Provenance for configuration language security

James Cheney Paul Anderson
University of Edinburgh
jcheney@inf.ed.ac.uk, dcspaul@ed.ac.uk

Dimitrios Vytiniotis
Microsoft Research
dimitris@microsoft.com

1. Abstract

Declarative, high-level configuration languages (e.g. LCFG, Puppet, Chef) are widely used in industry to configure large system installations. Configurations are often composed from distributed source files managed by many different users within different system and organisational boundaries. Users may make changes whose consequences are not easy to understand, and such systems also currently lack mature security access controls; the few currently available techniques have idiosyncratic behaviour and offer no formal guarantees. In the worst case, misconfiguration can lead to costly system failures; because of the complexity of the configuration build, it is difficult to recover from failures, trace the source of the error or identify the responsible party. In this project, we will explore the application of *provenance* techniques (originally developed in the context of databases) to establishing well-founded and effective techniques for security and audit for configuration languages.

2. Background

2.1 Motivation

In a computing infrastructure, *system configuration* [3] is the task of assigning configuration parameters to all of the individual components so that the overall system behaves according to requirements. The *infrastructure* may be a datacentre, or a distributed cloud application, or any other composition of connected systems. Typically, there will be many thousands of parameters which control the behaviour of the individual systems, and the relationships between them. Managing these configurations manually is difficult because of the complexity of interactions between different systems, and errors can easily occur. Oppenheimer et al. [27] identify configuration errors as a major cause of large-scale system failures; a recent example is the serious, multi-day failure of Amazon EC2 in April 2011, resulting from a network misconfiguration.

To manage this complexity, special-purpose system-configuration languages have evolved, such as LCFG [2, 4] and Puppet [28]. These allow the configuration requirements to be specified in a relatively high level way, and the specifications are compiled down to generate the individual configuration parameters (see Figure 1). The system configuration task can then be thought of as "Programming the Infrastructure" [5], with the configuration parameters analogous to the machine code, and the configuration specifications analogous to a high-level program.

Configuration languages differ significantly from conventional programming languages such as Java, C++ or C#. Most modern configuration languages adopt a (more or less) declarative approach to specifying the desired configuration. The configuration tool is then responsible for generating and sequencing the actions necessary to transform the current configuration into the desired one. This decouples the description of the configuration from the process of deploying it, and the configuration language becomes purely a data description language.

Because of the complexity, manual configuration of large infrastructures has become largely impractical, and configuration tools are now ubiquitous. However, these tools and languages are in their infancy. Unlike the programming languages used in mission-critical applications, there has been very little attempt to formalise their semantics, or verify their implementation. This can lead to a substantial discrepancy between the rigour of an application, and that of the infrastructure on which it depends. For example, air-traffic control systems are now being configured using Puppet[1]; thus, bugs in Puppet (or configuration errors arising from complex features of Puppet) could lead to failure of the system as a whole even though the individual components have been subjected to extensive (and expensive) testing and formal verification.

We currently lack the ability to control and predict the impact of changes to the configuration, and particularly to audit changes to the configuration retroactively after a failure to understand how (and by whom) such changes were made. Large configurations are collaborative endeavours, involving many authors, and are often spread across different organisations. To encourage modularity and separation of concerns, configuration languages permit a hierarchy of classes with value-inheritance (rather than type inheritance). This allows a chain of users to successively specialise descriptions provided by others, but it can also lead to counterintuitive consequences or system failures when different users (possibly from different sites or organisations) make changes that affect the same part of the final configuration.

Our hypothesis is that work from *provenance* research, primarily techniques developed for understanding how query results depend on inputs or how databases evolve over time due to updates, can be used to improve security and audit capabilities for configuration languages. Accordingly, we now discuss prior work on configuration language security and on provenance, focusing on provenance in databases. We also identify a promising connection with work on *software contracts* and *blame*.

[1] FOSDEM 2011 (`http://lwn.net/Articles/428207/`)

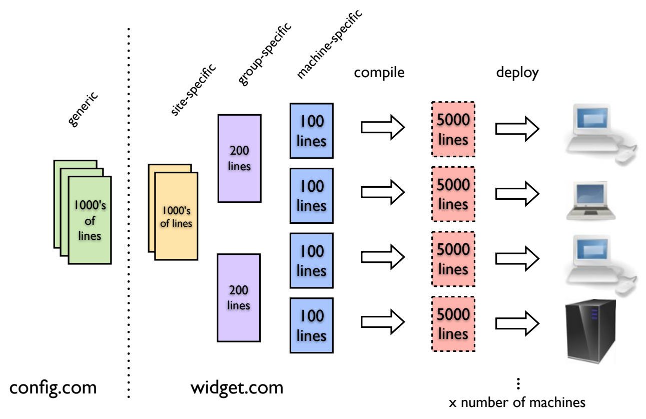
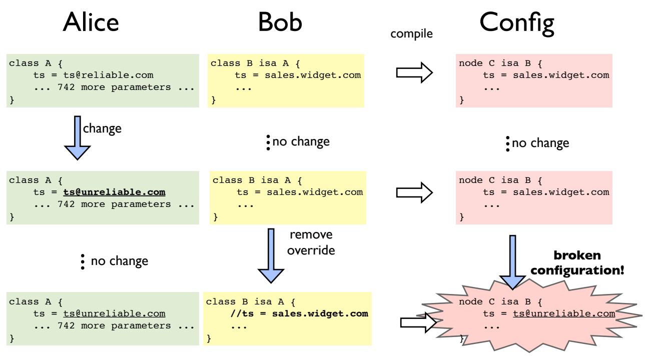


Figure 1. A typical configuration build process.



```
class A {
    ts = ts@reliable.com
    ... 742 more parameters ...
}
```

```
class B isa A {
    ts = sales.widget.com
    ...
}
```

```
node C isa B {
    ts = sales.widget.com
    ...
}
```

```
class A {
    ts = ts@unreliable.com
    ... 742 more parameters ...
}
```

```
class B isa A {
    ts = sales.widget.com
    ...
}
```

```
node C isa B {
    ts = sales.widget.com
    ...
}
```

```
class A {
    ts = ts@unreliable.com
    ... 742 more parameters ...
}
```

```
class B isa A {
    //ts = sales.widget.com
    ...
}
```

```
node C isa B {
    ts = ts@unreliable.com
    ...
}
```

Figure 2. Example of a delayed configuration failure due to inheritance masking a breaking change. Bob made the last change at step 3, but the reason for the failure is Alice's change at step 2.

2.2 Prior work

Access control for Puppet Vanbrabant et al. [29] proposed an access control mechanism in the context of Puppet [28], a popular, open-source configuration language. Their approach uses access control policies specified using XML path expressions. These policies are enforced by converting Puppet's abstract syntax to XML, determining which paths in the document are changed, and checking whether the policy allows the user who made the most recent changes to the source to make the changes.

This mechanism is clearly useful, but has several potential problems, because it checks only the final, compiled configuration, not the source files actually maintained by users, and attributes all of the changed values to the user initiating the change when determining whether to allow an update. In the presence of value inheritance or overriding, the results can be counterintuitive. For example (illustrated in Figure 2), suppose Alice, an external user, provides some default configurations which Bob, a sysadmin, subsequently uses. If Bob temporarily overrides the timeserver to a non-default value, for example to test a new configuration, any change made by Alice to the timeserver field will be masked by Bob's override and will not affect the final configuration. However, when Bob later reverts the change, the change originally added by Alice will be attributed to Bob, and will be allowed by Vanbrabant et al.'s access control mechanism; if the system fails as a result, Bob may be blamed improperly.

Aside from this, there is little work on security in configuration languages, or on their semantics. Some researchers, particularly Martin et al. [26], recently identified provenance as a potentially useful security control in its own right, and our recent workshop paper [6] gives more detail of the problems of security for configuration languages and identifies some future steps toward solutions, based upon adapting ideas from provenance in databases, which we elaborate upon in this proposal.

Provenance in databases Where-provenance [8, 11] is information linking data in the result of a query back to the source data from which it was copied. In contrast, why-provenance [11] and its generalisation how-provenance [22] summarise the input records that were needed to produce a given output record. Although these forms of provenance have been studied extensively for databases, they are somewhat tied to relational data models and query languages; there is not yet a clear general definition of why- or where-provenance for other data models or programming languages. In our work on dependency-provenance [13], we gave a semantic correctness criterion for a form of provenance that tracks dependencies on the input (inspired partly by work on non-interference in language-based security). Cheney et al. [14] survey the why, where and how provenance models and provide some formal comparisons.

Most work on provenance in databases has focused on explaining the results of queries by relating parts of the output to parts of the input. However, understanding the provenance of data that changes over time is also important, and sometimes more important; it is likely to be important for provenance and security in configuration languages, as well. Buneman et al. [8] made the first attempt to formalise and implement provenance-tracking for curated databases (databases that are manually edited over time), and subsequent work [10] defined a where-provenance semantics for both queries and updates in a nested relational data model. Ideas from this work should be applicable to configuration languages, which are also often based on nested record or list data structures; however, additional work will be needed to handle the distinctive features of configuration languages, such as object-oriented data inheritance and overriding, as well as language extensions such as user-defined functions.

Provenance and security Most work on provenance and security to date has considered applying standard security mechanisms, such as access control and cryptography, to keeping provenance information confidential or guaranteeing its integrity [18, 24]. There has been relatively little work on formal models for provenance security (or for correctness for provenance more generally), and further research is needed to understand what policies or correctness criteria are appropriate and how to ensure provenance security mechanisms really address the right problems. In the background, security and programming languages research has also begun to incorporate provenance-like techniques, in particular for auditing [25] and monitoring contracts [21].

Cirillo et al. [15] developed Tapido, a distributed object calculus that uses provenance to constrain delegation of rights. Provenance policies were considered among other security policies in SELinks [17], a dependently-typed extension of the Links programming language developed in Edinburgh [16]. SELinks was used to implement a secure wiki with provenance support, called SEWiki; more recently Links has also been used to develop the Database Wiki system, a flexible wiki-like Web application for collaboration on structured data resources. Database Wiki [9] provides built-in support for provenance, efficient archiving of past versions, and annotation of arbitrary parts of the database.

Our recent work [12] introduced a formal model for provenance security, along with definitions of security properties called *disclosure* and *obfuscation*. Disclosure states that some trace property of a system should *always* be obtainable from its provenance, whereas obfuscation states that users (or attackers) should *never* be able to infer some property of a trace from its provenance. Subsequently, we developed a *provenance core calculus* [1], a functional language equipped with a detailed tracing semantics, showed how to define other forms of provenance in terms of traces, and gave algorithms for slicing traces to enforce disclosure or obfuscation properties.

Audit, contracts and blame Another recent line of work has focused on checking and enforcement of security properties using increasingly powerful static type systems, dynamic monitoring, or a combination of static and dynamic methods. This work is connected to work on provenance in that it seeks to characterise what actually happened during execution in order to verify (either statically or dynamically) that security properties hold, or to determine who is responsible for a failure after the fact.

Vaughan et al. [30] introduced new formalisms for evidence-based audit. In Aura, an implementation of this approach [25], dependently-typed programs execute in the presence of a policy specified in authorisation logic, and whenever a restricted resource is requested, a proof is constructed at run time and stored in an audit log that can be inspected after the fact to identify reasons for granting access. In F7 [7], a log of assertions is used in the operational semantics to prove the soundness of its refinement type system (the assertions are erased at run time). F7 is used to prove security invariants statically, and has been applied in a number of other projects, including Guts et al.'s Auditability by Typing work [23], in which a custom logic is used to extend the F7 refinement type system with a policy that guarantees that any well-typed program logs enough information to convince a judge that certain program events happened in a given order.

Contracts (a generalisation of "assert" statements) have long been used in programming to check important properties at run-time and so ensure that any failures are detected early. Findler and Felleisen [21] introduced *higher-order contracts*, needed in higher-order programming languages such as Scheme or Haskell. The question of how to assign *blame* for run-time contract violations has since attracted a lot of interest, culminating in the recent development of a complete and correct semantics for assigning blame [19, 20]. This work draws in part on ideas from provenance-tracking, based on the idea that blame can be safely inferred from annotations that amount to a form of where-provenance. Although configuration languages differ significantly from higher-order programming languages, some ideas about correctness and completeness of blame should transfer to the configuration setting.

3. Research hypothesis and aims

Our research hypothesis is that *provenance management and programming language semantics techniques can provide increased security and auditability for configuration languages.* We will develop a formal model of one or more realistic configuration languages, since this is prerequisite to formalising and proving security guarantees. We will identify requirements (e.g. typical policies, audit queries, or blame policies) arising in practical configurations, constituting an expressiveness benchmark for the project. Finally, we will define security policies that capture the requirements, implement a prototype system that supports these policies, and verify its correctness.

Project summary This PhD project will develop well-founded and practical techniques for security and audit for configuration languages. This is likely to be challenging because the landscape of formal analyses or security for configuration languages is relatively unexplored, and research on provenance, audit and blame has so far focused on other paradigms such as databases, scientific workflows, or functional programming languages. The project will be rewarding because there is large potential for impact in large organisations and industry. There is an opportunity for a strong PhD student to do novel and significant work in this area that could have high impact.

While the proposed research focuses on using provenance to improve security for configuration languages, this work fits into a larger program of research on the role of provenance in language and system design. Language-based security has already benefited greatly from techniques such as effect systems, refinement types and dependent types; on the other hand, concepts such as contracts and blame have recently been related explicitly to provenance [19]. Thus, a broader aim of the project is to advance the study of language design to support principled provenance management alongside other concerns such as using contracts to improve reliability and identify parties responsible for problems.

4. Methods

Our primary research methods are *formalisation* and *programming language design*. By formalisation, we mean the use of mathematical models to explain and analyse the object of study. Here, we aim to study what people informally mean by provenance policies, including such concepts as trustworthiness, repeatability, error diagnosis, and explanation by defining mathematical models that can be used to compare proposed definitions and facilitate proofs of correctness of implementations. We will develop formal models that help explain the availability goals and security requirements for provenance.

The other primary methods we will employ are programming language semantics and language design techniques. We will focus on dynamic techniques such as instrumenting the system with additional provenance information, or allowing users to write contracts or audit policies that are checked at run-time. These techniques need to be formally verified to ensure that they guarantee good behaviour, and they may need profiling or experimental evaluation to ensure that their overhead is acceptable. Static analysis techniques may help improve performance.

The research will be evaluated by developing formal correctness criteria and proving correctness of proposed security mechanisms, and by validating the expressiveness of the possible policies on examples obtained from potential users. It may be possible to build on an existing system (such as LCFG or Puppet), making it easier to gather feedback from potential users, but this depends on the strengths of the student and on how easy it is to modify the existing systems.

References

[1] U. A. Acar, A. Ahmed, J. Cheney, and R. Perera. A core calculus for provenance. In *POST*, volume 7215 of *LNCS*, pages 410–429. Springer-Verlag, 2012.

[2] P. Anderson. Towards a high-level machine configuration system. In *LISA*, pages 19–26, Berkeley, CA, September 1994. Usenix.

[3] P. Anderson. *System Configuration*, volume 14 of *Short Topics in System Administration*. SAGE, 2006.

[4] P. Anderson. *LCFG: a Practical Tool for System Configuration*, volume 17 of *Short Topics in System Administration*. Usenix Association, 2008.

[5] P. Anderson. Programming the datacentre - challenges in system configuration. In *Microsoft Technical Report MSR-TR-2008-61 - The Rise and Rise of the Declarative Datacentre*, May 2008.

[6] P. Anderson and J. Cheney. Toward provenance-based security for configuration languages. In *TaPP*. USENIX, 2012. Online proceedings: `http://www.usenix.org/system/files/-conference/tapp12/tapp12-final15.pdf`.

[7] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2):8, 2011.

[8] P. Buneman, A. P. Chapman, and J. Cheney. Provenance management in curated databases. In *SIGMOD 2006*, pages 539–550, 2006.

[9] P. Buneman, J. Cheney, S. Lindley, and H. Müller. DBWiki: a structured wiki for curated data and collaborative data management. In *SIGMOD*, pages 1335–1338, 2011.

[10] P. Buneman, J. Cheney, and S. Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems*, 33(4):A28, November 2008.

[11] P. Buneman, S. Khanna, and W. Tan. Why and where: A characterization of data provenance. In *ICDT*, number 1973 in LNCS, pages 316–330. Springer, 2001.

[12] J. Cheney. A formal framework for provenance security. In *CSF*, pages 281–293. IEEE, 2011.

[13] J. Cheney, A. Ahmed, and U. A. Acar. Provenance as dependency analysis. *Mathematical Structures in Computer Science*, 21(6):1301–1337, 2011.

[14] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[15] A. Cirillo, R. Jagadeesan, C. Pitcher, and J. Riely. Tapido: Trust and authorization via provenance and integrity in distributed objects. In *ESOP*, pages 208–223, 2008.

[16] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: web programming without tiers. In *FMCO*, volume 4709 of *LNCS*, 2007.

[17] B. J. Corcoran, N. Swamy, and M. Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD*, 2009.

[18] S. B. Davidson, S. Khanna, T. Milo, D. Panigrahi, and S. Roy. Provenance views for module privacy. In *PODS*, pages 175–186, 2011.

[19] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *POPL*, pages 215–226, New York, NY, USA, 2011. ACM.

[20] C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitors for behavioral contracts. In *ESOP*, pages 214–233, 2012.

[21] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, pages 48–59, 2002.

[22] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40. ACM, 2007.

[23] N. Guts, C. Fournet, and F. Z. Nardelli. Reliable evidence: Auditability by typing. In *ESORICS*, pages 168–183, 2009.

[24] R. Hasan, R. Sion, and M. Winslett. Preventing history forgery with secure provenance. *Trans. Storage*, 5:12:1–12:43, December 2009.

[25] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: a programming language for authorization and audit. In *ICFP*, pages 27–38, 2008.

[26] A. Martin, J. Lyle, and C. Namilkuo. Provenance as a security control. In *TaPP*. USENIX, 2012. Online proceedings: `http://www.usenix.org/system/files/-conference/tapp12/tapp12-final17.pdf`.

[27] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[28] J. Turnbull. *Pulling Strings with Puppet: Configuration Management Made Easy*. Apress, September 2008.

[29] B. Vanbrabant, J. Peeraer, and W. Joosen. Fine-grained access control for the Puppet configuration language. In *LISA*, December 2011.

[30] J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. In *CSF*, pages 177–191, 2008.