Configuration Pinocchio: The Lies Plainly Seen and the Quest to be a Real Discipline

Andre P. Masella
Ontario Institute for Cancer Research

Abstract

The construction of configuration files has long been considered outside of the domain of "programming". However, configuration files have a way of growing more complex. There is a struggle between keeping a configuration terse, by having the system infer information automatically, and explicit, without having excessive duplication. Either the configuration file develops embedded domain-specific programming languages or a text-based macro language is put in front. I will identify and categorize patterns in the evolution of these programming languages and describe what kinds of patterns are needed to avoid them.

1 Introduction

There has been a shift in the kinds of configurations written. Previously, a server would be purchased, configured to perform some role, and largely left alone for its serviceable life. In compute clusters, the pattern is very different. Virtual machines allow repeated creation and deployment of machine configurations. Compounding this problem, applications have sprawled into many tiers of services. It was previously common for a single service to be a single binary; now, it is likely to be several binaries, running on different virtual machines, and the logic of the program is also likely to be spread out into the database and peripheral services. This poses a problem for testing as it is often impractical for a developer to replicate the production environment on their desktop.

Due to these factors, configuration of an application has become much more complex. At the very least, configuration of an application extends beyond the application itself and must encompass all the peripheral services and the cluster environment–to say nothing of the build, test, and deployment logic. This started happening in the late 1990s with the advent of multi-tier applications and accelerated with the use of cloud-based virtual machines, particularly Amazon Elastic Compute Cloud in 2006.

As the layers of indirectly coupled components have increased, the complexity of configurations have increased to match. As a central problem, the composability of the configurations does not match the composability of the servers they describe. Moreover, the configurations of different servers operate in radically different ways, each developing unique methods of propagating default values and concisely expressing repetitive information.

To build better configuration management software, one must first understand the problem being solved. Configuration seems simple enough: simply provide some values to an application on start-up. Unfortunately, the reality is more nuanced. Firstly, the number of methods for informing an application of its configuration is larger than expected. These include command line arguments, configuration files, environment variables, and values in a database. Secondly, those values can contain simple data types, paths, composite data types, macro languages, and even programs written in Turing-complete languages that run inside the binary as part of its serving logic. A large number of these values also interact with each other in non-trivial ways; a change in one parameter can affect the interpretation of another. Finally, the output of the configuration is usually opaque to the programmer since the true output of the program is hidden inside the server. That is, there is part of the server that takes the configuration and interprets it, but there is generally no way to view the interpreted result. This interpretation depends on three conflated sources of information: the execution of the configuration (*e.g.*, the macro expansion), the defaults provided by the binary, and the invisible interface of the binary (*i.e.*, the parameters used by the server itself). It is concerning that the defaults provided by the server are not necessarily static themselves; they can be influenced by the server's build process.

Language-theoretic security researches have described

Table 1: Configurations analyzed.

Server	Function
Apache	Web server
Asterisk	Telephony server
BIND	DNS server
CUPS	Printing server
Make	Build system
NGINX	Web server
Samba	File sharing server

an exploitation path where any input data that can direct the flow of a program has the potential to *be* a program that exploits its host program. In this case, the host program is abstracted as a very unusual virtual machine, called a *weird machine*.[9, 3] Configuration is intimately tied to this concept as the server is a weird machine for the configuration and the configuration and the server taken together can be a weird machine for the queries. That is, a configuration can define new exploits in a server.

To determine the extent of the problem, a survey of common servers will be conducted, the problems categorized, and potential solutions discussed.

2 Survey of Configurations

In order to draw patterns, I analyzed the configurations of several common servers shown in Table 1. The goal is to look for patterns of three types: how defaults are propagated, how transformations are done on the configuration itself (*i.e.*, macro languages), and the kinds of programming languages that are embedded in the configuration to be used during serving (as distinct from macro languages which are only processed at configuration time). One of the major concerns with these embedded programming languages (EPL) is that they are underspecified; the semantics of their behavior is not laid out as cleanly as would be expected in a normal programming language. They also may have incomplete separation from the macro languages in the configuration itself.

2.1 Formats

At first glance, it would seem that that the complexity of the configuration is related to the structure of the configuration format. That is, there is a tacit assumption that INI configurations are semantically simpler than JSON ones; which is demonstrably false. Although the servers studied use their own formats, they share remarkably similar structure.

Most of the servers use hierarchical key-value stores; that is, something like an INI file, but the sections have implicitly nesting. In fact, the Windows registry is a hierarchical key-value store and can be serialized to INI. Samba's configuration format is exactly INI.[2] Asterisk stores data in a modified INI file; it has an extra layer of hierarchy by storing many separate files.[4] CUPS and Apache HTTPd look different from INI files, but this is only superficial. In both formats directives are effectively keys and the sections are nested into a hierarchy.[5, 1] Again, BIND and NGINX looks very different from INI files, but describe semantically similar content.[7, 8] All of these servers have an additional property: the order of directives matters. They all have access control lists (ACL) that can be specified by a collection of "allow" and "deny" directives, for which the order of the directives matters. Most of the other directives are order independent.

Make has the most different format. I wish to justify the inclusion of Make as a configuration format at all. Build configurations of all kinds tend to straddle the divide between configuration format and script. For the purposes of this discussion, a script has control over the execution flow; in traditional programming languages, the author of the program has control over the order in which pieces of the program execute–this is true even in functional programming where the language itself has more control of the real behavior of the program. In a configuration, this is not the case. Make provides an example of this: it is not possible to create a circular dependency in Make as the Make interpreter can detect and block it. A Makefile is really a declaration of a desired scheduling behavior of rule bodies that the Make interpreter executes. Make does have key-value pairs for variables, but the main focus of a Makefile is the build rules. A rule could be considered a key-value pair, with the body as the value and the sources and targets as the key, but the algorithm by which Make examines the composite keys would make that an inaccurate analogy. The rules are clearly separate configuration entities.

Both Make and Apache HTTPd also have macro rules embedded in the configuration.

Despite the simplicity of these configuration files, Asterisk, Apache HTTPd, BIND, and NGINX all describe Turing-complete EPLs.

2.2 Default Propagation

In any binary, the configuration elements must eventually be rendered to data structures or object graphs that direct the behavior of the running program. When describing these elements in the configuration, some information is elided. For example, each CUPS printer has a maximum page limit, `struct cupsd_printer_s.page_limit`, but the matching configuration directive, `PageLimit`, is not required. Therefore, CUPS must impute the elided

value by some method. This is what is meant by default propagation.

Default propagation has two modes: implicit, where the binary has rules that control how defaults are propagated, or explicit, where the programmer specifies how defaults are propagated. The ultimate value for a default must be present in the binary itself. Sometimes, that choice could be determined at compile time. In commodity software, this is a potentially unpleasant surprise for a user, since two identical configurations could produce differing behavior in two identical versions of the software depending on the options selected by the packager at build time.

Make has a very complex set of defaults, but ones that are extremely visible. Every copy of Make contains a default rule set that is compiled into the binary. A command-line flag can cause make to display this default configuration. GNU Make has some additional quirks: it will check for multiple Makefiles of differing names and capitalizations. If multiple files are found, it will use a composite of them.[6]

CUPS and Samba have the simplest model. For CUPS, the configuration parameters for a printer can be inherited from a printer class or, if not present, assume a default from the binary.[5] Here, the propagation mode is explicit because the user chooses a printer class for a printer. Printer classes cannot be nested, so the propagation is at most two steps. For Samba, the configuration parameters are inherited from the `global` section, or, if not present, assume a default from the binary.[2] Here, the propagation model is implicit, since the user cannot specify from where defaults are copied.

Asterisk has a more complicated model that is both explicit and implicit. Stanzas may be inherited by other stanzas and used to supply defaults. For instance, to have stanza *x* inherit from *y*, the first line of the stanza would be `[x](y)`. Some stanzas can be declared as templates; to be ignored in the configuration itself and only to serve as a default for other stanzas. To create such a stanza *x*, the first line would be `[x](!)`. Unfortunately, this is not the end of the algorithm. Some keys can also be inherited from the `general` stanza in a configuration. Not all attributes support inheritance; the binary defines which ones do.

Apache has an implicit model based on the structure of the nesting of configuration elements. Not all configurations elements may be nested in one another, but ones which can implicitly take the same values as their containers unless overridden. There is some relationship between the URL-space generated by the configuration. `Options` directives control the behaviour of specific parts of the URL space and are configured by `Directory`, `Location`, and `Files` stanza in the configuration file and `.htaccess` files in the directories being served.[1] NGINX has a similar, though simplified, model.[8]

BIND has a very complex model. It is worth noting that BIND has two configuration formats: one for the server itself and one for the DNS zones it is serving. In the zone format, which is much simpler, there is only one default, the TTL for a record, which is either specified for a record, or inherited from the `$TTL` directive, which is required. For the server configuration, the default propagation algorithm depends on the directive. For instance, both the `dialup` and `notify` options can be specified in the individual zones or the common `options` stanza. The `dialup` directive in the common stanza overrides the option set in the zone stanzas, while the reverse is true for `notify`; the common `notify` value is inherited by the zones unless they override it.[7]

2.3 Macro Languages and Embedded Programming Languages

I define a macro language to be a programming language that is embedded in the configuration and completes execution before a server beings processing user data. This is distinct from an EPL, which executes during processing of user data. A schematic example is shown in Figure 1. This distinction is necessary but not necessarily obvious when inspecting the languages in configuration files. The distinction is useful because it determines the possible forms the implementation might take, and the level of coupling with the rest of the server. The macros in a configuration can be run to completion on a configuration independently of starting a server, and the language interacts with the server unidirectionally and "by value". This is not the case for an EPL, which can be affected by or have direct access to user input during processing, making its interface to the server necessarily bidirectional. Some EPLs may also mutate the server's internal state, or themselves possess long-lived mutable state.[1]

The number of macro languages is surprisingly small as many of the things that appear to be macros are, in fact, EPLs. BIND posses a macro system in zone files. There are a number of preprocessing directives, including `$TTL` and `$ORIGIN`, which are meant to provide defaults for the remainder of the configuration. There is also the `$GENERATE` macro that allows construction of large blocks of similar resource records. The names of resource records are passed through a transformation to ensure they are suffixed with the correct domain and `@` is replaced with the current domain. All of these transformations can be done statically before BIND starts and produce a new zone file that is semantically equivalent to the original. BIND's configuration files have no macros of any kind.

Figure 1: A schematic showing the difference between a macro language and an embedded programming language. The diagram shows an example query flow through a system. The blue components are defined in the configuration, while the white components are from the binary. The database configuration defines multiple configurations of a database, but this does not control the query flow in a non-deterministic way. The logic below, also defined in the configuration, does introduce query-dependent logic, making this an embedded programming language.

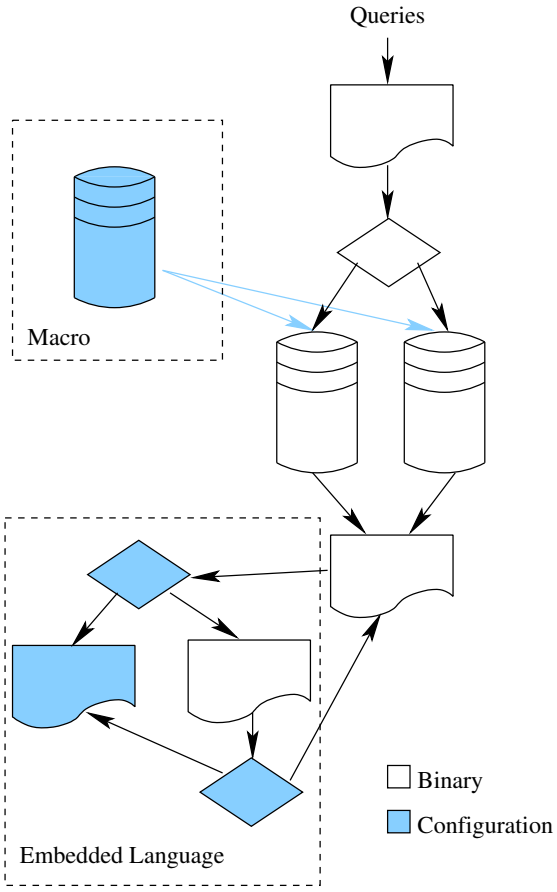

Figure 2: An accidental Apache infinite rewrite loop from Stack Overflow (`http://stackoverflow.com/questions/2635611`).

```
RewriteRule ^about/$       about.htm [L]
RewriteRule ^about\.htm$   about/    [R=302]
```

There are EPLs in Asterisk, Apache, BIND configuration file, Make, and NGINX. CUPS is the sole example studied that has no macro system or EPL.[2] Samba straddles this line as it does not have an embedded programming language, but it can call shell commands and create files using a well-defined string substitution mechanism.

NGINX has the simplest EPL used to perform URL rewriting. Each rewriting block can invoke the instruction `last`, which causes the URL rewriting to start again based on modified URL. This instruction is a jump instruction. Coupled with the conditions provided by other rules, NGINX's rewriting system is now a Turing-complete programming language that is executed for every incoming query. Since an unprivileged user of NGINX now has the opportunity to control the behavior of URL rewriting programs NGINX is executing, this qualifies as a weird machine.

BIND also has a query rewriting system know as a response policy. This systems is much more restricted and not Turing complete.

Apache's `mod_rewrite` has similar behavior to NGINX and can also be used in the same way. The regular expression matching can use back-references and the scope of a back-reference of a condition includes the subsequent rewrite rule. Apache's EPL is much more pervasive than this. Firstly, the URL remapping system extends beyond `mod_rewrite` and includes `mod_actions`, `mod_dir`, `mod_imagemap`, and `mod_negotiation`. Secondly, there is also the `mod_envsetif` system which allows examining the query to set binary flags that can alter the meaning of any part of the configuration. Apache's entire configuration is thereby transformed into one large weird machine, controllable through every query entering the system.[3] For example, Figure 2 shows a configuration from a Stack Overflow post where a user has unintentionally created an infinite loop.

Make's EPL is the most sophisticated and the most difficult to understand. Make supports eager and lazy evaluation during assignments, referred to as "immediate" and "deferred". Make's EPL also interacts with rule definitions which are a somewhat separate stage. To correctly analyse a Makefile, it is best to think of it in three stages. In the first stage, all instructions that do not contain one of Make special rule variables can be evaluated; these are `$<`, `$@`, `$^`, `$%`, `$?`, `$|`, and `$+`. Then rule targets and prerequisites can undergo wildcard (`%`) substitution. Finally,

Figure 3: An example Asterisk dial-plan for handling four-digit extensions.

```
exten => _ZXXX,1,Dial(SIP/${EXTEN}, 60)
exten => _ZXXX,n,Goto(in-${DIALSTATUS},1)
exten => _ZXXX,n,Hangup

exten => in-BUSY,1,Voicemail(210@default,u)
exten => in-BUSY,n,Hangup(17)
exten => in-CONGESTION,1,Hangup(3)
exten => in-CHANUNAVAIL,1,Voicemail(210@default,u)
exten => in-CHANUNAVAIL,n,Hangup(18)
exten => in-NOANSWER,1,Voicemail(210@default,u)
exten => in-NOANSWER,n,Hangup(16)
exten => _in-.,1,Hangup(16)
```

the bodies of the rules, which may contain the special rule variables, can be expanded.

Asterisk's EPL is called the dial-plan, found in `extensions.conf`, and it describes the logic that defines how a telephone call is routed based on the number dialed and source of the call. Since the `Goto` instruction performs variable expansion, to handle the status of a call attempt, the special variable `${DIALSTATUS}` is inserted into the `Goto` instruction so that the desired labelled block can be reached. Figure 3 shows a simple pattern to handle four-digit extensions.

This embedded language has become quite complicated, and some Asterisk developers have created a module that allows it to be replaced with the general-purpose embedded scripting language Lua.

3 Common Problems

All of the configurations surveyed share behavior that is undesirable to their users. As the maintainer of a system, one's desires are: ease of writing and adjustment of that configuration, ease of debugging, robustness of the configured software.

3.1 Terse versus Explicit – The Macro and Defaults Problem

As the person responsible for initial configuration of a program, there is a strong desire for the configuration to be as terse as possible. However, for the person debugging the configuration, it is desirable for the configuration to be as explicit as possible. Ultimately, these two goals are in direct opposition.

Both macro systems (and EPLs) and default propagation are attempting to solve the same problem: making the configuration more terse. Default propagation is a way of providing values the user does not know how to set, but this is equivalent to copying values out of the manual, making it only a means to achieve terseness. In the final in-memory representation of the configuration objects, many of the configuration values will be duplicated. Recall the example of CUPS's page limit, which is set for each printer known to the server. Default propagation can be seen a kind of macro system: it is a rewriting of the configuration before any user input is handled. In theory, one could separate the default propagation into a separate step. Some binaries, including CUPS and Samba provide options to "pretty print" their configurations, but what they are doing is providing a version where all the defaults have been propagated. This elaborated configuration file is now as explicit as possible, but much less terse.

BIND's `$GENERATE` directive is clearly a way to avoid writing very repetitive resource records into the zone file. This is important from a human perspective: a block of very repetitive records is likely to accumulate unnoticed errors. The terse form is more semantically relevant to a human than the explicit form, though the fully expanded form can be clearer.

The explicit form has two major advantages: it is stable and it comparable. If a default is changed in the binary, then it will be invisibly changed during upgrade if the configuration is terse. If the configuration is explicit, then all values have been stated in every required place, so the changed default is easily detectable. This makes the configuration more robust to upgrades. When the time for change has come, it also allows direct comparison of all the values. It will provide a clear and complete, though tedious, comparison of the changes made. If the configuration contains embedded programming languages, these cannot be compared, even if it is explicit, since comparing programs in Turing-complete languages is undecidable.

3.2 The Common Bridge Problem

Often, a configuration must represent disparate objects in a uniform way. The simplest example comes from the `fstab`, where each entry for a mount point contains a device, a target path, a file system type, some options, and `fsck` ordering.[11] Initially, this was fine, but it has become increasingly mismatched. For network file systems, the `fsck` ordering is completely ignored and the "device" is really a network identifier. As file systems become increasingly exotic (*e.g.*, FUSE, loopback, or distributed file systems), the lack of extensibility in the `fstab` becomes increasingly apparent. Ultimately, the real pattern is that there is one driver for each file system, part of which lives in the kernel and part of which lives in the `mount` program. The `fstab` is part of a communication channel between the two.

This pattern repeats itself many times. There are solutions that have been adopted. URIs are a simple ab-

straction over a very complicated configuration problem. CUPS uses URIs to identify printer backends. The abstraction is good because it allows CUPS to find the right backend and define a backend interface with the necessary complexity but forces the backend to solve the representation of its configuration space independently.

LDAP has also solved this problem by defining a uniform way to represent and query different kinds of data. The application has a well-defined interface to the LDAP server, but the user can provide a sophisticated request to the LDAP server that is proxied, but not understood, by the application.

3.3 Composition

In the traditional model where a service is set up on a machine once and left to run, composition seems like a pointless endeavor: what is there to compose? In the cluster model, composition is suddenly more important. There is a linked configuration between the server and the container and the servers in different containers need to contact each other in order for the tiers in an application stack is communicate. The configuration of the cluster system and the applications is now mutually dependent.

However, composition has always been important. Composition can be difficult to see because, when it is present, it fades into the background. Escaping is one of the most obvious and pervasive symptoms of poor composition. A user should not have to know the number of layers data will pass through to escape input data properly.

For good composition, LDAP is a worth-while example. LDAP has three key features that provide excellent composability: the interface with the program is well-defined, the interface with the user is well-defined, and the marshalling between the two is well-defined. The LDAP interface from the binary is straight forward: open a connection, send a query, get back a list of self-describing dictionaries. Similarly, the LDAP interface for the user is straight forward: provide a URI for a connection, provide a query, list the fields of interest to access from the results. The query passed is simply a string and, unlike SQL, it is a sufficiently well-behaved entity that the client can send its request to the server with a minimum of string bashing and rewriting.

For bad composition, any build tool will provide examples. The typical C compiler and tool-chain is a sufficiently obtuse build system that it needs Make to manage compilation of all but the simplest programs. Because of the division of header files and libraries, compile and link flags must be passed around separately. Although `pkg-config` has become the canonical source for flags, many libraries do not provide `pkg-config` definitions, so each program's build system must go through a platform-dependent set of steps to discover the correct flags to use a particular library. Determining the per-file build dependencies is also a non-trivial exercise and necessary to give Make the correct information to allow fast and accurate rebuilding of changed files. This lead to the development of tools like GNU AutoTools and CMake. In GNU AutoTools, the relay race is as follows: write a configuration script in two different languages (AutoMake and AutoConf's M4), which get translated to Make and a shell script, respectively; run the shell script so that it can detect the state of the build system and rewrite the Make file; then run the Makefile to invoke a shell to invoke the compiler with the correct flags. This is grossly simplified and ignores some of the other features of the programs involved. That being said, it is still exceedingly complicated and involves many layers of fragile escaping.

Demand for composition only increases as layers of automation increase.

3.4 Weird Machines

A weird machine is program whose input handling mechanism allows the input to use the program as an exploitable interface for running arbitrary code in a Turing-complete language. Imagine if a properly crafted `fstab` could turn `mount` into a virtual machine. Weird machines occur either by design or through bugs in input handling, often due to ill-designed attempts to parse or validate input[3]. Configurations are even more confusing because they offer the opportunity to create nested weird machines.

In most programming languages, the compiler is *not* a weird machine. That is, for any input file, it will read any input (source code) and produce output (object code or errors) in a deterministic way, and it will halt for all inputs. C++'s template are a Turing complete language, so it is fair to consider the C++ compiler as a weird machine. The process of expanding templates by the C++ compiler is itself the interpretation of a program written in a Turing-complete language: in particular, it is impossible to know whether the compilation of a given program will even halt. Having programs which may not halt would be entirely undesirable.

Any program might be a weird machine. For instance, imagine an application server that receives RPCs. Even if the behavior of the RPC is well defined, there's a possibility for a bug in the unmarshaling code that would allow an attacker to exploit the binary. In this example, the parser-validator of the RPC layer could be a weird machine.

In complicated configurations, there is the possibility for two levels of weird machines. At the first level, the

configuration parser-validator could be a weird machine controlled by the configuration data. The second level is created whenever EPLs in the configuration allow one to *define* a weird machine which is controlled by incoming requests. The first is less troublesome since anyone capable of creating a configuration is already trusted, but the second is extremely worrying.

Apache and NGINX's rewrite rules provide an example of this. The rewrite rules in these servers are Turing complete, and can be thought of as defining a bytecode that runs atop a virtual machine in Apache or NGINX. "Programs" written in this rewrite-rule bytecode *themselves* define a virtual machine, atop which every HTTP request is executed. If the virtual machine so defined has a Turing complete bytecode (encoded as HTTP requests) then a nested weird machine is formed that is remotely exploitable. Supposing that Apache or NGINX's rule rewriting engines are infallible, it is possible to write two different configurations: one which will allow a remote attacker to exploit the rewriting and one which will not. That difference exists because of the *configuration*.

Although embedded programming languages exist in the configuration, they are run-time entities that effectively define new servers. Stated another way: there is no material difference between creating rewrite rules in Apache's configuration versus writing them as C code a linking them into Apache.

Embedded programming languages are badly underspecified. They do not generally define formal semantics, describe a machine state, or explain all their side-effects. Even in Apache's documentation, there are simple gaps. For instance, a `RewriteRule` may access back-references from its regular expression match, but the behavior for accessing an undefined back-reference is not specified.

4 Recommendations

It is possible to make configurations more manageable. I recommend the following:

- separate the macro systems so that all macro processing is a separate step that happens before running the server.
- remove the weird machines by recognizing EPLs and replacing them with existing programming languages, treating them with the rigor of traditional programming languages, or demoting them to bytecodes, ideally non-Turing-complete ones, while delegating the high-level programming to the macro system.
- recognize the common bridge problem and solve it using established techniques.
- configurations that act as intermediates (*e.g.*, Docker), should always support raw strings. When embedding one configuration in another, escaping becomes difficult to do correctly and even more difficult for a human to read. It would be ideal if the configuration could read a user-specified number of bytes from the input with no escaping or translation of any kind.

For clarification, I am discouraging the use of Turing-complete languages at run-time (*i.e.*, on the path taken by queries) while encouraging the use of Turing-complete languages before start-up. The power of using a Turning-complete language is ideal for writing more terse, intelligent configurations, but allowing user-defined programs in Turing-complete languages to run during execution is risky, and often unnecessary.

4.1 The Macro Solution

Expansion of terse configurations into elaborate ones is a useful and desirable goal. The same arguments for having general-purpose languages that compile to machine code apply to having terse configurations that expand to general ones. Most of the languages used to expand configurations are text-based macro systems, yet text-based macro systems have been abandoned as a compiler for general-purpose languages in favor of compilers capable of analysis and verification of the program being compiled. One should have the same expectation of configurations. I propose using the term *configuration language* to describe a general-purpose language intended for creating configurations, as opposed to the executable code generated by general-purpose programming languages.

By creating a separate configuration language that processes a terse configuration file and generates an explicit one, there are several immediate benefits:

- the explicit configuration can be compared when changes are made. That is, it is diffable.
- default propagation can be made uniform, since it is implied by a standard language, rather than each binary.
- input handling in the application becomes simpler:
  - defaults can be moved entirely out of the binary.
  - the structure of the configuration can closely resemble the data structures in the binary.
  - the data is more likely to be representable as JSON, YAML, INI, or another standard format for which robust parsers already exist.

- libraries can be built for the configuration allowing code reuse.

- upgrading can be separated into two stages: upgrading the binary that reads a configuration file and upgrading the configuration language library that contains the defaults. For example, suppose there was a policy change for a default. Normally, upgrading the binary would change to the new default. If the configuration templates are independent, the configuration changes could be upgraded, then the new default run with the old binary, then the binary upgraded. Effectively, it becomes possible to backport configuration policy changes to old binaries.

- a small number of popular configuration languages can develop tool and debugger support, whereas this is impractical if each binary has a unique configuration format.

- configurations can be composed because the same language is used for the various binaries being composed.

4.1.1 The Failings of Traditional Programming Languages

There is an obvious question to be answered: what is the benefit of creating a new class of programming languages to solve this problem? Why not use existing languages that bring with them tools, libraries, and experience? The glib answer is that: if any traditional programming languages were good at being configurations, they would have achieved some wide-spread adoption for that purpose.

For the sake of argument, programming languages can be separated into two groups: functional and procedural. The difficult parts of configuration files are default propagation and composition.

Functional languages are very good at composition of data, but have no easy mechanism for default propagation. In most functional languages, all data needs to be passed explicitly as a parameter. This is cumbersome for configurations. It is possible to use function composition to achieve some of this, but it requires considerable rigidity in the format of the data, which is at odds with configurations where most values are defaults most of the time. This is partly because most functional languages use algebraic data types and the receiver must know the entire structure of a type to use it; it is not possible in most functional languages to create an updated version of a type with new fields and use it with existing code even if these new fields are unneeded or can be safely ignored by existing code.

Procedural languages, especially object-oriented ones, are good at composition of program flow and data. Default propagation is better: objects and initializers can propagate defaults. However, explicit control of data flow becomes cumbersome as the values propagated become sensitive to the order of execution since they can be mutated after propagation. One advantage to object-oriented procedural languages is that the receiver of an object can ignore new or uninteresting features of an object, unlike algebraic data types.

Both groups of languages have a heavy focus on data flow. The order of execution, or implied lack of execution, is a prominent feature in both groups. Functional languages often proclaim the order of execution is unimportant, but this is not strictly true when considering error handling. In most functional languages, a program stops at the first error discovered. For configurations, the order of execution is not important and it *is* reasonable to continue executing to discover more errors. Most functional languages still imply some linear order of execution, even if the compiler has control over that order, whereas configuration languages do not require this behavior. The configuration, in general, can be treated in parallel, and only certain operations will cause junctions in execution flow.

Input-output is also a focus of traditional languages and not of interest in configuration languages. Configuration languages will always be run in the same manner: take some collection of input files and produce an output configuration. There's no desire to have a long-running configuration language. Writing the next generation of servers is not a goal for a configuration language, so stateful file and network access is not needed. There will need to be some method to import sources of data, but this is a much more restricted case of the general input-output required in general-purpose programming languages.

A useful configuration language will inherit some of the behavior of functional and procedural languages, but, perhaps a more insightful focus, it can jettison large amount of unnecessary behavior from general-purpose programming languages.

4.1.2 Current Configuration Languages

Presently, there are a handful of configuration languages that attempt to solve some of the above problems. They provide uniform default propagation models and reusable ways to write terse configurations that become more elaborate. These languages are in their infancy, so this survey only attempts to draw attention to their prominent features. A summary of features is shown in Table 2.

Table 2: Comparison of configuration languages

	Coil	Flabbergast	HOCON	Jsonnet	NixOS	Pan	Pystachio
Paradigm	Functional	Functional	Imperative*	Functional	Functional	Imperative	Imperative
Side-effect Free	Yes	Yes	No	Yes	Yes	No	Hybrid*
Inheritance	Prototype	Prototype	Prototype	Prototype	None	Class-based	Class-based
Typing Strength	Weak	Strong	Weak	Strong	Strong	Strong	Strong
Typing Enforcement	Dynamic	Dynamic	Dynamic	Dynamic	Dynamic	Hybrid*	Dynamic
Schema Validation	None	None	None	None	None	Assignment	Request
Turing Complete	No	Yes	No	Yes	Yes	Yes	No
Scoping	Lexical	Dynamic	Lexical	Lexical	Lexical	Lexical	Hybrid*
Default Propagation	Inheritance	Scope, inheritance	Inheritance	Inheritance	Operator	Inheritance	Inheritance
Output Format	Python objects	Text, Custom	Java, Python, or Ruby objects	JSON	Java objects	JSON, XML	Python objects

* Depends on context. See description for details.

Coil: Coil defines a key-value hierarchy. Any object can inherit another object by way of an absolute or relative path. The inherited values, or values of child objects, can be overridden. Values can be strings with substitutions, integers, or lists with substitutions. String substitution allows templating using values defined in the hierarchy. Coil also has a `@map` operator to generate similar objects.
`https://code.google.com/p/coil/`

Flabbergast: Flabbergast constructs a key-value hierarchy similar to JSON. Each value is an expression that can reference the values of other keys using an unusual dynamic scoping method. Flabbergast allows prototype inheritance through "templates", which also function as lambdas with multiple return values. There are also map and reduce operations to manipulate objects. Flabbergast does not imply a particular output format; it allows the program to construct an arbitrary string which it writes to a file as output.
The author of this paper is the developer of Flabbergast.
`https://github.com/apmasell/flabbergast`

HOCON: Human-Optimized Config Object Notation (HOCON) is a superset of JSON that includes string substitution, prototype inheritance, and concatenation. HOCON allows strings and arrays to be concatenated using previously defined variables. The prototype inheritance is semantically similar to concatenation: an existing object is used, but the operation behaves as a replacement rather than an append. HOCON appears to have imperative semantics, but the language is analyzing the references between definitions and redefinitions, so that certain definitions are considered self-referential and illegal. The configuration is meant to be consumed by an application directly. The original version targeted the Java Virutal Machine, including Java, Scala, and Clojure, and it has been ported to Python and Ruby.
`https://github.com/typesafehub/config`

Jsonnet: Jsonnet looks very similar to JSON, but reintroduces some JavaScript-like features. Each value is an expression that can reference other values specifying a path of key names from the root of the tree to the desired value. Any object in the tree can be used as a prototype for another object. The standard library provides map and reduce operations to manipulate lists.
`http://google.github.io/jsonnet/doc/`

NixOS: NixOS constructs a key-value hierarchy. Each value is an expression that can reference the values of other keys referencing other values using a path of key names. The resolution starts in the current collection, or one of the capturing constructs, such as `lambda` or `let`, and checks through the nested constructs until it reaches the root. While NixOS does not have an inheritance mechanism, it does have a set of default propagation operator: the `or` operator is a null coalescence for a value, and the `//` operator is a null coalescence for all the values of a pair of collections.
`http://nixos.org/`

Pan: Pan defines classes for objects where fields can have very specific types, including range types and string matching expressions. Instances of types can be instantiated and mutated through a series of imperative operations. There is an output tuple-space containing the target configuration. Local variables created in the imperative language are in a separate name-space from the keys in the object hierarchy being constructed. The tuple space can be read from or written to using paths of key names; paths may be relative or absolute.
`http://www.quattor.org/`

Pystachio: Pystachio is a configuration data model and string template system built on top of Python. Any imperative features come from Python itself, rather than the Pystachio data model. `Structs` define classes for data object and the required types. Instantiated objects be associated with environments, that provide values for templates defined in `Structs`. Once all the templates variables are resolved, the resulting value can be type checked and used.
`https://github.com/wickman/pystachio`

4.2 Abstraction of the Common Bridge

The common bridge problem has been solved in many instances. Ultimately, solving the common bridge problem comes down to being appropriately agnostic of the data being handled.

In the `fstab` example, there are functions in the kernel capable of creating a mount point and user space needs a way to call those functions. That is essentially a remote procedure call, except the remote procedure is in kernel space rather than a remote system and the bridge is a system call rather than a TCP socket. The code in the user space must take a complex data representation, serialize it, and send it over the channel, where it must be deserialized and processed.

While any serialization method is fine, one which is human readable is more convenient. The format needs to be standard enough that the remote function is easily extracted. URIs excel at this for many applications. Again, looking to CUPS, each URI's scheme indicates to which backend (*i.e.*, function) CUPS should send the data. The remainder of the URI is entirely in the hands of the backend, and outside CUPS's boundary of competence. URIs also have well-defined format, parsing, and escaping semantics.

4.3 Composition

Providing good composition has already been partially address through configuration languages and the common bridge problem. Composition works well when:

- the interfaces between layers are well-defined.
- proxying is simple and does not involve fragile operations (this requires that the interface is well-defined so the intervening layer can know how to proxy).
- adding intervening layers does not change the proxying behaviour. It should be the case that if data needs to be proxied, the same proxying semantics should work irrespective of the number of layers. Escaping is an example of a failure here: a user should not have to know the number of layers data will pass through to escape input data properly.

The common bridge problem occurs when the interfaces between layers are defined incorrectly and the proxying is fragile. Configuration languages need to be good proxies and allow users to define interfaces between layers.

Escaping is one of the most obvious and pervasive symptoms of poor composition. The examples are everywhere: OpenSSH's `scp` does not escape certain file names properly, the escaping rules for strings in shell are extremely complicated and context dependent (*e.g.*, escaping for command line arguments is different from here-doc), and the great complexity of HTML escaping.

It would be ideal if all composing layers could handle proxying of strings with a length; such a model would allow a configuration language, which knows the length of a string, to pass it to a subsequent layer without the need to escape it. Specifying string lengths is a tedious prospect for humans, but trivial for software–much more than requiring the programmer to understand the nuances of escaping at every level. Modifying length-specified strings is non-trivial for human programmers, but so is modifying heavily escaped strings and modifying heavily escape strings is much harder to debug.

To some degree, creating well-defined interfaces will always be the domain of programmers, but configuration languages can provide better support to specify and communicate those interfaces. In the long term, configuration languages will have to develop both in-language features and conventions and styles for defining interfaces between layers.

4.4 Embedded Languages

Embedded programming languages are a bottomless well of concerns for security. As a general guide, these questions need to be answered, ordered from most desirable to least desirable:

1. Can this be replaced with something entirely static? (*e.g.*, a look-up table) Even if this table requires a program to generate it, it is better that it be done before the application runs.
2. Can this be replaced by a language which is not Turing-complete? (*e.g.*, a formula, or regular expression)
3. Can this be replaced by an existing scripting language?

Language-theoretic security researchers recommend stripping out Turing-completeness where possible to avoid creating weird machines. If it can be replace with something static, this is the best, and most trivial solution. I will elaborate on solutions in the other cases.

The first step should be to identify whether Turning completeness is required at all. It is recommended to try to reduce the problem to a regular language or deterministic context-free language. If this is possible, then the problem is a great deal simpler. If it is impossible, then the programming language should be considered carefully as if designing a general-purpose programming language. There are many obscure programming languages that few people know; when you create a new programming language, you can be guaranteed that no one will

know it.[10] If an embedded programming language is needed, why not use Guile, Lua, FORTH, GameMonkey script, AngelScript, TCL, Squirrel, or JavaScript? Any of these or similar languages has an easily-embedded runtime that allows plugging-in foreign functions to interface with the host program. All the issues of language design, validation, verification, type checking, the object model, documentation, optimization, and future development are externalized to a community of people focused on doing that well.

If using an existing language is impractical or provides too much power (*i.e.*, the program can be restricted to a non-Turing-complete language), a lot of thought should be put into the design of the language. Language-theoretic research has outline common pitfalls in input handling that create unintentional weird machine.[3] If there needs to be an input program, Turing complete or not, I recommend the following: minimize the language and plan the machine design to take advantage of existing knowledge in language and VM design. Minimizing the surface of the language also minimizes the potential for a weird machine. As a first step, define the machine on which the language operates. Is the machine going to have named registers (*i.e.*, variables) or be a stack machine? A stack machine is simpler to implement and simpler to write a verifier. What will the language look like? If the language looks more like assembly language or like a stack language (*e.g.*, PostScript or FORTH), the parser will be simpler to implement. What types are needed in the language? Obviously, fewer is better, as is avoiding mutable state. Next, define all the operations in the language and specify how they alter the machine state. Now, it is possible to implement this machine. As per standard practice, parsing the language and verifying it should be separate steps. Writing an efficient interpreter is something that should be avoided. If running on a virtual machine, such as the JVM or CLR, there are libraries that allow dynamic compilation and loading of bytecode; this dynamically generated bytecode can be optimised and in-lined by the JIT. If running on the native machine, LLVM provides a framework to compile and JIT and retrieve a function pointer to a dynamically compiled function.

This approach provides many secondary advantages. Suppose a runtime error is detected in the program; simply crash the virtual machine, dump the machine state to a log, and fail the incoming query. This output will be more than sufficient to correct the program and more comprehensible than an opaque stack trace of the interpreter. It is also possible to separate the compiler for this language from the binary and provide a simulator so that the programmer can debug their program using test queries, ideally ones that can be logged by the server. Also, by separating the implementation of each byte code, the entire language is now easily unit tested.

There is an argument to the utility of having Turing-complete languages on the serving path. However, it seems to be source of confusion. Apache in particular has made an effort to prevent this by providing a `LimitInternalRedirects` option which the manual states:

> `LimitInternalRecursion` prevents the server from crashing when entering an infinite loop of internal redirects or subrequests. Such loops are usually caused by misconfigurations.

Clearly, the *authors* of Apache consider the ability to perform Turing-complete redirection to be a misfeature as they cap the number of redirects at ten, by default.

Using either a custom language or an existing one, it is important to extricate the embedded programming language from the macro system in the configuration language. The more intertwined these two entities are, the harder each of them is to reason about. Using an existing language is a good way to avoid this problem: it becomes impractical to modify since the language is an externally-defined entity.

4.5 Conclusions

Configuration files are far more complicated than they first appear. Many configuration files define embedded programming languages that can alter the execution flow in a binary in Turing-complete, potentially exploitable ways. There are a variety of default propagation schemes used by different binaries. Currently, configurations are not easily composable. There is value in tools that provide a standard default propagation scheme and provide some level of composition. Some tools are being developed to fill this niche, but they are still fairly immature.

5 Acknowledgments

Thanks to Kyle W. Schaffrick for editing and assisting in classifying the patterns. Thanks to Dr. Gráinne Sheerin, Dr. Daniel G. Brown, and James L. Schofield for editing.

References

[1] THE APACHE SOFTWARE FOUNDATION. *Apache HTTP Server Version 2.4 Documentation*, Apr. 2015.

[2] AUER, K. *smb.conf Manual Page*, 4.0 ed., Feb. 2015.

[3] BRATUS, S., LOCASTO, M. E., PATTERSON, M. L., SASSAMAN, L., AND SHUBINA, A. Exploit programming: from buffer overflows to weird machines and theory of computation. In *USENIX ;login:* (2011).

[4] DAVENPORT, M. *Asterisk Wiki*. Digium, Inc.

[5] EASY SOFTWARE PRODUCTS. *The CUPS Software Administrators' Manual*, 1.1.21 ed., 2004.

[6] THE FREE SOFTWARE FOUNDATION. *The GNU Make Manual*, 0.73 ed., Sept. 2014.

[7] INTERNET SYSTEM CONSORTIUM. *BIND 9 Administrator Reference Manual*, 9.10.1 ed., Jan. 2015.

[8] NGINX, INC. *NGINX Documentation*, 1450 ed., Apr. 2015.

[9] PATTERSON, M. L. Cats and dogs living together: Langsec is also about usability. In *SEC-T* (2014).

[10] SCHAFFRICK, K. private communication, 2013.

[11] UTIL-LINUX. *fstab Manual Page*, Aug. 2010.

Notes

[1]Macro languages might be procedural in design and allow the declaration and use of mutable state, but this mutability does not escape into the server—configurations written in macro languages, taken in their entirety, are by definition referentially transparent. The inputs may include data from the environment or filesystem, however this data is assumed to be static for the duration of an individual execution.

[2]CUPS does have PostScript Printer Descriptions which contain PostScript programs that are sent to printers, but these do not directly alter the flow inside CUPS itself. From CUPS's perspective, they are magic strings it sends to hapless printers.

[3]Roland Illig has created a Towers of Hanoi solver in `!mod_rewrite!`available at `http://roland-illig.de/hanoi.mod_rewrite.html`