Atlas: A New Way to Explore Software, Build Analysis Tools*

Tom Deering, Suresh Kothari
Dept. of Electrical and Computer Engineering
Iowa State University, Ames, Iowa, USA
{tdeering, kothari}@iastate.edu

Jeremias Sauceda, Jon Mathews
EnSoft Corp., Ames, Iowa, USA
{pi, jmathews}@ensoftcorp.com

ABSTRACT

Atlas is a new software analysis platform from EnSoft Corp. Atlas decouples the domain-specific analysis goal from its underlying mechanism by splitting analysis into two distinct phases. In the first phase, polynomial-time static analyzers index the software AST, building a rich graph database. In the second phase, users can explore the graph directly or run custom analysis scripts written using a convenient API. These features make Atlas ideal for both interaction and automation. In this paper, we describe the motivation, design, and use of Atlas. We present validation case studies, including the verification of safe synchronization of the Linux kernel, and the detection of malware in Android applications. Our ICSE 2014 demo explores the comprehension and malware detection use cases.

Video: *http://youtu.be/cZOWlJ-IO0k*

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering—*Design Tools and Techniques, Coding Tools and Techniques, Software/Program Verification*

General Terms

Theory, Verification

Keywords

Static analysis, Analysis platform, Human-in-the-loop

1. MOTIVATION

Today's software is growing larger and more complex at an alarming rate, but our cognitive abilities as practitioners are fixed. We increasingly rely on tooling to break through the comprehension threshold shown in Figure 1. Many tools remain isolated to academia [7], but Atlas aims to bring practical software analysis to the masses.

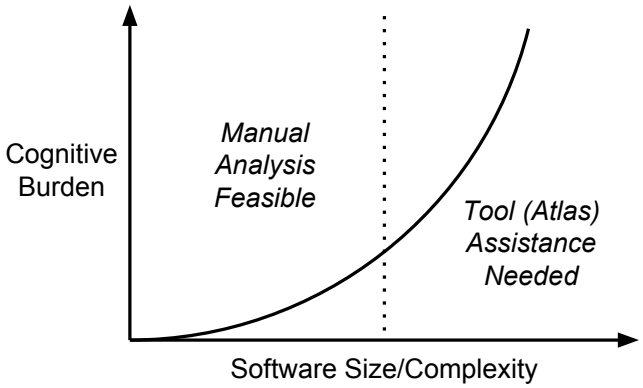


Figure 1: Atlas helps comprehend complex software.

Static analysis tools traditionally suffer from false positives due to overapproximation of program behaviors. Prior work seeks to address this with increasingly sensitive and expensive analyses. A common technique is to express analyses properties in terms of satisfiability (SAT), where basic blocks have been labeled with boolean reachability predicates, relying upon advancements in SAT solver scalability. Recent tools pursuing this approach include SATURN[2] and CALYSTO[4]. Reduction techniques such as Binary Decision Diagrams (BDD)[3] and Event Flow Graphs (EFG)[1] may allow the sizes of the necessary control flow graphs and boolean formulations to be greatly reduced; however, not every analysis query can be easily formulated as a SAT problem, and fully-sensitive analyses remain prohibitively expensive.

While it is possible to script a traditional static analysis using Atlas, we propose an alternative. Atlas breaks analysis into two phases. First, a set of conservative, polynomial time static analyzers translate the software AST into a graph database of precomputed artifacts and relationships. Then, a user can directly query and interact with that database to quickly discharge false positives and iteratively refine the results (see Section 3). This approach allows an analyst to supply critical invariants, insights, and software design knowledge which would otherwise be unavailable to a fully-automated approach. Software mining has been explored in the past in work such as CIA [6], GENOA [8], SCRUPLE [14], SCA [15], Software Bookshelf [11], GUPRO [10], Metanool [5], and GReQL [9]. However, to our knowledge,

*This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05...$15.00
http://dx.doi.org/10.1145/2591062.2591065

none of these tools offer the same ease of use or blend of automation and interactivity. In this way, Atlas seeks to be an intelligence amplifying system as proposed by Fred Brooks: "*If indeed our objective is to build computer systems that solve very challenging problems, my thesis is that IA > AI that is, that intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself.*" [13]

2. AUDIENCE & USE CASES

Atlas is useful for nearly any analysis task, but its fundamental use case is code comprehension. The precomputed relationships include sufficient material for building call graphs, data flow graphs, type hierarchies, dependency graphs, and many other useful results. A number of out-of-the-box scripts are provided for common queries. These lightweight analyses are invoked automatically in Eclipse Smart Views as the user clicks on code artifacts, providing instant feedback and interactive software graph visualizations. Figure 2 shows an example data flow graph produced when the user clicks on the field $DIGEST\_ALGORITHM$. It shows the field definition, plus that part of the data flow graph which is reachable via forward and reverse traversals on data flow relationships. The declaring control flow, method, class, package and project artifacts are shown automatically to provide visual context. In addition to Smart Views, Atlas provides an Interpreter View which allows the user to make on-the-fly queries using a provided API. These code comprehension features of Atlas are ideal for development, code reviews, bug reports, software audits, documentation, managerial review, and much more.

In addition to its out-of-the-box functionality, Atlas provides a rich API for writing custom analyzers. The library provides capabilities for selecting, traversing, and showing subgraphs from the larger software graph. Figure 3 shows a short Atlas script which would produce the data flow graph shown in Figure 2. The scope of possible analysis use cases is bounded only by the creativity of the user. For example, a script which performs global type inference, re-resolves dynamic dispatches, and modifies the graph database to reflect the results can be written with only a few hundred lines of Atlas code! We describe two real-world analysis applications, verifying safe synchronization in Linux and detecting malware in Android applications, in Sections 5 and 4, respectively.

```
Q field = fields("DIGEST_ALGORITHM");
Q df = edges(Edge.DATA_FLOW);
Q result = df.forward(field).union(df.reverse(field));
show(result);
```

Figure 3: An example Atlas script which would produce the graph shown in Figure 2.

3. DESIGN & ARCHITECTURE

Atlas is available today as a plugin for the popular Eclipse IDE. It is architected to achieve several design goals:

1. Provide sufficient material to solve difficult analysis problems.
2. Provide lightweight built-in analyses which scale to millions of lines of code.
3. Enable both automation and interaction.

To provide sufficient analysis materials, Atlas employs a number of polynomial time static analyzers to index an attributed, directed graph representation of the program's Abstract Syntax Tree. The nodes in the graph are software artifacts, and the edges are relationships between the artifacts. Each node or edge contains a set of tags and a map of attributes which contain additional information about that element (eg the local alias of the object instance in use). Sufficient raw material is provided to solve arbitrary analysis problems. Table 1 shows a high-level view of the artifact types and relationships that Atlas currently indexes.

Table 1: Artifacts and relationships in Atlas graphs.

Nodes	PROJECT, PACKAGE, CLASS, INTERFACE, ENUM, ANNOTATION, PARAMETER, FIELD, ENUMCONSTANT, DATA_FLOW, METHOD, CONTROL_FLOW, INVOKE
Edges	DECLARES, ANNOTATION, ELEMENTTYPE, OVERRIDES, PARAM, RETURNS, SUPERTYPE, THROWS_CLAUSE, TYPEOF, CALL, INVOKE, CAST, CATCH, CATCH, THROW, READ, WRITE

To achieve scalability, Atlas indexes all of the necessary raw material, but only pre-computes a first approximation of relationships. For example, when encountering a method invocation representing a dynamic dispatch, Atlas conservatively indexes CALL edges to all possible method targets. This first approximation allows Atlas to avoid doing the heavy lifting necessary for global type inference, keeps the indexing process fast, and may be good enough to satisfy most use cases. However, if the user really *does* want to do global type inference to tighten this approximation, he or she can do so by writing an Atlas script. This philosophy allows Atlas to index millions of lines of code in minutes and avoid unnecessary work.

Automation and interaction are provided via multiple Eclipse views. Atlas provides a "Smart View", which allows for immediate analysis results to be shown to the user in response to clicks on software elements. Atlas also provides an "Interpreter View", which allows the user to compose on-the-fly queries in a Scala interpreter environment and show results in an Eclipse graph editor. In both cases, the graph layout is customizable, and clicking on graph elements brings the user directly to the corresponding code locations. The Interpreter also allows the user to invoke automated scripts from a special project in the workspace, called a "toolbox". In Sections 4 and 5, we describe the use of toolbox projects to perform custom analyses.

4. MALWARE DETECTION STUDY

Analysis problems which require high-specific domain knowledge are a natural reason to extend the capabilities of Atlas. Consider the problem of detecting novel malicious behavior in Android applications, as in DARPA's Automated Program Analysis for Cybersecurity (APAC) program. Iowa State University is a performer on the project, using Atlas as the foundation of its approach. In order to detect malice, we must first define what it means for a behavior to be "malicious". Unfortunately, this depends on the purpose of the app. For example, it is expected for a navigation app to send

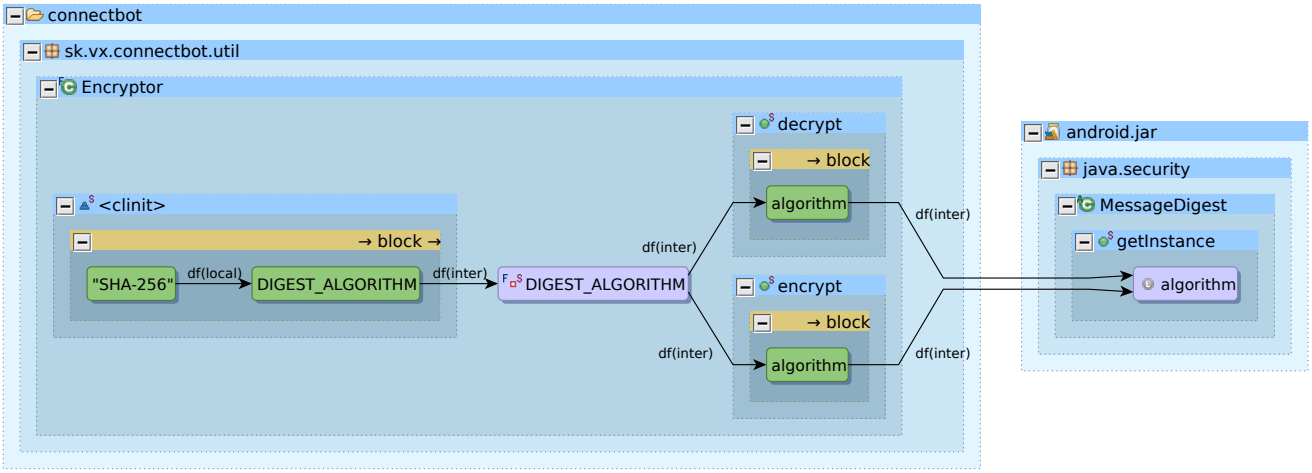


Figure 2: An Atlas graph showing flow of data to and from the DIGEST_ALGORITHM field.

the device location to the Internet, but that is unexpected for a podcast player.

Human judgment is required to determine the legitimacy of a behavior, so fully automated detection of malware is not possible. Instead, we seek to amplify the natural intelligence and intuitions of the human analyst by providing the "ISU Security Toolbox", an analysis suite built on top of Atlas. The Toolbox contains Atlas scripts which detect "smelly" software patterns, such as reflectively calling private library methods and dynamic code loading. We also provide parameterized detection scripts which locate violations of the well-known CIA (Confidentiality, Integrity, Availability) security model. The analyst pre-encodes his domain knowledge of what is security-relevant for a particular application, and the Toolbox runs the selected scripts and parameters in an automated way. After the results are aggregated, the analyst systematically reviews them. An overview of the approach is shown in Figure 4.

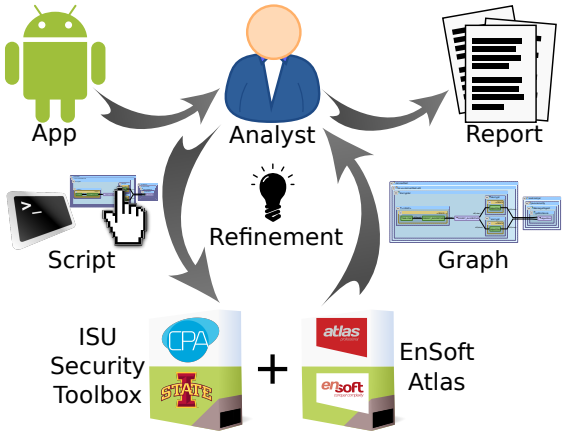


Figure 4: ISU detects novel malware using Atlas.

In the ISU approach, the human provides the creativity and insight, while Atlas performs the mechanical burden of finding the requested behaviors. As with synchronization verification, Atlas frees the ISU analyst to focus on the problem domain. Meanwhile, the analyst provides key insights and domain knowledge that Atlas, by itself, is lacking. In the first four APAC experiments, we analyzed a mix of 76 malicious and benign challenge applications provided by the adversarial challenge performers. To date, we correctly classified 90% of the apps, with an average analysis time of 2 hours. These results exceed those of any other APAC performer.

5. LINUX SYNCHRONIZATION STUDY

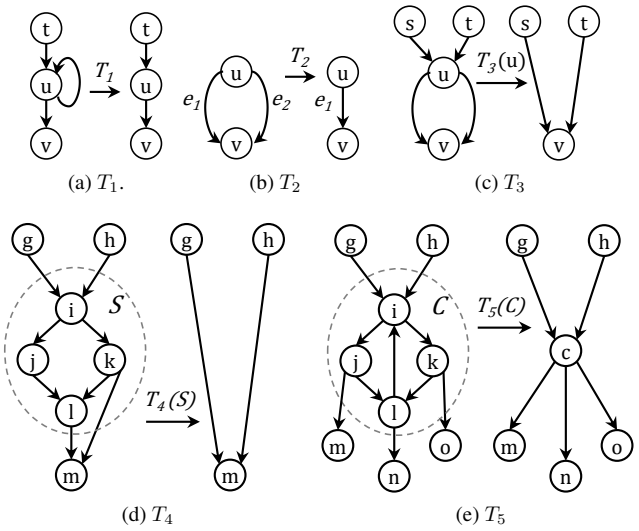
In Section 4 we described the semi-automated task of detecting novel malware, which begins by understanding an app's purpose. Consider a more concrete problem which lends itself more naturally to automation. Many important software properties can be modeled as two-event problems, wherein we wish to verify that event B follows event A on all possible execution paths. Examples include allocation and deallocation for memory management and locking and unlocking for safe synchronization. In our group's prior work, we employ a version of Atlas for the C programming language to verify the latter property in the Linux 2.6.31 kernel, where the two events correspond to calls to *mutex_lock()* and *mutex_unlock()*. Taking full advantage of the Atlas man + machine philosophy, we successfully demonstrate the correctness of synchronization in Linux. [12]

Let us refer to a locking event on mutex signature X as L(X), and the unlocking event as U(X). To show that U(X) follows L(X) on every execution path, we must demonstrate that all paths along which a violation may occur are infeasible. At first, the problem appears to be intractable due to the exponential number of execution paths. However, we observe that the number of ways in which U(X) may follow L(X) is limited:

1. U(X) follows L(X) within the same function.
2. Token X is passed on the stack as a parameter to the forward call graph, which performs U(X).
3. Token X is returned on the stack to the reverse call graph, which performs U(X).
4. Token X is written to a global variable and is later read to perform U(X).

The first scenario is trivial to check, the second and third are a bit more complex, and the fourth is the worst case. If Linux is well-designed, we hypothesize that scenarios 1-3 are the prevailing pattern. Our analysis begins by using Atlas to locate all L(X) and U(X) events in the kernel. We utilize a custom Atlas script to automatically discharge the simple scenarios from category 1. Next, we develop the notion of a Matching Pair Graph (MPG(X)). MPG(X) encapsulates categories 2-4 to provide the minimal subset of the kernel's call graph which must be considered to verify a locking scenario. We compute MPG(X) for all X automatically using an Atlas script. For a more about MPG(X), see [12].

We observe that the average MPG(X) in Linux 2.6.31 contains only 8 functions, and the majority of cases are, in fact, smaller. Only 3 of the 249 scenarios have a size above 50 functions. We also observe that MPG(X) reduces the size of the event RCG that must be considered by an average of 56%, with reductions as high as 99%. From MPG(X), the relevant interprocedural control flow graph can be automatically generated. In later work we refine this notion further by using Atlas to compute an Event Flow Graph (EFG), which discards irrelevant branch conditions of the CFG by retaining only the governing conditions which may affect the two-event property. To accomplish this, Atlas is used to perform a series of graph transformations to the CFG, shown in Figure 5. We find that the EFG is often 80-90% smaller than the original CFG. [1]



(a) T_1. (b) T_2 (c) T_3

(d) T_4 (e) T_5

Figure 5: Graph transformations used to reduce a CFG into an EFG.

The use of Atlas to verify safe synchronization in Linux is a perfect example of the potency of the man + machine philosophy. The automation features of Atlas can be used to discharge simple cases. For the remaining cases, automation can be used to compute MPG(X) and EFG. From an EFG, we are able to discharge the vast majority of potential synchronization scenarios in a fully-automated fashion. We are left with a tractible number of complex synchronization scenarios which require human insight to check. The quantity of original synchronization scenarios makes manual analysis intractable, while the complexity of the remaining few scenarios makes automated analysis intractible. Our man + machine hybrid system combines the strengths of both approaches to solve the problem in its entirety.

6. CONCLUSION

Atlas is a powerful new software analysis platform. It can be used out-of-the-box to facilitate rapid code comprehension, and it can be extended with toolbox projects to create custom analysis tools. We describe two such custom use cases in Sections 4 and 5, and the success results of each. Our ICSE 2014 demonstration will explore the code comprehension and malware analysis use cases. By embracing the Fred Brooks hypotheses [13], Atlas brings feasibility to difficult analysis problems which are resistant to manual effort or automation alone.

7. REFERENCES

[1] S. K. Ahmed Tamrawi, Gui Kang. Event Flow Graphs to Verify Absence of Vulnerabilities and Malicious Behaviors. *IEEE Trans. Softw. Eng.*, 2013.

[2] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins. An Overview of the Saturn Project. In *Proc. 7th ACM SIGPLAN-SIGSOFT Work. Progr. Anal. Softw. Tools Eng.*, PASTE '07, pages 43–48, New York, NY, USA, 2007. ACM.

[3] S. B. Akers. Binary Decision Diagrams. *Comput. IEEE Trans.*, C-27(6):509–516, June 1978.

[4] D. Babic. Exploiting structure for scalable software verification. 2008.

[5] A. Brühlmann and T. Gîrba. Enriching reverse engineering with annotations. *Model Driven Eng. ...*, 5301:660–674, 2008.

[6] Y. Chen. The C information abstraction system. *..., IEEE Trans.*, 16(3), 1990.

[7] J. Cordy. Comprehending reality-practical barriers to industrial adoption of software maintenance automation. *..., 2003. 11th IEEE Int. Work.*, 2003.

[8] P. Devanbu. GENOA: a customizable language-and front-end independent code analyzer. *Proc. 14th Int. Conf. ...*, pages 307–317, 1992.

[9] J. Ebert and D. Bildhauer. Reverse engineering using graph queries. *... Transform. Model. Eng.*, pages 335–362, 2010.

[10] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO - Generic Understanding of Programs An Overview. *Electron. Notes Theor. Comput. Sci.*, 72(2):47–56, Nov. 2002.

[11] S. Elliott Sim, C. Clarke, R. Holt, and a.M. Cox. Browsing and searching software architectures. *Proc. IEEE Int. Conf. Softw. Maint. - 1999 (ICSM'99). 'Software Maint. Bus. Chang. (Cat. No.99CB36360)*, pages 381–390, 1999.

[12] K. Gui. *Proving safety properties of software*. PhD thesis, Iowa State University, 2012.

[13] F. B. Jr. The computer scientist as toolsmith II. *Commun. ACM*, 39(3):61–68, 1996.

[14] S. Paul and A. Prakash. A framework for source code search using program patterns. *Softw. Eng. IEEE Trans. ...*, 1994.

[15] S. Paul and A. Prakash. A query algebra for program databases. *Softw. Eng. IEEE Trans. ...*, 1996.