Ethos' Distributed Types

W. Michael Petullo* Wenyuan Fei* Pat Gavlin† Jon A. Solworth*
*University of Illinois at Chicago †Microsoft

Abstract

Programming languages have long incorporated type safety, increasing their abstraction level and thus aiding programmers. Type safety eliminates whole classes of bugs, avoiding the tedious and error-prone search for these bugs. Type-related bugs are particularly dangerous and often result in security holes because they violate programmer expectations. Despite its benefits, type safety protections traditionally end at the process boundary.

We describe the deep integration of type safety in Ethos, a clean-slate operating system, which ensures that communication and files conform to their expected type. Ethos types are language- and runtime-agnostic, and incorporate a new mechanism for automatically creating type identifiers. We explore the impact of type safety on type descriptions, filesystem structure, scripting languages, file streaming, and application programming.

1 Introduction

Type safety prevents **untrapped errors**—where an error goes unnoticed and computation continues—and can reduce **trapped errors**—where an error is detected and computation stops [12]. Untrapped errors are particularly pernicious since they result in arbitrary behavior; this arbitrary behavior occurs when a program's runtime state becomes inconsistent. Deep implementation knowledge and extensive analysis can be required to predict the execution after an untrapped error. For example, C's lack of type safety permits untrapped errors such as buffer overflows [15, 13].

Thus Programming Language (PL) designers have long incorporated type safety to reduce the effort of writing applications. Milner stated "*well-typed programs cannot 'go wrong'*" [36]. While surely an exaggeration, strongly checked type systems increase program quality.

But type-safe PLs are not sufficient: distributed systems are often written in multiple languages, and their applications must be integrated. Even for distributed applications written in a single language, it is beneficial to ensure consistent type use. PL type systems (and related techniques, e.g., [4]) primarily provide internal program consistency. In contrast, Operating System (OS)-based types can constrain input and output to conform to program expectations.

The most compelling reason for adding types to an OS is for security. Existing distributed systems often act insecurely due to ill-structured input and output (§5.1). Many applications dedicate a significant amount of code to parsers (or in some cases decoders) that translate untyped data into a language's native form. Attackers often can craft input to exploit vulnerabilities in this code (§5.2). An attacker might also fool a program into generating data that it should not.

To address such issues, we have developed Ethos, a clean-slate OS designed for security. Because OSs provide complete mediation, they uniquely can provide system-wide guarantees that complement the application-local guarantees of a PL. We explore here the role OSs can have in performing type checking on the data that flows between programs in a distributed system.

We call Ethos' type system **Etypes**. Processes send data to other processes—either directly through Inter-Process Communication (IPC)[1] or indirectly through the filesystem—in the form of (typed) objects. Each object has at least two representations, one in a running program and another external representation. Etypes converts between external serialized objects and internal program memory and Ethos subjects all process input and output to a type checker. This type checker maintains a property we call object integrity, which we define in §2.1.

Ethos is the first OS to subject its system calls to a language-agnostic system-wide type checker. Like authorization, programmers define types in advance of use, and Ethos restricts (in conjunction with authorization) what types programs can read or write. This limits the attacker's ability to introduce ill-typed data and reduces surprises—and therefore vulnerabilities—during application reads. Etypes' contributions include:

C1 Universally Unique IDs (UUIDs) for types without naming authorities (§4.4). Types are independently defined, guaranteed not to conflict with types created elsewhere, and usable anywhere. This is necessary for loosely coupled [40] distributed systems.

C2 Enabling OS-based type checking of all process input and output (§4.3) through OS-defined types.

C3 Tightly integrated PL/system support (§4.2, §4.3), reducing application code, eliminating type errors, and impacting the design of scripting languages.

We next survey types in general, including a series of definitions. In the remainder of the paper, we discuss types in general, including a series of definitions (§2); an overview of Etypes (§3); our design (§4); an evaluation

[1] Both local IPC and networking share one API (§4.3).

of Etypes (§5); and related work (§6). Our evaluation focuses on Etypes' security benefits and performance.

2 Types

PLs can be **untyped**, in which a variable can hold values of any type; or they can be **typed**, in which each variable is restricted by a type [12]. Type-safe PLs have no untrapped errors. This is ensured either by **dynamic checking** in untyped languages (or certain language features) or by **static checking** a typed program prior to execution. 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 with respect to applications. The result is that applications written on top of such OSs can fail silently and in unpredictable ways. Some OSs use type safety for internal consistency, to eliminate errors within the OS or at the OS interface. For example, dynamic checkers have been used to prevent buffer overflows within kernels. Static type checkers have been used to regulate either internal Remote Procedure Calls (RPCs) or code extensions.

The above use of types for internal consistency is much more tightly-coupled than the approach we pursue here; their environment is characterized by fixed PLs and are used by OS maintainers which are a tightly-knit group. In contrast, we examine the scenario where one application reads data that another produces; these applications may have different developers, be on different systems, and use different PLs. In such a setting it is important not to overly restrict types. The mechanisms presented here support an open-ended system.

Hence, Ethos' type system is different in several ways from PL type systems. First, the equivalent of variables—files and IPC—are typed like a typed PL, but checked dynamically. Second, we have made our design as modular as possible: Etypes is not limited to a single application PL or application virtual machine (e.g., Common Language Runtime (CLR)). Finally, we have kept our type system general (as OS mechanisms must be) so that it may be applied universally while assuring useful soundness properties.

2.1 Object integrity

Ethos assures object integrity by using its type checker, as we will describe in §4.3. We define **object integrity** as: (1) objects are read or written in whole, thus preventing short or long reads/writes [47], (2) objects (external or memory) must be consistent with their type, and (3) an object which is written by one program and read by another must produce an *equivalent object* (see below). A system call that writes/reads object o will either succeed if o is well-typed or return an error without application side effects. Managing side effects in this way simplifies application development, and is a strategy shared with other systems [34].

Our notion of object includes not only the object directly referenced (o_0), but also those objects which are reachable from o_0 via pointers ($o_1, o_2, \ldots o_n$). Given this numbering of objects, the reconstituted object must consist of n objects $o'_0, o'_1, \ldots, o'_n$, such that the type of o_i is equal to o'_i, and the value of o_i is equivalent to o'_i. Two objects o_i and o'_i are **equivalent** if (1) each pointer field in o_i that points to o_j has a counterpart field in o'_i that points o'_j and (2) each non-pointer field in o_i is equal to its counterpart in o'_i.

2.2 Business rules

There may be additional restrictions on an object 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. We call these higher-level semantics **business rules**.

Though rules are the responsibility of applications, they can be described with a type's Etypes annotations (§3.2). Thus an application developer has documentation of the business rules expected to be enforced by his application.

3 Etypes

With Etypes, programmers specify types and RPC interfaces using Etypes Notation (eNotation) which, like External Data Representation (XDR), is a data description language (§3.1). Etypes' serialized wire format is called Etypes enCoding (eCoding). Etypes has three fundamental operations: **encode**, which takes a PL type and serializes it to an eCoding; **decode**, which takes an eCoding and sets an appropriately typed PL variable to its value; and **check**, which Ethos uses to implement its type checker. Etypes addresses both the syntax and semantics of types (§3.2).

3.1 Etypes Notation

eNotation describes types using a syntax based on Go. Table 1 lists the eNotation types, a syntax example for each, and their corresponding eCoding. Primitive types include integer (both signed and unsigned), floats, and booleans. Composite types include pointers, arrays, tuples, strings, dictionaries, structures, discriminated unions, any, and RPC interfaces. RPCs, unions, and any types, warrant further description.

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. Although eNotation specifies RPC func-

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

Table 1: 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.

```
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 }
```

Figure 1: An eNotation structure representing a bank transfer certificate

tions in a traditional way, their 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. One-way RPCs are a very simple, yet entirely general, mechanism.

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, Etypes includes the object's actual type identifier (§4.4) in the encoding.

3.2 Syntax and semantics

eNotation's **annotations** informally describe semantics beyond types. Annotations (1) contribute to a type's UUID, binding syntax and semantics together; (2) differentiate structurally identical types; and (3) enable integrity requirements beyond typing to be expressed (§2.2). Application programmers, administrators, and users refer to a type's annotations to determine an object's meaning, and thus annotations minimize the *semantic-level difference* [5] as understood by Ethos users.

Consider the certificate type described in Figure 1, which is a digitally signed bank transfer order. It contains a certificate header and four certificate-specific fields: a bank ID, two account numbers, and the transfer amount. Field names, i.e., bankId, contribute to type UUIDs but are insufficient for detailed descriptions. Annotations narrow the semantic-level difference, for example by describing the bankId field as an American Bankers Association transit number (at line 10). More complex annotations can be in an external file referenced by an eNotation specification.

4 Design

We designed Etypes for Ethos, an OS designed to provide the foundation for next-generation robust distributed systems. Ethos is decidedly clean-slate, with a goal of substantially reducing overall software complexity. Since it's possible to prove an OS correct [27], but infeasible to do so for every application, Ethos focuses on reducing application complexity and vulnerabilities. Ethos currently provides memory paging, processes, encrypted networking, and a filesystem. We implemented 39 system calls, 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.

Ethos is a "security first" OS, even to the extent of being incompatible with existing applications and network protocols. Ethos provides compatibility with existing network protocols through network proxies. But internally, Ethos is designed without compatibility constraints—avoiding existing complex and not-designed-for-security interfaces. Adequate security—beyond that in today's systems—cannot be added after the fact [14]. We focus here on the design, implementation, and analysis of general-purpose interfaces which protect applications against attack.

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 are building an Etypes scripting language called eL. Ethos prohibits, via authorization, applications in C in keeping with Ethos' security first goal, as it is far more difficult to write secure programs in C.

Targeting different PLs presents both a challenge and opportunity. In §4.1, we describe how Etypes remains PL-agnostic through the use of type graphs, and we explain why multiple PLs are needed to meet the various requirements of Ethos in §4.2. We discuss issues related to the deep integration of Etypes into Ethos in §4.3.

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

Etypes identifies its types using something we call a **type hash**—a UUID based on a cryptographic hash. The type hash is a fully distributed mechanism, naming each type in a unique but predictable manner, yet does not require *any* naming authorities. Type hashes also support the requirement of versioning. We describe the algorithm to compute type hashes in §4.4.

4.1 Type graphs

Given an eNotation specification, a utility named et2g generates a PL-independent type graph and its type hashes. A **type graph** is a directed graph containing type descriptions as nodes and type references as edges. A type graph identifies types by universally unique hashes rather than the local names used in its eNotation specification. 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.

Ethos forbids creating an OS object of type T unless T is present in the system's type graph (§4.3). Ethos uses the type graph to check types, and user-space utilities (such as eg2source as described below) also make use of the same graph. Type graphs themselves are stored as eCoding. Table 2 shows the contents of a graph node.

4.2 PL integration

Our type system is PL-agnostic; nevertheless it has profound impacts on the PLs Ethos supports. In particular, 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 to encode and decode types. For RPCs, eg2source creates stubs and skeletons, similar to ONC RPC or CORBA.

Unsafe PLs Types can be added to a running system, so check must be table-driven and thus able to verify unanticipated types. On the other hand, encode and decode require compile-time type definitions, due to C's static typing. Currently encode and decode are table-driven, but unlike for check, this is not a requirement (§7). Thus Etypes' C implementation is table-driven, because the Ethos kernel uses check. eg2source generates tables which describe types, and a library called liben walks these tables to encode, decode, or check the types. Since liben depends only on external malloc- and free-like functions, it is easily integrated into both the OS kernel and PL runtimes. Etypes simplifies kernel code by subsuming tedious, manual encode, decode, and verification routines.

C is not type-safe. Using Etypes in C can have untrapped errors, such as a mismatch between an eNotation type and C variable. However, since the use of C on Ethos is limited to system software, its use can be subject to more rigorous code inspection. We intend to eventually provide a proof of correctness.

Type-safe, statically-checked PLs Go is type-safe and statically-checked; such languages are where eNotation is aimed. Each Go program contains only a fixed number of compile-time types. Like Go, eNotation types are declared in advance and the types throughout Ethos are restricted. This restriction increases application security by reducing the surprises with which applications need to contend.

Our Go Etypes implementation generates per-type encode/decode routines using eg2source, unlike C's liben- and table-based implementation. The code for using an arbitrary type T is shown in Figure 2. Figure 2a creates an IPC channel and sends the value t of type T on it. Figure 2b accepts an IPC channel and receives an object of type T on it. The procedures Ipc, WriteT, Import, and ReadT are generated by eg2source; thus the system calls necessary to implement these operations are hidden behind typed APIs. In keeping with Ethos' goal of minimizing application complexity, Etypes calls requires no more application code than untyped I/O calls in other OSs. But Ethos calls do more, reducing the total amount of application code.

Type-safe, dynamically-checked PLs Statically-checked languages are preferred for traditional applications, because they provide higher integrity. However, utilities often benefit from dynamic types. Consider traversing a filesystem recursively and displaying the contents of files—here types may not be known at compile time. We are building eL, a dynamically-checked language that will integrate with Etypes.

```
1 e,d := en.Ipc(hostname, serviceName)
2 e.WriteT(&t)
```

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

```
3 e,d,u := en.Import(serviceName)
4 t := d.ReadT()
```

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

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

```
1 seen = {}

2 isATree(x)
3   forall f in fields(x)
4    if isPtr(f) then
5     if x.f in seen then
6      return false
7     seen = seen union { x.f }
8     if !isATree(x.f) then
9      return false
10  return true
```

Figure 3: isATree in eL

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. 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 are the text-based utilities used to manipulate system state. Since UNIX was designed, these utilities have been diminished as types have become richer. Generic utilities, which processes types dynamically, are essential to making Ethos accessible to system administrators and others. Etypes will enable UNIX-like utilities that manipulate richer data.

4.3 OS integration

Here we discuss the integration of Etypes with Ethos. In particular, we describe how Ethos associates a type with every streaming file descriptor or file, and how Ethos uses such types to regulate system calls.

Type checking overview Ethos verifies all network reads and writes as well as file or local-IPC writes using check. Filesystem encryption ensures that file writes must go through the Ethos kernel. Ethos traps ill-formed objects when writing rather than reading to aid correctness and problem diagnosis. Such type checking, like authorization, prevents mismatches between processes.

Associating types with objects Ethos applies the same type to all objects in a given filesystem directory; that is, a directory may contain only objects of a single type. Since filesystem paths also name IPC services, directories determine the types of both files and IPC connections. Thus a program can only read/write object types corresponding to directories it is allowed to access. While this increases the number of directories, it has the advantage that Ethos can enforce the type-safety of file creation transparently—a write does not need to specify a type. We provide an example Ethos program that accesses a file in §4.5.

In Ethos, types must be specified only 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 *dirFd*, *name*, authorization *label*, and type *uuid*:

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

We expect directory creation to be primarily handled by administrative tools. If the directories are set up outside of a particular application, the application and type policy are completely independent. When a directory naturally contains various types, applying the any or union types allow it to be used 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).

Ethos provides a writeVar system call to write a file in its entirety—an Ethos file is not a streaming object.

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

The inverse of writeVar is readVar.

```
readVar(dirFd, fileName)
```

Rather than directly use the above system calls, application programmers use encode/decode interfaces (§4.5).

Seek It may seem odd not to support seeking within files; after all, video files can span many gigabytes. But this is necessary for simple failure semantics—an object is either of the correct type and the readVar/writeVar syscall succeeds or it is not and the operation fails with no application side effects.

When designing Ethos, we had originally sought to provide 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 an application would likely introduce side affects 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.

Ethos supports very large objects with directory streaming rather than with very large files, which we discuss next. Individual files must be read in their entirety, and so Ethos bounds file sizes to conserve memory. (To support legacy file formats, Ethos uses network proxies which convert between legacy formats and an eCoding.)

Streaming IPC and directories In Ethos, IPC and directories are streaming entities, and Ethos enforces the type of each write. The write system call sends data out on a streaming descriptor:

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

Likewise, the read system call reads from a streaming descriptor. Again, the data must conform.

```
read(descriptor)
```

We provide an example of Ethos IPC in §4.5. Streaming directories present another challenge, because their files must be named to preserve their order. The write system call, when applied to a directory, creates a file named with the current time; the read system call, when applied to a directory, reads the next file in filename order. Thus Ethos streams over a directory of files (e.g., video frames), whereas each file is non-streaming (i.e., read in its entirety).

Networking Programmers on Ethos open network connections using the same Application Programming Interface (API) as local IPC; a non-empty hostname argument to lpc implies a network connection. Etypes takes care of many networking chores at the OS-level including encryption, user authentication, endianess, alignment, parsing, and data value encoding, so that networking just works. Because network data comes from a foreign system, Ethos applies check to network reads as well as writes. Higher-level interfaces also allow Etypes optimizations independent of application code. Space prevents us from describing further details of Ethos networking which we discuss elsewhere [38, 37]. Here it suffices to say that no Ethos application can receive ill-typed data from the network, because first the data must satisfy Ethos' type checker.

Type graph Ethos stores the system type graph in its filesystem at /types. Both kernel-internal 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. It is the directory /types/all/ that is loaded by Ethos at boot time and reloaded on demand.

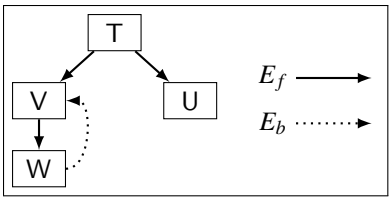

Figure 4: A graph partitioned into E_f and E_b

We envision that application type hashes will be installed by the packaging system and will remain until the 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.

4.4 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.

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'.) Thus E_f contains all edges excluding those that would result in a cycle. The remaining backward edges, $E_b = E - E_f$, are those which would introduce a cycle. 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

```
1 T struct {                                  h5 = hash(T struct { aRef *h2 anotherRef *h4 })
2         aRef *U
3         anotherRef *V
4 }

5 U struct { anInteger uint32 }               h2 = hash(U struct { anInteger uint32 })
6 V struct { leadsToACyclicRef *W }           h1 = hash(V struct { leadsToACyclicRef *W })
                                              h4 = hash(V struct { leadsToACyclicRef *h3 })
7 W struct { aCyclicRef *V }                  h3 = hash(W struct { aCyclicRef *h1 })
```

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

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

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.

Algorithm 1 typeHash(t)

```
1: [E_f, E_b] ← 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 ∃[n'', n] ∈ E_f | noOut(n, E_f) do
2:     for all [n, n'] ∈ E_b do
3:         h ← hash(n')
4:         replace references to n' in n with h
5:         E_b ← E_b − {[n, n']}
6:     end for
7:     E_f ← E_f − {[n'', n]}
8: end while
```

Algorithm 3 collapse(E_f)

```
1: while ∃[n', n] ∈ E_f | noOut(n, E_f) do
2:     h ← hash(n)
3:     replace references to n in n' with h
4:     E_f ← E_f − {[n', n]}
5: end while
```

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 edge that satisfies Line 1 is $[V,W]$, 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.

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

(a) Example eNotation

```
5 a := en.TypeA {uint32(0), uint64(1)}
6 d := syscall.OpenDirectory ("/someDir/")
7 e := en.WriteVarTypeA(d, fileName, a)
```

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

```
8 d := syscall.OpenDirectory ("/someDir/")
9 a := en.ReadVarTypeA(d, fileName)
10 switch a.V.(type) {
11  case uint64: // Of actual type uint64.
12 }
```

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

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

4.5 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 any. Figure 6b demonstrates how to write TypeA. Line 5 initializes a to a TypeA structure, Line 6 opens a directory, and Line 7 writes a to a file named *fileName* in the directory. Attempting to write an ill-formed object relative to the type associated with /someDir/ would cause a runtime error.

Figure 6c provides the inverse. Here we decode to a native Go struct. Trying to decode from a file using the wrong decode function (e.g., ReadVarTypeB) would result in a compile-time error.

Any types We now describe the details of using the any type in Figure 6. The any type in TypeA must be of an actual type known to Ethos. (Not shown is the error handling at 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. Decod-

```
1 I interface {
2   Add(i uint32, j uint32) (r uint32)
3 }
```

(a) Example eNotation: an RPC interface

```
4 e, d := en.Ipc (hostname, serviceName)
5 e.IAdd(0, 1)
6 d.IHandle(e)
```

(b) Example code: invoke RPC

Figure 7: Invoking an RPC and handling the response

ing 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 7 provides an example of invoking an RPC. Figure 7a defines an example RPC interface containing a single function, Add. Not depicted is a detailed annotation describing Add.

Figure 7b provides the body of an application. It opens a network connection using Ipc and initializes *e* and *d* to the returned encoder and decoder objects, respectively. These, in turn, are wrappers for Ethos' read and write system calls, and provide access to the generated RPC stub/skeleton routines. The program next invokes *e*'s IAdd function, thereby making an RPC request. Calling *d*'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 call 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.

5 Evaluation

Our evaluation focuses on the security properties provided by the integration of Etypes with Ethos. We first consider how Etypes addresses semantic-gap attacks, addresses parsing bugs, and conserves application code. We then present performance measurements of our C and Go implementations using microbenchmarks; here we compare our results to XDR/ONC RPC and JavaScript Object Notation (JSON). Finally, we present a more realistic use case, analyzing the performance of a messaging system we wrote for Ethos. Although Ethos has security over performance as a design goal, its performance must be acceptable—we show that Etypes performs well.

Ethos is a paravirtualized OS running on top of Xen 4.1 [6]. We ran our tests on computers with 4.2 GHz AMD FX-4170 quad-core processors, 16GB of memory, and a gigabit Ethernet adapter.

5.1 Semantic-gap attacks

Buccafurri et. al presented the Dalí attack in the context of digital signatures [11], and Jana et. al presented chameleon attacks in the context of anti-virus software [32]. In these attacks, the type is unknown and is therefore determined in an ad hoc, and sometimes erroneous, manner. Ethos provides countermeasures for both attacks. The type of each Ethos object is known (§4.3), every application and utility interprets an object consistently by its type, and each type has a universal meaning defined by its annotations.

Consider two types t_1 and t_2 with identical structure; they nonetheless have different hashes in Ethos due to their semantic description (§3.2). An Ethos application commits to a type when it reads or writes data as an Ethos object, and Ethos enforces the type it chooses. Of course, an application could erroneously swap data of type t_1 for t_2. This error is unavoidable through structural type checking alone, but Ethos' authorization system can restrict the application so that it can only write Ethos objects of type t_1. 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 beyond the object is read [47]. In these attacks, it is possible to get confused as to the sources of input, mistaking untrusted for trusted information.

Certificates present a particular example of the interaction between Ethos authorization and Etypes. Without Etypes, an attacker might trick a flawed program (such as a user-downloaded game) to sign a bank transfer request with a user's secret key. Thus we integrate types into Ethos' certificate system; the CertificateHeader fields in Figure 1 include a type hash. When servicing a sign system call, Ethos checks the type of the output directory, and sets the certificate's type to match before signing it on behalf of the user. This binds a meaning to the certificate (which might otherwise have an identical structure to another), and prevents the game from producing one, as the game would be restricted from writing to any directories bearing the transfer certificate type. Of course, a system could alternatively maintain different keys for different purposes, but eventually two seemingly separate certificates will need to be bound to the same person.

5.2 Parsing vs. encoding/decoding

Parsing is vulnerable to subtle attacks. Jana describes the difficulty of writing parsers that prevent werewolf attacks [32]; for anti-virus, the difficulty is that virus detectors and applications can parse data differently. Similar difficulties are encountered when defending against Cross-site Scripting (XSS) attacks because a server's input validation must anticipate how a client browser will parse HTML [10]. Ethos takes a different approach—it prefers decoders, types are enforced by the OS, and for any type only a single decoder exists.

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

Table 3: 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 4: Lines of code in selected libcamel components

In comparison with decoders, parsing has two disadvantages. First, parsing can be ambiguous, leading to confusion. Second, parsing is incomplete—it produces a parse tree which still must be processed—while decoding produces an actual object to be operated upon.

Parsing is complex and deals with untyped input streams, perhaps directly generated by an attacker. In Ethos, there are no untyped input streams—Etypes translates between external and internal representations while being subjected to a type checker. Thus applications know their input is well-typed. Furthermore, Etypes' encoders/decoders must be generated from a high-level eNotation description. Only one encoder/decoder generator (eg2source) is needed per PL, and formal methods can ensure that they are correct. We describe next how eg2source benefits the kernel and applications.

Kernel As previously discussed, we use liben in the Ethos kernel. Replacing our original hand-written RPC code with eNotation machine-generated code added 1,124 while removing 1,778 Lines of Code (LoC). This is a net reduction of 654 LoC. More importantly, using machine-generated code reduces the hard-to-isolate bugs often resulting from RPC interface changes. This brings liben, 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 3. Etypes' language support is less than 25% the size of ONC RPC, even though it supports both C and Go.

Applications Applications also save many lines of code through the use of eNotation. For example, we observe that existing mail user, transfer, and delivery agents require an implementation of several parsers, including for 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 4. Its encoders/decoders support communication and storage; its parsers are for languages which are used in networking for both client and server.

Tools such as bison and PADS [20] exist to aid in writing parsers. They clearly reduce parser costs, and they are useful on Ethos too when parsing is a necessity. They compliment Ethos' system-wide protections, which are independent of application programmers.

With Ethos, eg2source generates the code for serialization/deserialization from an eNotation description. Programs cannot accept or produce data not in accordance with its eNotation specification since Ethos will not allow it.

5.3 Encoding density

The use of the type hash to identify an any type means that it takes 64-bytes to explicitly encode the actual type associated with an eCoding. For types other than any, eCoding's use of implicit encoding results in a minimal encoding size. There are a few exceptions, where we made decisions affecting overhead. eCoding encodes NULL pointers with a single byte, and the overhead for other pointers is also a single byte. Arrays need not have their known length 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 §4.3.) RPC calls contain a procedure ID as overhead, and discriminated unions contain a tag. eCoding is not 32-bit aligned; thus it can encode values of less than 32-bits more efficiently than XDR.

5.4 Performance

Microbenchmarks We first analyzed performance by measuring the speed at which our implementation can check, encode to, and decode from a memory buffer (Figure 8). 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 JSON. We also measured the speed of type checking. An average for each test of encoding, decoding, or checking is provided.

Figure 8 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.

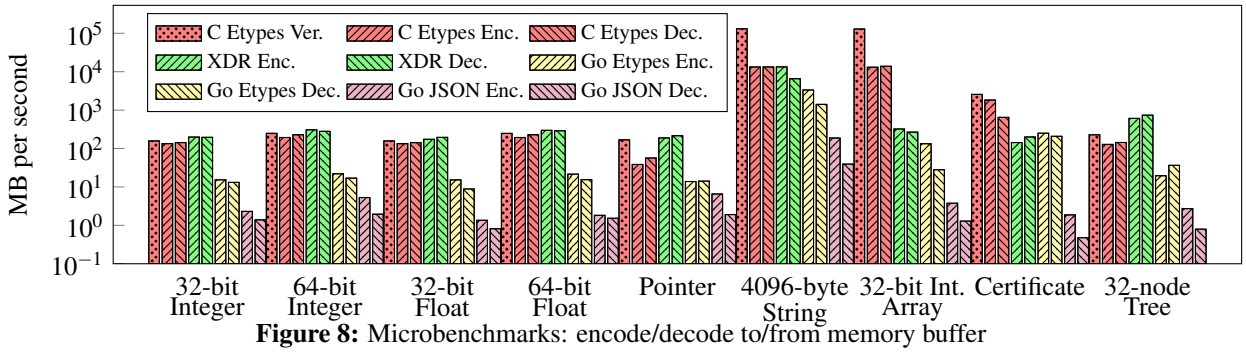

Figure 8: Microbenchmarks: encode/decode to/from memory buffer

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 at 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 Ethos kernel and its speed is 1.375–9.568 times faster than C Etypes encoding.

eMsg performance 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.

We compare the performance of eMsg to Postfix. Since Ethos encrypts and cryptographically authenticates all network connections, 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 must check the well-formedness of messages sent to the network and the server must do the same for received messages, as well as check the data that the receiver writes to a user's spool directory.

Figure 9 shows the performance of eMsg and Postfix, over three different 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 overhead is actually smaller because it does not use IMF headers.) For each message size, eMsg was faster than Postfix.

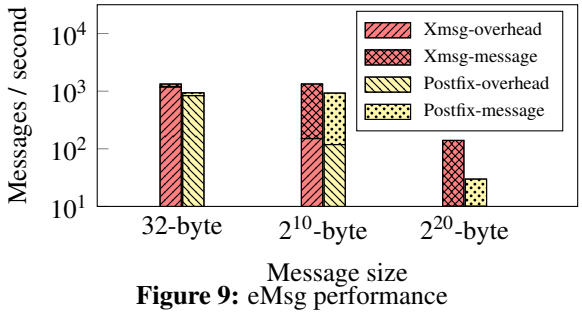

Figure 9: eMsg performance

6 Related work

6.1 Types and operating systems

Types are widely used and have been developed primarily for PLs. Specialized types in PLs are useful for preventing errors, but in general depend on PL semantics and tightly-coupled code. Examples include linear types [45] which can ensure at most one reference to each object, and union types [31] which can allow performing only operations valid for two or more types. In our environment, such semantics appear to be too restrictive.

Java's type system provides isolation within the JX OS [22]. JX OS does not require a Memory Management Unit (MMU) and instead implements software-based stack overflow and NULL reference detection. Singularity is written in Sing♯, provides Software Isolated Processes (SIPs), and uses linear types to prevent shared-memory-related race conditions [30]. LISP and Haskell have also been used to construct an OS kernel [24, 26]. Ethos has a different goal than these OSs: it enforces object integrity without imposing a particular language or runtime.

SPIN allows applications to specialize the OS kernel using extensions [8, 25]. 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 [19]. This has been shown to provide performance benefits, for example by allowing applications to interact more directly with network interfaces [21].

One difficulty of requiring type safety in compiled programs system-wide is that it either (1) has an expensive run-time requirement or (2) requires that the compiler is part of the TCB. While compilers are typically large, certifying compilers avoid the above issues by relegating trust to a small certificate checker [41].

Fabric labels objects with their security requirements and provides dstributed-system-wide, language-based access and flow controls [35]. HiStar provides a simplified, low-level interface and implements mandatory information-flow constraints without a language dependency [49]; DStar builds on HiStar to provide information flow across hosts on a network [50]. Developments in HiStar/DStar and Ethos are complimentary: flow control will benefit from a higher semantic understanding of the types in a system.

6.2 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. Some schemes such as JSON do not directly support cyclic objects, but standard techniques exist to address these requirements [29]. 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**, types are encoded along with data.

RPCs are procedure calls that are executed in another address space [9]. Some RPC systems allow communication with remote computers and others—often for performance reasons—allow communication only within a single computer [7, 48]. RPC systems are used both in applications and microkernels. Microkernel RPC is particularly performance-sensitive, and so its performance is often highly optimized [33].

PL-specific Python [3], Java [23], C♯ [28], and C++'s Boost [2] provide examples of native serialization facilities. RPC systems in particular benefit from PL-specific serialization. The homogeneity of Java's Remote Method Invocation (RMI) [46] makes it easier to use than heterogeneous systems since it does not need a PL-agnostic Interface Description Language (IDL). PL-specific systems can also provide conveniences such as the ability to pass previously unknown objects such as subtypes to remote methods. This enables polymorphism (in other systems a subtype might be interpreted as its parent at the distant end).

PL-agnostic XDR [18], ASN.1, JSON [17], and Protocol Buffers [39] can serialize objects from any PL. Google designed the latter after experience demonstrated settling on one PL was infeasible.

ONC RPC builds on XDR [43]. The rpcgen utility generates client stub and server skeleton code from a high-level declaration, and programmers fill in the rest. CORBA provides inheritance and an any type [44], and identifies actual types using a type code. CORBA's type codes are ambiguous [16], but Etypes' are not. Ambiguity can violate the type safety of inheritance and any types. Thrift [42] is particularly flexible. Instead of dictating a single encoding format or transport protocol, Thrift exports an interface with which to implement both.

7 Conclusion and further work

An OS-wide type system can make it easier to develop and administer a robust distributed system. Etypes is similar to ONC RPC which is PL-agnostic, has an implicitly typed encoding, and generates code based on type descriptions. What makes Etypes unique is its use of the type hash as a UUID and its tight integration into the Ethos OS.

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

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.

Ethos performs type checking 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. 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.

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.

References

[1] Camel—a mail access library. http://live.gnome.org/Evolution/Camel.

[2] Boost documentation, serialization. http://www.boost.org/doc/libs/1_48_0/libs/serialization/, 2009.

[3] Python documentation, python object serialization. `http://docs.python.org/library/pickle.html`, 2010.

[4] AKRITIDIS, P. ET AL. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security* (2009).

[5] ARNELLOS, A. ET AL. A framework for the analysis of the reliability of digital signatures for secure e-commerce. *The electronic Journal for e-commerce Tools & Applications (eJETA)* (2005).

[6] BARHAM, P. ET AL. Xen and the art of virtualization. In *SOSP* (2003).

[7] BERSHAD, B. N. ET AL. Lightweight remote procedure call. *ACM Trans. Comput. Syst.* (1990).

[8] BERSHAD, B. N. ET AL. Extensibility, safety and performance in the SPIN operating system. In *SOSP* (1995).

[9] BIRRELL, A. D. AND NELSON, B. J. Implementing remote procedure calls. In *SOSP* (1983).

[10] BISHT, P. AND VENKATAKRISHNAN, V. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *DIMVA* (2008).

[11] BUCCAFURRI, F. ET AL. Fortifying the Dalí attack on digital signature. In *Proceedings of the 2nd International Conference on Security of Information and Networks* (2009).

[12] CARDELLI, L. Type systems. In *The Computer Science and Engineering Handbook*. 2004, ch. 97.

[13] CASTRO, M. ET AL. Securing software by enforcing data-flow integrity. In *OSDI* (2006).

[14] *Common Criteria for Information Technology Security Evaluation*, version 3.1 ed., 2009. `http://www.bsi.de/literat/doc/cc-pdf.zip`.

[15] COWAN, C. ET AL. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, 1998.

[16] CRAWLEY, S. C. AND DUDDY, K. R. Improving type-safety in CORBA. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing* (1998).

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

[18] EISLER, M. RFC 4506: XDR: External data representation standard, 2006. Status: INFORMATIONAL.

[19] ENGLER, D. R. ET AL. Exokernel: An operating system architecture for application-level resource management. In *SOSP* (1995).

[20] FISHER, K. AND GRUBER, R. Pads: a domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005).

[21] GANGER, G. R. ET AL. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst.* (2002).

[22] GOLM, M. ET AL. The JX operating system. In *USENIX ATC* (2002).

[23] GREANIER, T. M. Flatten your objects: Discover the secrets of the Java Serialization API. *JavaWorld* (2000).

[24] GREENBLATT, R. D. ET AL. A lisp machine. In *CAW '80: Proceedings of the fifth workshop on Computer architecture for non-numeric processing* (1980).

[25] GRIMM, R. AND BERSHAD, B. N. Separating access control policy, enforcement, and functionality in extensible systems. *TOCS* (2001).

[26] HALLGREN, T. ET AL. A principled approach to operating system construction in Haskell. In *Proceedings of the tenth ACM SIGPLAN International Conference on Functional Programming* (2005).

[27] HEISER, G. ET AL. What if you could actually trust your kernel? In *HotOS* (2011).

[28] HERICKO, M. ET AL. Object serialization analysis and comparison in java and .net. *SIGPLAN Not.* (2003).

[29] HERLIHY, M. P. AND LISKOV, B. A value transmission method for abstract data types. *ACM Trans. Program. Lang. Syst.* (1982).

[30] HUNT, G. C. AND LARUS, J. R. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.* (2007).

[31] IGARASHI, A. AND NAGIRA, H. Union types for object-oriented programming. In *Proceedings of the 2006 ACM symposium on Applied computing* (2006).

[32] JANA, S. AND SHMATIKOV, V. Abusing file processing in malware detectors for fun and profit. In *Proc. IEEE Symp. Security and Privacy* (2012).

[33] LIEDTKE, J. Improving IPC by kernel design. In *SOSP* (1993).

[34] LISKOV, B. AND SCHEIFLER, R. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst.* (1983).

[35] LIU, J. ET AL. Fabric: a platform for secure distributed computation and storage. In *SOSP* (2009).

[36] MILNER, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* (1978).

[37] PETULLO, W. M. AND SOLWORTH, J. A. Simple-to-use, secure-by-design networking in Xos. in submission, 2013.

[38] PETULLO, W. M. ET AL. Minimalt: Minimal-latency networking through better security. in submission, 2013.

[39] PIKE, R. ET AL. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming* (2005).

[40] PIKE, R. AND WEINBERGER, P. The hideous name. In *USENIX Summer 1985 Conference Proceedings* (1985).

[41] SCHNEIDER, F. B. ET AL. A language-based approach to security. In *Informatics - 10 Years Back. 10 Years Ahead.* (2001).

[42] SLEE, M. ET AL. Thrift: Scalable cross-language services implementation, 2007.

[43] THURLOW, R. RFC 5531: RPC: Remote procedure call protocol specification version 2, 2009. Status: INFORMATIONAL.

[44] VINOSKI, S. CORBA: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE* (1997).

[45] WADLER, P. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods* (1990).

[46] WALDO, J. Remote procedure calls and Java remote method invocation. *IEEE Concurrency* (1998).

[47] WANG, R. ET AL. 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* (2012).

[48] WEGIEL, M. AND KRINTZ, C. 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* (2010).

[49] ZELDOVICH, N. ET AL. Making information flow explicit in HiStar. In *SOSP* (2006).

[50] ZELDOVICH, N. ET AL. Securing distributed systems with information flow control. In *NSDI* (2008).