and Java's Remote Method Invocation (RMI) [54].

Programmers who use certain PLs or runtimes benefit from the convenience of PL-specific serialization, but many large distributed systems require language independence. Multilingual serialization and RPC systems include XDR [19], ASN.1, JSON [18], Protocol Buffers [46], ONC RPC [51], and CORBA [52]. Etypes behaves most like ONC RPC which is multilingual, has an implicitly typed encoding, and generates code based on type descriptions (eg2source follows rpcgen). CORBA's any types use identifiers that are ambiguous [17], but Etypes uses UUIDs. Ambiguity can violate the type safety of inheritance and any types.

VII. Conclusion and Further Work

An OS-wide type system can make it easier to develop and administer a robust distributed system. In several ways, Etypes resembles ONC RPC which is multilingual, has an implicitly-typed encoding, and generates code based on type descriptions. But Etypes is unique due to its tight OS integration.

Ethos' clean-slate design enables deep integration and simple semantics. The filesystem plays a critical role in maintaining type information and simplifying failure handling through streaming directories. Surprisingly, Etypes impacted the design of other parts of the system, most notably the filesystem. Because of Etypes, Ethos does not have seeks (it uses directories instead), treats directories as streaming entities, and associates file metadata with the file's parent directory. This has a profound impact on Ethos' shell, which is type, rather than string, based.

Ethos performs recognition in its kernel, ensuring that applications only see well-formed data that matches their expected type. This provides important security protections such as ensuring consistent treatment of objects, removing parsing ambiguities, and removing substantial parsing code.

Parsing code is an especially attractive target for attackers due to its size, complexity, and direct availability. Input is traditionally the Achilles heel for security, a direct entry point into the program which can be (and often has been) exploited by attackers.

The greatest advantage of parsing over Etypes-style typing is that parsing can recognize legacy protocols. On the other hand, typing not only recognizes input but creates a data structure for processing. Parsing typically creates such a data structure in the actions associated with syntax recognition, rather than as a formal component of the parser. Etypes handles legacy protocols through the use of protocol proxies.

Ethos' application APIs are as straightforward to use as untyped APIs on traditional systems, while performing many chores for the application programmer and thus reducing application code size.

Ethos' type hash allows separately developed components to be later combined with predictable results, and annotations remove type ambiguity while documenting processing invariants. Most importantly, both eliminate the need for central type naming authorities.

We plan to build on Etypes' guaranteed properties. It is already *not* possible to evade the OS' conformance checks. Currently, programmers are discouraged from directly using Ethos' low-level systems calls; a future implementation will eliminate this access, forcing the use of Etypes' encode/decode routines.

One of the most interesting areas is the design and implementation of eL. Etypes' uniformity makes it easier to write utilities and scripting languages which enable system administrators to better manage their systems. Our goal is to make Ethos accessible to system administrators through eL scripts that are as useful as UNIX's text-based scripting languages even while manipulating richer types.

ACKNOWLEDGMENTS

This material is based upon work supported by the US National Science Foundation under grant CNS-0964575. We also thank Chris Okasaki and Xu Zhang for their reviews of an earlier draft, our anonymous reviewers for comments on subsequent drafts, and the United States Military Academy and Army Cyber Center for their support.

REFERENCES

[1] Camel—a mail access library. https://wiki.gnome.org/Apps/Evolution/Camel (accessed Feb 5, 2014).

[2] Hammer parser generator. https://github.com/UpstandingHackers/hammer (accessed Feb 5, 2014).

[3] Microsoft windows PowerShell. http://technet.microsoft.com/en-us/library/ms714418.aspx (accessed Apr 1, 2014).

[4] Boost documentation, serialization. http://www.boost.org/doc/libs/1_48_0/libs/serialization/ (accessed Apr 1, 2014), 2009.

[5] Python documentation, python object serialization. http://docs.python.org/library/pickle.html (accessed Apr 1, 2014), 2010.

[6] Argyris Arnellos, Dimitrios Lekkas, Thomas Spyrou, and John Darzentas. A framework for the analysis of the reliability of digital signatures for secure e-commerce. *The electronic Journal for e-commerce Tools & Applications (eJETA)*, 1(4), 2005.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, Bolton Landing, NY, USA, October 2003. ACM.

[8] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *SOSP*, pages 267–284, Copper Mountain Resort, Colorado, December 1995.

[9] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8:37–55, February 1990.

[10] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. In *SOSP*, SOSP '93, pages 217–230, New York, NY, USA, 1993. ACM.

[11] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. In *Proceedings of the 9th Symposium on Operating System Principles*, New York, NY, USA, October 1983. ACM.

[12] Prithvi Bisht, A. Prasad Sistla, and V. N. Venkatakrishnan. Taps: automatically preparing safe sql queries. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 645–647, New York, NY, USA, 2010. ACM.

[13] Prithvi Bisht and V.N. Venkatakrishnan. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *DIMVA*, Paris, France, 2008.

[14] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distrbution and abstract types in emerald. *IEEE Trans. Softw. Eng.*, 13(1):65–76, January 1987.

[15] Francesco Buccafurri, Gianluca Caminiti, and Gianluca Lax. Fortifying the Dalí attack on digital signature. In *Proceedings of the 2nd International Conference on Security of Information and Networks*, SIN '09, pages 278–287, New York, NY, USA, 2009. ACM.

[16] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, 2004.

[17] S. C. Crawley and K. R. Duddy. Improving type-safety in CORBA. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 291–304, London, UK, UK, 1998. Springer-Verlag.

[18] D. Crockford. RFC 4627: The application/JSON media type for JavaScript Object Notation (JSON), July 2006. Status: INFORMATIONAL.

[19] M. Eisler. RFC 4506: XDR: External data representation standard, May 2006. Status: INFORMATIONAL.

[20] Dawson R. Engler, M. Frans Kaashoek, and Jr. James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *SOSP*, pages 251–266. ACM SIGOPS, December 1995.

[21] Kathleen Fisher and Robert Gruber. PADS: a domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 295–304, New York, NY, USA, 2005. ACM.

[22] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Briceño, Russell Hunt, and Thomas Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst.*, 20(1):49–83, February 2002.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[24] Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX operating system. In *USENIX ATC*, pages 45–58, Berkeley, CA, USA, 2002. USENIX Association.

[25] Todd M. Greanier. Flatten your objects: Discover the secrets of the Java Serialization API. *JavaWorld*, July 2000.

[26] Richard D. Greenblatt, Thomas F. Knight, John T. Holloway, and David A. Moon. A lisp machine. In *CAW '80: Proceedings of the fifth workshop on Computer architecture for non-numeric processing*, pages 137–138, New York, NY, USA, 1980. ACM.

[27] Robert Grimm and Brian N. Bershad. Separating access control policy, enforcement, and functionality in extensible systems. *TOCS*, 19(1):36–70, 2001.

[28] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 116–128, New York, NY, USA, 2005. ACM.

[29] Robert J. Hansen and Patterson Meredith L. Guns and butter: Towards formal axioms of input validation, 2005.

[30] Maurice P. Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM Trans. Program. Lang. Syst.*, 4:527–551, October 1982.

[31] Jim Horning, Bill Kalsow, Paul McJones, and Greg Nelson. Some useful Modula-3 interfaces. Report SRC-RR-113, Digital Systems Research Center, December 1993.

[32] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[33] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1435–1441, New York, NY, USA, 2006. ACM.

[34] Suman Jana and Vitaly Shmatikov. Abusing file processing in malware detectors for fun and profit. In *Proc. IEEE Symp. Security and Privacy*, San Francisco, CA, May 2012.

[35] Jochen Liedtke. Improving IPC by kernel design. In Barbara Liskov, editor, *SOSP*, pages 175–188, New York, NY, USA, December 1993. ACM Press.

[36] Barbara Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, March 1988.

[37] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst.*, 5(3):381–404, July 1983.

[38] Jed Liu, Michael D George, K Vikram, Xin Qi, Lucas Waye, and Andrew C Myers. Fabric: a platform for secure distributed computation and storage. In *SOSP*. ACM Request Permissions, October 2009.

[39] Moxie Marlinspike. Null-prefix attacks against SSL/TLS certificates, July 2009. http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf (accessed Feb 5, 2014).

[40] Bob Martin, Mason Brown, Alan Paller, and Dennis Kirby. 2010 CWE/SANS Top 25 Most Dangerous Software Errors. Technical report.

[41] Per Martin-Löf. Intuitionistic type theory, 1984.

[42] Giovanni Nebbiante and Jon A. Solworth. El: Ethos shell language, in submission.

[43] W. Michael Petullo and Jon A. Solworth. Digital identity security architecture in Ethos. In *Proceedings of the 7th ACM workshop on Digital identity management*, pages 23–30, New York, NY, USA, 2011. ACM.

[44] W. Michael Petullo and Jon A. Solworth. Simple-to-use, secure-by-design networking in Ethos. In *Proceedings of the Sixth European Workshop on System Security*, EUROSEC '13, New York, NY, USA, April 2013. ACM. https://www.ethos-os.org/papers/.

[45] W. Michael Petullo, Xu Zhang, Jon A. Solworth, Daniel J. Bernstein, and Tanja Lange. MinimaLT: Minimal-latency networking through better security. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS '13, New York, NY, USA, November 2013. ACM.

[46] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming*, 13(4):277–298, October 2005.

[47] Rob Pike and Peter Weinberger. The hideous name. In *USENIX Summer Conference Proceedings*, pages 563–568, Portland, Oregon, USA, June 1985.

[48] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto. Security applications of formal language theory. *IEEE Systems Journal*, 7(3):489–500, 2013.

[49] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Anna Shubina. The halting problems of network stack insecurity. *;login: the USENIX Association newsletter*, 36(6):22–32, December 2011.

[50] Zhendong Su and Gary Wassermann. The essence of command injection attacks in Web applications. *ACM SIGPLAN Notices*, 41(1):372–382, January 2006.

[51] R. Thurlow. RFC 5531: RPC: Remote procedure call protocol specification version 2, May 2009. Status: INFORMATIONAL.

[52] Steve Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, February 1997.

[53] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 347–359, Sea of Galilee, Israel, 1990. North Holland.

[54] Jim Waldo. Remote procedure calls and Java remote method invocation. *IEEE Concurrency*, 6:5–7, July 1998.

[55] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through Facebook and Google: A traffic-guided security study of commercially deployed single-sign-on web services. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 365–379, Washington, DC, USA, 2012. IEEE Computer Society.

[56] Michal Wegiel and Chandra Krintz. Cross-language, type-safe, and transparent object sharing for co-located managed runtimes. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 223–240, New York, NY, USA, 2010. ACM.

[57] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazires. Making information flow explicit in HiStar. In *OSDI*, Seattle, Washington, November 2006.

[58] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *NSDI*, NSDI'08, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.