

# Avoiding Information Disclosure due to Software Faults

Eurosys 2010 PhD Workshop

Ioannis Papagiannis  
ip108@doc.ic.ac.uk  
Imperial College London  
Supervisor: Peter Pietzuch

## 1. ABSTRACT

No software system can claim that it does not contain any faults. In the same time, software systems accumulate personal information that must remain confidential despite these faults. Thorough software testing can never provide strict guarantees; instead we need systematic techniques of capturing and enforcing privacy concerns. In order to be widely adopted, these techniques must be easy to use and should not require fundamental changes to existing software.

## 2. MOTIVATION

Protection of sensitive user data is an important aspect of all but the most trivial software systems. Every software system that stores, manipulates or transmits such data must respect the privacy of their owners. Users want to be assured that their personal information is secure and that no unauthorised parties may gain access to it. The recent increase in the use of social networking sites was accompanied by a similar increase in public awareness: the slightest change in Facebook's privacy settings or terms of use generates enough controversy to capture even traditional media's attention. Often, respect for individual privacy is a constitutional right; system administrators face financial or civil penalties if they fail to comply. Stories of security bridges caused by software errors and resulted in credit card leaks or profile disclosures [5] are disturbingly common and have resulted in job suspensions, official inquiries and fines [4].

Software systems though, are written by developers that will never be able to produce 100% correct code: errors from their part will always potentially jeopardize users' privacy. In order to cope with this problem, developers thoroughly test their programs in order to gain confidence about their functional correctness. Since any developer may overlook potential interactions in a software system, test sets are never complete and usually get updated as new bugs appear. This approach, especially when followed thoroughly, has produced fairly secure systems. Unfortunately, this is not always enough: one careless developer or an unforeseen interaction over the lifetime of a system's development is all that is required for a severe data leak.

## 3. RESEARCH OVERVIEW

The research problem that I investigate is how to create a systematic approach of guaranteeing the privacy of user data independently of the functional correctness of a program. The overall aim is the development of a comprehensive,

fast, discretionary and easy to implement method that can initially capture high level data privacy requirements and later enforce them without assuming that each line of code in the system is correct.

To this goal, two main approaches exist: (1) automatic discovery of software errors that may result in information disclosure and (2) confinement and check of dangerous operations during execution so that inadvertent leaks are prevented. Automatic test generation, an example of the first approach, has provided rather limited results in securing existing systems [1]. Instead, examples of the second approach that isolate components inside applications and track all flows between them, have been more promising [2, 7]. My efforts thus far focus on the second approach: I try to provide for solutions that render the benefits of information confinement accessible to more developers. By focusing on the runtime level, fundamental redesign of system software is avoided, adoption of new operating systems is not required and developers are not tied to the proposed mechanisms.

## 4. EXISTING APPROACHES

**DIFC.** A promising technique to avoid data disclosure is the tracking of information as it flows inside a software system, known as *Decentralised Information Flow Control (DIFC)*. The main idea is to use program defined tags that, once attached to data and principals, will control how the data can be perceived and manipulated. For example, a diagnosis labeled with the tag "*private*" should only be able to flow to principals that have clearance to see private patient data (i.e. general doctors, nurses). DIFC was introduced in the context of Jif [3], a static approach that annotated Java variables with labels. Jif's compiler calculated permitted flows and rejected programs that potentially leak data, acting as an example of automatic error discovery. Jif's DIFC model resulted in a breed of DIFC compliant operating systems (Asbestos [2], HiStar [7]) that are examples of the confinement approach. Their main principle is to track the taint of processes as they observe labeled data and then limit all their potentially dangerous operations. For example, a Web Server can use a single tag for each user and label with it all the user's state. By preventing code that services this user to output state that belongs to other users, large scale information leaks are contained.

**Data Flow Assertions.** A very recent confinement approach is Resin [6]. Resin recognises that a solution based on OS redesign can be very hard to adopt; instead most existing privacy sensitive applications tend to execute on top of runtimes. Moreover it does not care for implicit channels:

security leaks are mostly a result of forgotten flow checks. With these two key observations, Resin introduced a Python and PHP interpreter that support policy objects, code that can be attached once to objects and then propagates transparently with them. When an object is about to leave the runtime, a method of the attached policy object is evaluated automatically and it either verifies or rejects the flow.

## 5. CURRENT RESEARCH

The results above highlight that the research problem can be tackled effectively. Unfortunately, both existing approaches require respectively either extensive changes to the operating system or to the runtime. My research focuses on how to broaden the applicability of these methods. DIFC methods can be improved by being implemented in runtime level so that adoption of new operating systems is not required. The Data Flow Assertions method should be applied without any changes to the runtime at all; Aspect Oriented Programming can be the tool towards this goal.

**DIFC for Event Driven Applications.** One category of software systems that require information flow guarantees is financial processing. Since automated trading algorithms greatly benefit from reduced latency, most exchanges worldwide provide collocation facilities. With space often being a scarce resource, only the most privileged traders have the ability to place machines near the exchange. An alternative is to use a single machine to collocate many traders and rely on DIFC to prevent trading data flow violations. Designing an implementation in Java required careful avoidance of shared state: once two threads get a reference to a shared mutable data structure no data flow can be tracked. Additionally, a new label model was created that can capture different privacy requirements on a single financial event. The resulting latency was lower than a similar trading system underlining the potential of the approach<sup>1</sup>.

**DIFC for Concurrent Applications.** An important issue with DIFC that leads to fundamental design changes is the shared memory paradigm commonly used for concurrency. When traditional threads are used as DIFC principals, the DIFC implementation must be able to track all communications between them. OS processes enable separation of address spaces but come at a substantial cost: in most cases it is infeasible to assign a separate a new process for each user of a system. Instead, the highly scalable concurrency model used by languages such as Erlang is an answer. The non existence of shared state facilitates both concurrent development in multicore CPUs and enforcement of isolation between application components. Similarly, asynchronous message passing as the only form of communication can be naturally combined with labels. Finally, tags, being numbers, can be efficiently copied without the serialization overhead that a Resin-style policy objects would require. These ideas were used to create DIFC enabled Erlang applications with promising results<sup>1</sup>.

## 6. FUTURE RESEARCH DIRECTIONS

**Zero Runtime Modifications.** Unfortunately, in order to enforce data flow checks, all proposed systems require changes to the runtime that prevent wider adoption. An approach that may result in the same kind of protection but

without the need for such changes can come from Aspect Oriented Programming (AOP). AOP enables the isolation of functions that capture cross cutting concerns from the main business logic. Then, and in order to enforce their corresponding concerns, these functions are transparently interposed at specific points in the code. In our domain, calls to potentially unsafe libraries (i.e. socket/file IO) can be intercepted dynamically and augmented with policy checks. The overhead of such an approach is an open question. On one hand, disabling checks on demand may result in better performance; on the other, AOP style code interception is not free especially when the interception points are frequent.

**DIFC Policy.** Even though DIFC operating systems have shown the solution's ability to provide for unintentional data protection, they have also underlined the complexity of tag and privilege allocations. The number of labels required, even for relatively simple applications, can easily overwhelm the developer. These errors may result in system policies different than the ones expected and can lead to violations of the intended information flow. Moreover, this intended flow is often unknown when the developer creates a component, only to be defined at deployment time. In other words, information flow policies should not be hard-encoded in the components but they should be decided on a per-context basis instead. For the above reasons, a higher level information flow policy language will improve usability and will allow less experienced users to use it effectively.

## 7. CONCLUSION

Large scale data leaks are often the result of developers failing to consistently track all possible information flows in software systems. A solution to this problem involves capturing the intended information flow in a few places in the code and then relying on the runtime to enforce it. Developers can gain confidence that even in presence of bugs or forgotten flow checks, the system will not invalidate the intended data flow. The two main solutions that have been presented in literature are DIFC and Data Flow Assertions. Both can be improved by focusing on the runtime level and avoiding extensive changes that prevent wider adoption.

## 8. REFERENCES

- [1] C. Cadar, D. Dunbar, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.
- [2] P. Efstathopoulos, M. Krohn, et al. Labels and event processes in the Asbestos Operating System. In *SOSP '05*, New York, NY, USA, 2005. ACM.
- [3] A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.
- [4] V. Prevelakis and D. Spinellis. The Athens affair. *IEEE Spectrum*, 2007.
- [5] Robert Westervelt. XSS bugs, information leakage top list of website vulnerabilities, 2009.
- [6] A. Yip, X. Wang, et al. Improving application security with data flow assertions. In *SOSP'09*, New York, NY, USA, 2009. ACM.
- [7] N. Zeldovich, E. Kohler, et al. Making information flow explicit in HiStar. In *OSDI '06*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

<sup>1</sup>The corresponding publications are under review.