

Controlled Declassification with Software Transactional Memory

Jan Vitek, Antonio Cunei, Suresh Jagannathan - Purdue University

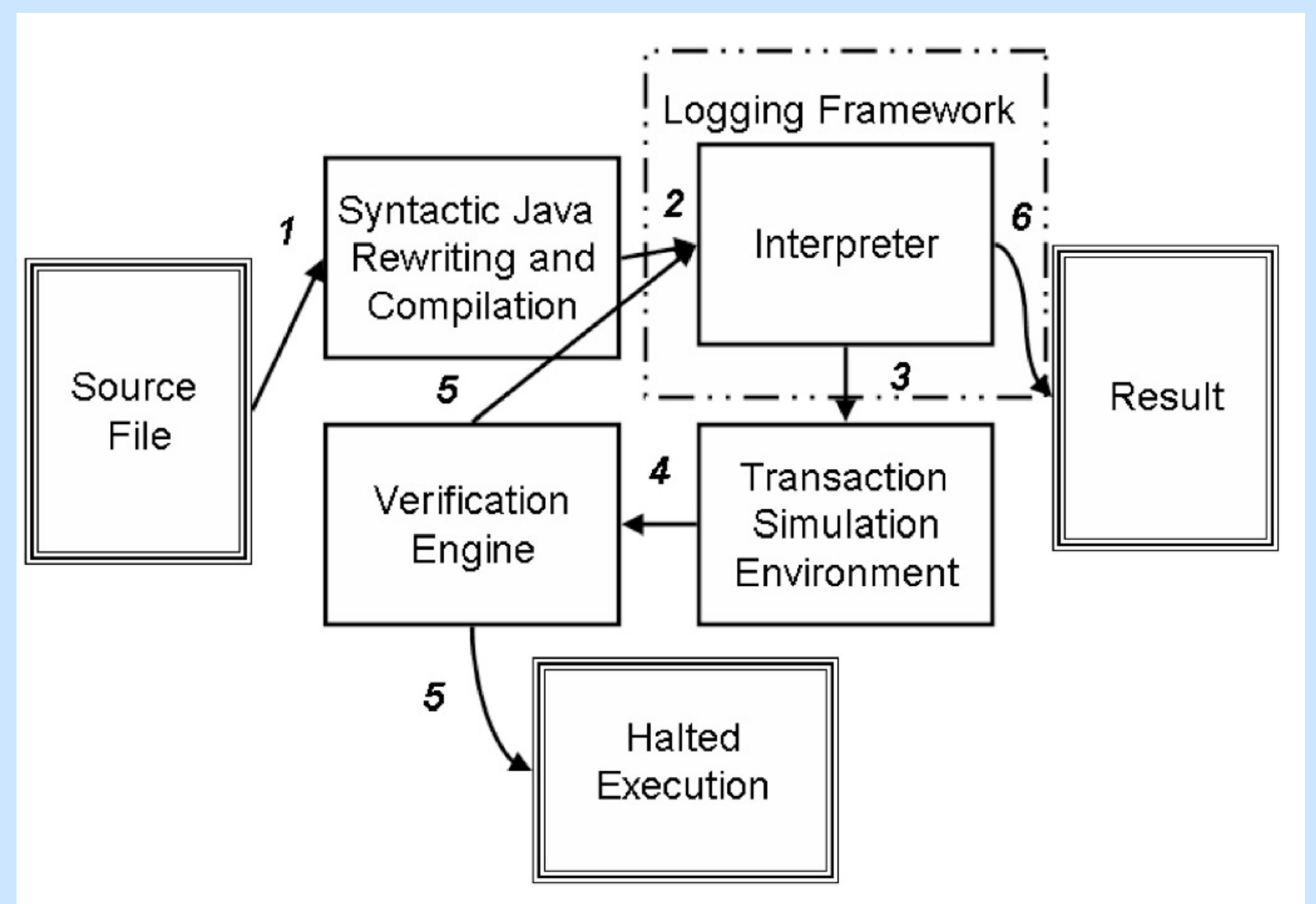


Preventing data leaks by means of transactions

We want to allow programs to handle classified information, while allowing selected parts of the data to become public in a controlled way: only those bits of information can become visible, not the rest.

Computer systems are complex: the possible data flows would suggest most programs have potentials for unauthorized data leaks. Just analyzing a program would lead to rejecting all but the most trivial code. We want a more flexible approach: run in an undo-able context and rewind if leaks would occur.

Safe information flow and declassification are critical problems to the cyber-infrastructure, homeland security, and commercial interests. Techniques that provide scalable, transparent, and effective solutions to this problem are of immediate benefit to current government and business initiatives and concerns.



Approach and Impact

New approach

- Transactions and randomization
- Detect secure data leaks
- Roll back if leaks detected

Research Impact

- Controlled declassification
- More general approach
- Enhanced security

Modern computer systems are frequently exposed to environments in which **malicious access is possible**; however, those systems must be functional, reliable, and **handle sensitive information securely**. Security breaches or accidental exposures of confidential data have a huge economic, legal, and personal impact. **Reducing or eliminating such threats** is therefore a concern that is destined to assume growing importance as modern society becomes increasingly wired and dependent on computing systems. It is **extremely difficult** to develop systems that exhibit the security that users and companies demand, as current techniques do not scale, are incompatible with modern programming languages, or impose unrealistic constraints on development. Information flow systems have historically tried to preserve secrecy by **stringent constraints on causal flows in a program**: if it is possible to infer that there is no causal relationship between operations and data of different security levels, the system can be deemed secure. Unfortunately it is well known that static information flow control is so stringent that it is **virtually impossible** to write non-trivial programs. More importantly, **realistic programs have to leak information**. When this leakage is intended, it is called **declassification**. However, determining whether the declassification of certain values can result in the unintended leakage of others is extremely difficult.

We exploit the **isolation and atomicity properties enjoyed by transactions**, implemented by the ability to perform rollback of monitored regions, **to enforce information flow security**. Transactional implementations log reads and writes to data, and roll-back execution state using these logged values, if a transaction commit fails. Transactions commit if there are no serializability violations with respect to logged data. For our purposes, **rollback occurs whenever a security leak is detected**: we detect and avoid data leaks before any observable effect can be exploited. Transactions are therefore used to encapsulate critical regions that either cannot be analyzed effectively statically, or declassify some set of confidential data. The use of transactions also ensures the approach is safe in a concurrent context: all committed transactions appear as if they are executed in some serial order. Our system operates as follows. High security data are annotated via Java language extensions. Operations that operate on declassified data are **executed within transactions**. Manifest declassification actions thus defines a declarative **description of allowed leakage** in a program. The transactional infrastructure keeps track of causal relationships among data items, and can thus record when there is unintended information flow. To do so, sensitive data (not intended to be declassified) accessed within a transaction is **randomized in a separate isolated run** (possibly several runs); if the non-secure values produced at the end of this run are different from the non-secure values produced initially, it implies that the values of sensitive data are **potentially visible** within a non-sensitive context. If a leak is discovered, the **transaction is aborted**, and data produced within it is discarded.