# Secure Information Flow Using Compiler Techniques

Anna Thomas
*University of British Columbia*
*Vancouver, BC, Canada*
*annat@ece.ubc.ca*

***Abstract*—Protecting confidential data in computer systems is an actively researched problem with no complete solution. While access control and encryption prevent confidential information from being read or modified by unauthorized users, they do not regulate the information propagation after it has been released for execution. An approach proposed to handle this is secure information flow which has been one of the main areas of research with respect to security in programming languages. The applications of this is many-fold in computer security, for ensuring end-to-end confidentiality in file systems, building secure web applications, and secure code development. There are different approaches to ensuring integrity by preventing malicious interactions from low to high variables, and confidentiality by prohibiting leaks from highly secure variables to public outputs. They include type based systems, program analysis – static and dynamic analysis, and taint analysis. This paper is a survey of existing research work that leverages compiler techniques, and a critical review with suggested future research directions in the area of program analysis and tainting for secure flows.**

***Index Terms*—Information Flow, program analysis, dynamic taint analysis, security**

## I. Introduction

Information flow is the flow of information from one variable to another variable in a program. This happens in all programs since instructions are dependent on other instructions. However, in the security context, certain flows are not permitted, i.e., the flow from secret data to public output variables. Information flow control (IFC) is an important technique for discovering security leaks in software. The two main tasks of information flow control are

1) confidentiality : confidential data do not leak to public variables
2) integrity : critical computations can not be compromised or manipulated from outside, i.e., low variables cannot maliciously affect high variables.

The applications of secure information flow span various domains, for example, ensuring security in web applications using Javascript, building efficient intrusion detection systems [1] etc. The second task of information flow control to ensure integrity is analogous to solving some aspects of intrusion detection. Some of the terms and concepts about secure IFC to ensure confidentiality in information flow are described below using an example.

In the widely influential work by Denning [2], variables in information flow are divided into different security levels and this is viewed as a lattice with information flowing only upward in the lattice, i.e., from low to high. Variables in a program can be divided into high(H) and low(L) types based on the two security levels. The two main types of flow are explicit and implicit flow. Explicit flow is a direct flow from high to low security variables, while implicit is indirect and is control dependent on branches being taken or not taken. In the example in figure 1, the code `l1 = h1` is an explicit flow. The implicit flow is setting `l1` to `true` or `false` which is control dependent on the high variable `h2`.

```c
int l1,h1,h2;
bool l2;
l1 = h1;
if(h2 != correctpassword())
  l2 = false;
else
  l2 = true;
```

*Fig. 1:* Example of C code for implicit and explicit flows

Other forms of information leaks occurs in timing attacks or the power analysis attacks where the value of a secret variable can be inferred from the amount of time or power it takes to perform the task associated with it. For example, if some time consuming work is performed based on the condition that the value of `h1` is 1, then based on the time taken for the program to run, it can be inferred that the value of `h1` is 1.

An important concept with respect to security and information flow is non-interference. This policy states that an attacker should not be able to distinguish between two different computations based on their outputs, if the only difference is in their secret inputs. An example of this is entering the password in a website and the output of this action either states the password is wrong or authenticates the user. The current mechanism to reduce non-interference in this case is to specify that username or password is incorrect, thereby divulging lesser information regarding the high secret value, i.e., the password.

Since non-interference is a rather strict property for realistic programs, another concept called declassification was introduced. It is the controlled release of information based on dimensions like what information, when it will be released, who can access it and where it will be released [3]. A robust declassification prevents attackers from being able to manipulate the system by learning more than what is presented to a passive attacker.

Various methods for information flow control have been proposed and adopted, each of which have their advantages and disadvantages. Type systems and language design have been common methods to ensure secure flow [4], but they generally require non trivial effort from the programmer to adopt the new constructs. While these techniques are provably sound, they are conservative. They reject safe programs as ill-typed and the current research is in improving this false positive rate.

The second method is program analysis, both static and dynamic analyses of programs. Static analysis has lesser overhead compared to the latter, since the analysis is at compile time, but it tends to be more conservative, and when used alone, it mostly detects explicit flows. However, while dynamic analyses have been augmented with other techniques for implicit flow handling, it does not guarantee non-interference when used alone, since it considers a single execution of the program and does not consider all paths. Dynamic analysis tend to have lower false positives than static analysis since pointer and array references are resolved at runtime. The various methods employed is discussed in more detail in section V.

Another method employed is dynamic taint analysis [5], which is actually a form of dynamic analysis but, does not employ language based inference unlike other dynamic analyses. It can perform precise analysis of information flow based on an actual execution of program. It runs the program and observes which variables are affected by predefined taint values like user inputs.

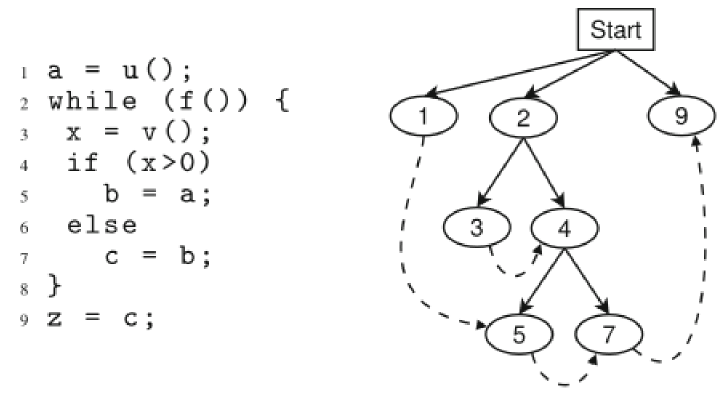
## II. Motivation and Challenges

There are many applications in security which require robust information flow control. There has been a lot of recent research work around secure flow in Javascript based web applications. It aims at preventing security vulnerabilities of input flow from untrusted user to security critical operations and preventing the leakage of confidential information.

There are many challenges involved in achieving secure information flow. Firstly, the techniques that have been proposed and adopted until now have some trade-off involved, either with respect to runtime overhead (dynamic analysis) or conservativeness (static analysis).

Secondly, it is a hard problem to quantify the channel capacity, i.e., information leakage in terms of the maximum amount that can be leaked [6]. There are different attack models possible and the quantitative definition of information leak depends on the capabilities of the attacker. For example, the strongest attacker can observe the value of *low* variable at each step during program execution while the weakest attacker can observe only a single *low* value at some stage in execution.

Thirdly, in languages like Java, there are many libraries that are imported. The source code for these libraries might not be available, and hence it is difficult to track information flow into or out of these libraries. We plan to focus on how current solutions handle different attack models and various kinds of attacks common to information flow such as timing attacks and covert channel attacks.



```
1 a = u();
2 while (f()) {
3  x = v();
4  if (x>0)
5    b = a;
6  else
7    c = b;
8 }
9 z = c;
```

*Fig. 2:* A sample program and the corresponding PDG (taken from [7])

## III. Background

Information flow Control (IFC) using program analysis – static and dynamic analysis, dynamic tainting, program slicing etc requires some knowledge of terms specific to compilers, and they are described below. Program analysis can be classified into three types:

1) Flow Sensitive – This analysis respects the program control flow and computes a separate solution for each program point.
2) Context Sensitive – This is an interprocedural analysis that considers calling context when analyzing the target of function call. This might not be known in advance in the case of static analysis when considering virtual functions.
3) Field Sensitive – When considering aggregate data structures, each instance of a field is modeled by a separate variable. This is in the context of pointer analysis, i.e., deciding where the pointer actually points to. This cannot be done precisely at static time.

Each analysis, static or dynamic, can be classified as flow, context or field sensitive. When the analysis is more sensitive, it is more precise, but usually at the cost of scalability.

The data structures used in analysis are usually different types of graphs. The program dependence graph (PDG) is widely used in static analysis. It models both data and control flow, between the nodes of the PDG which represent program statements. An example PDG for a sample program is shown in figure 2. The dataflow is represented by dashed edges, while control flow is represented using solid edges. Since it is derived at static time, it is more conservative and contains more edges than is actually present during execution.

In program slicing, the backward slice of a variable is all the variables or statements along the control flow path that affects the value of this variable. Similarly, the forward slice of a variable is all statements that this variable affects. In the example in figure 2, the backward slice of node 4 is {2, 3} and the forward slice is {5, 7, 9}.

## IV. Adversary Model

In this section, we consider the adversary model for the attack on secure information flow. Most of the research done using these techniques have some trade-offs and they do not explicitly state their adversary model, i.e., the strength and capabilities of the attacker. The research work surveyed in this paper would be critically evaluated against the three adversarial models described below, and the validity of these solutions under those models would be examined. We look into three kinds of models, a limited adversary with minimum capabilities, another adversary with higher capabilities, and a model which considers a very different attacker objective, i.e., using the program characteristics as a covert channel to transfer information among adversaries.

**Limited Adversary Model**: The first model considers a passive adversary, i.e., an observer who tries to guess the secret or high values based on information leaked or output available. However, the adversary does not inject malicious code into the application, and can neither modify nor mutate program variable values. The objective of the adversary is being able to gain the secret information or inputs, and hence affect confidentiality. These inputs can be used for other malicious purposes. The capabilities of the adversary before the attack include knowledge of the syntax and semantics of the language employed in the victim application. After the attack, the adversary can see the values of the low variables. However, in this model, the attacker can not see the values of low variables during different stages of execution.

**Broader Adversary Model**: This model considers an active adversary, one who can mutate or modify the low variables such that it can affect the high variables, during the attack. He has all the capabilities of the limited model, augmented with the above code injection or mutation. However, the adversary does not have the ability to mutate the high variables in the code, which are protected by other security techniques. He can also perform side channel attacks by gaining timing information during the attack. Also, he can observe the values of the low variables during each stage of execution.

Note that since security by obscurity is never a viable solution, and following the principle of open design, in both the models, we can assume that the adversary has access to the source code of the application. However, this is a probabilistic capability, i.e., the adversary may have the source code depending on the context. In some cases, like third party plugins or web applications, the source code might not be available even to the user [8]. Recent work in information flow has looked into both confidentiality and integrity, while the focus of past research has only been confidentiality. Solutions that handle both integrity and confidentiality prevent attacks by the broader adversarial model. We will be analyzing if the ideas proposed for secure information flow works under these two adversary models.

**Covert Adversary Model**: There are some cases where the adversary could have another objective, i.e., to communicate secret information between adversaries. This is done through a covert channel. It is important to note that this is different from the side channel attack where information is leaked through some parameters like power or time. Covert channels are hidden channels used to transfer information between two adversaries, using storage or timing channels, when they are actually not allowed to communicate by the security policy specified.

## V. Scope and Literature Survey

This section discusses the scope of the survey in terms of the methodology, the properties of IFC that are handled and the programming paradigm along with the corresponding references being surveyed.

### A. Methodology

We will be focusing on secure information flow using different techniques within program analysis, out of the three methods enumerated for enforcing secure information flow in section I. This encompasses the following areas:

1) static analysis using different approaches which include handling of explicit flows [9], and solving information flow control using program dependence graphs (PDG) [7]. PDG has long been used as a structure for program analysis to handle secure information flow control. Recent research work at Microsoft Research [10] has focused on applying static information flow policy to make privacy control on mobile devices more transparent. There is similar work done for handling information flow in iOS [8], but in this model, the static analysis is done at the binary code level.
2) dynamic analysis which handle implicit flows by considering control dependence [11], another method using control flow regions [12], and a novel method of using analysis on dynamic slices, i.e., slices during a single program execution [13]. Another interesting work [14] proposes a preliminary solution to tackling integrity using dynamic information flow tracking, and is one of the few works that focus on integrity as opposed to confidentiality.
3) program slicing along with static analysis, which refactor programs into secure and insecure slices [15]. This technique of leveraging program slicing generally performs better than static analysis alone since the entire program is not rejected in case of an insecure flow, but rather the program is transformed to eliminate such flows [16]. Another interesting work which uses static analysis along with dynamic slicing, and handles the third adversary model of covert attacks, is the work of Shaffer at al [17]. A recent work by Hammer et al [18] proposes a solution for identifying path conditions in Java programs, that can be used in finding the reason for a particular information flow, and hence is more precise in identifying specific instances of invalid information flow.

4) combination of static analysis and dynamic tainting to either reduce security leaks in web development languages like Javascript [19], or to reduce the overhead in dynamic tainting [20] and hence make it more suitable for secure flow handling.
5) combination of static and dynamic analyses, which is now used in secure flows in web applications [21] and specifically in information flow control in Javascript [22].

We would be looking into if these research works violate or reduce the effectiveness of the principles of secure design, namely by questioning the assumptions they make about their model, the simplicity of their design, i.e., the size of the trusted computing base, and whether they rely on security by obscurity. The final point is very relevant to information flow, since assuming that the adversary does not have access to some parts of the code, simplifies the problem of secure IFC, but does not adhere to the security principle.

Metrics that we would be critically reviewing include:

1) Scalability: There is usually no analysis done on the scalability of these techniques, for real world applications. While this paper [23] handles the quantitative nature of information leak by bounded model checking and tests the model on the Linux kernel, we plan to give a qualitative analysis based on intuition and reasoning for the scalability of the techniques used above.
2) Precision: Program analysis techniques are usually conservative approximations, i.e., they might generate false alarms but a security leak is usually not missed. We would analyze the false positives and also make sure that security leaks are not missed by the techniques considered. Precision is usually inversely proportional to scalability, higher precision algorithms usually do not scale well due to the complexity involved.
3) Practicability: how easy it is to employ these techniques in industry projects. While we do not plan to conduct any empirical studies on these techniques, this analysis will be based on the amount of work from the programmer's side, in terms of annotations and level of code understanding required.

### B. Information Flow Properties

There are properties of non-interference and declassification that have been formally defined in language based security for information flow, which we would not be focusing on. These properties can not be theoretically proven in program analysis given the false positive rate. This is essentially the trade-off with respect to language based or type system based IFC, which have the disadvantages of being tough to use practically.

### C. Programming Paradigm

Our survey would be focusing on single threaded programs at source code and bytecode level. We plan to study the nuances involved in the program analysis done at the bytecode level. As a starting point, we would look into how this is handled at the Java bytecode level [24]. There is research done for information flow in multithreaded programs which we would not be surveying since there are inherently different concepts that need to be taken care of. The programming paradigm would be imperative with some focus on secure flow in object oriented languages [25]. This work uses the program slicing technique to characterize information flow that pertains specifically to object oriented programs in Java. We would not be considering other paradigms like functional languages.

In the following sections, we will consider the solutions proposed using various high level techniques like static analysis, dynamic analysis, program slicing, tainting, and combinations of these techniques.

## VI. Static Techniques

Static analysis is one of the program analysis techniques that have been used for various applications like program understanding, debugging, finding critical regions of code etc.

It has also been widely used in information flow control. However, as with all other applications, static analysis tends to be conservative, i.e., there is higher number of false positives, but static analysis techniques would never miss an information flow leak that it is meant to protect against. The high rate of false positives is because static analysis tends to be conservative, since all information about a program will not be available at static time. However, static analysis tends to have a much lower overhead than dynamic analysis, which considers each execution, and hence is more precise. Moreover, the nature of static analysis prevents it from being used in systems where information flow policies are inherently configurable.

Static analysis has been used to handle explicit flow of information in programs. Different techniques employ program dependence graphs to handle IFC. There is ongoing research work to reduce the number of false positives and hence make the analysis more precise.

Liu et al [9] employed a static analysis technique called fragment analysis developed by Rountev [26]. The benefit of this technique is that it can consider software components, and hence can work on analysis of partial programs too. They developed this technique for Java programs and components. Their threat model assumes that the given Java classes in a Java component can be trusted, but the client code built on top of it cannot be trusted. Hence, the problem involves identifying invalid flows, i.e., flows of sensitive or secret variables from the trusted component into untrusted client code. This handles the problem of confidentiality in IFC. They handle software components or partial code by having a dummy main method as a placeholder that simulates the flow between client code and the trusted components. In this work, already builtin Java components and classes are part of the trusted computing base, and this is a valid assumption, if there are techniques to thwart injection of malicious code into these predefined Java libraries.

However, while the analysis is context sensitive, and does not require any annotations from the programmer, we believe it suffers from precision since the dummy main method which is added, serves to approximate all possible clients that can be built on top of the software component. With respect to scalability there might be issues since the underlying point-to analysis takes a lot of time even for small benchmarks like Gzip. Points-to analysis is a type of pointer analysis that tries to approximate what objects or variables a given reference field may point to. The authors show that their static analysis also handles integrity, i.e., flow of information from untrusted component to trusted component which might cause malicious manipulation of the software components. Moreover, it performs better, with cubic complexity, than some other techniques with type based solutions.

A novel technique that seems promising in terms of static analysis for IFC was developed by Hammer et al [7] which leverages path conditions in program dependence graphs (PDG) for IFC in Java programs. PDG has the advantage of being flow sensitive, and hence it is more accurate in terms of avoiding flows that can never occur at runtime. This analysis is flow, context and object sensitive. However, this solution will not be valid in the broader adversary model, since PDGs cannot protect against side channel attacks like timing or power leaks. The model has the advantage of taking care of leaks through the covert channel of uncaught exceptions. The context sensitive slicing takes care of maintaining security levels from the formal parameters to the calling function. In terms of scalability though, we think the path conditions will be a major bottleneck for large programs, containing legacy code and interactions between small trusted components and large untrusted third party code or libraries. The implementation was tested on medium sized programs in Java. This analysis will not work in other languages like C and C++ which require pointer analysis and makes the analysis infeasible or inaccurate. However, there is potential in extending PDGs to consider a general pointer analysis technique that would work for languages that use pointers.

Recently there has been ongoing research for secure information flow for third party applications installed in the mobile OS, with the advent of smartphones, Major players like Apple, Google and Microsoft have actively looked into solutions for their respective mobile operating systems and third party application framework. The solutions for detecting information flow leaks in such applications have ranged from dynamic information flow tracking in the case of TaintDroid for Google's Android OS, to static analysis in the case of iOS and Windows Phone. We now consider these two solutions for this problem, that employ static analysis techniques. It is important to note that research in this area focuses mainly on the confidentiality aspect of IFC, and does not handle integrity.

A recent work done by Xiao et al at Microsoft Research [10] employed static analysis to detect flows of private information of the users of Windows Phone, to the public web, via third party applications that the users install on their phones. The motivation behind this work is to enable users to transparently control their privacy and hence provide security to the users. The adversary in this model is the application developer of the third party application whose objective is to gain secret or classified information of the customers who download that application. Moreover, since this work is inherently different from other work done using compiler techniques to guarantee secure information flow, we delve into its details.

A key point to note here is that in this context, invalid or insecure information flow is defined as information that leaks to the web, *without explicit permission from the users*, and also flows that tamper with the information before presenting to the web. The latter is necessary, since if the information is tampered, the user might be deceived that the data flowing to the public is potentially harmless to the user. For example, some pixels in a photo that is uploaded by the user through the application, might encode the user's password on some website or other sensitive data, that the user would definitely not permit had he/she known of the leak. We think that this might be a conservative over-approximation, since some applications tampering with data posted to the public domain might be benign, i.e., they do not contain any secret variables. For example, applications that are low on budget or performance, have a higher variation in the peak signal-to-noise ratio (PSNR) when displaying the images – rather than displaying images in HD, and this technique might classify it as tampered data.

It is not considered an invalid flow, if the users allow the application to publish their classified information on the web (such as sharing on Facebook or geolocation in Twitter). The static analysis relies on the fact that the source code of the application is available, and is also viewable by the users who try to install the application. However, this might be specific to how Microsoft makes the third party applications available. Hence, this solution might not be generalizable to other mobile devices that leverage third party applications.

After the invalid flow is detected, there is a user dialog box presented, asking if the user is okay with this information leak when installing the particular third party application. This is analogous to how Facebook handles third party applications, where users are explicitly asked for permissions before installing the application. However, here the exact information leakage is also presented to the user, and hence the user can make a more informed decision. The language they consider is the TouchDevelop language, and it is transformed into a simpler model, where some values might have a mutable and an immutable part. The information flow characterization is done separately for each part for all those values.

The analysis is done on each basic block in the code, and the analysis also considers implicit flows. The results, i.e., how information flows from sources to sinks, are presented at the coarse granularity of the entire script, and at a finer granularity for each procedure. Moreover, this is one of the few papers in information flow analysis, that does a usability study on user experiences on using the new privacy control mechanism available. This is however necessary in this context, since there is a much wider range of customer base compared to the single programmer or security expert who usually handles information flow leakages. Moreover, the level of expertise between various members of the customer base, and the security expert might be completely different. This work does not protect against the covert adversary model, but works well for the limited adversary model as well as handling implicit flows. However, side channel attacks are not handled.

With respect to scalability, the analysis will not scale when considering large code bases, since the analysis is done even at very fine granularity of procedures. There might be much higher number of false positives, since the implicit flows considered statically, is a conservative analysis. This in turn might lead to poor user experience, with users not downloading some third party application due to false alarms about leakage.

The second major work with respect to static analysis is the PiOS [8]. This differs from the work done in Windows Phone, since the analysis is done at the binary code level, and the source code is unavailable. It attempts to identify all the information leaks from the third party applications. However, the solution is very specific to the mobile device OS architecture, like the Objective-C language, and the Mach-O binary format. It involves constructing the control flow graph (CFG) from the objective binaries, which is not very straightforward due to the intricacies involved in the programming constructs of Objective-C like the dynamic dispatch, handling class hierarchies, and taking care of external class dependencies.

The second step performs a reachability analysis to see if there exists potential paths in the CFG where the information flows from the source nodes to the sink nodes(output channels). Finally, dataflow analysis is done on the potential leaks found in the second stage, to improve precision.

We believe there might be more avenues to use static analysis for secure information flow, apart from the work discussed above. It is interesting to note how static analysis techniques have adapted to different models, like the absence of source code and handling properties of object oriented languages.

## VII. Dynamic Techniques

Dynamic analysis is also used in information flow control, and it is usually more precise than static analysis with respect to false positives, since it considers one actual execution. However, when used alone, they can not consider implicit flows, and hence can lead to some false negatives. This is because of the same property which is its advantage with respect to low false positives, i.e., it is based on one actual execution – ignoring other control paths can lead to missing implicit flows caused by omitting variable assignments. Hence, dynamic techniques are usually preceded by some sort of static analysis or program slicing technique. Current research focuses on the tradeoff involved in precision versus runtime overhead. While there is research done that focuses on just explicit flows, this is not surveyed in this paper, since such techniques are essentially imprecise by nature.

Bao et al [11] developed an interesting metric in dynamic analysis which aims at being precise, while still reducing the runtime overhead considerably. The metric is strict control dependency (SCD), instead of considering all control dependency or data dependency. An example of strict control dependency versus a normal or non-strict control dependency is shown in figure 3. In the case of non-strict control dependency,

```
int secret1, secret2, output_low1, output_low2;
if(secret1 == 10000) //Strict Control Dependency
  output_low1 = 1;

if(secret2 > 10000) //Non-Strict Control Dependency
  output_low2 = 1;
```

*Fig. 3:* Example of C code for strict versus non-strict control dependency

even when the information is leaked, there is much lesser damage done, unlike the case of strict control dependency.

The authors use static analysis to identify control dependencies that are strict. However, for higher precision in handling implicit flows, they develop runtime instrumentation for SCD, which has considerably lower overhead than all control or data dependencies. Moreover, when considering precision for lineage tracing, the false positives and negatives are much lower than that of data or control dependencies,

While the authors have explained its applications in lineage tracing and tainting, they are not very clear on how it could be used in information flow analysis. We believe this technique seems promising for dynamic information flow analysis (DIFA), but using this technique alone might lead to higher false negatives. This is because, when the attacker has higher capabilities, like in the case of the broader attack model, where all values of the low variable can be seen at all stages of execution, even a small amount of leakage can make the system vulnerable. In other words, *not all non-strict dependencies can be ignored.* In the context of the security lattice, and varying levels of security, it would make sense to quantify the non-strict levels, rather than having only two levels – strict and non-strict. With respect to scalability, it might not scale to large programs, especially since the runtime instrumentation uses stack to keep track of SCD. However, to bound stack growth, in case of nested SCD, only the nearest SCD to the statement in question is pushed into the stack. This solution is directly related to solving the case of implicit flows, or omission flows, where the omission of a variable definition is dependent on a branch condition. For example, the taken branch might have defined a particular variable, while the false condition of the branch would omit the variable definition. Hence, there is an implicit dependency between the branch predicate and the variable definition. There might be some potential reduction in these false negatives, if we can quantify how much information can be leaked, and use that value as a basis for detecting if non-strict dependencies should be included in the invalid flow list.

This is what has been handled in the work by McCamant et al [12], where they quantify dynamic information flow as network flow capacity. This is one of the very few works that mention the solution with an explicit adversary model, which considers the third model – the covert adversary model. The adversary's objective in this attack is to use the program as a covert channel, to communicate messages to another adversary using the program's secret input. Hence, the capability of the adversary is the knowledge of input distribution, the source code, semantics of the program, and the ability to use the program for this purpose.

They motivate the problem by stating that declassification cannot be avoided completely, and there will be flows from secure to public outputs. The idea is how can we quantify these flows to distinguish between harmless and harmful flows, and they do this by computing the maximum flow between inputs and outputs using static control flow regions. When the adversary compromises the application for using it as a covert channel, this solution bounds the information flow – both implicit and explicit flow – to the public output.

Masri et al [13] came up with a novel forward slicing algorithm during dynamic information flow analysis to detect and debug insecure information flows. The tool developed as part of this work can handle configurable information flow policies, and because it uses forward slicing, it is more useful for interactive debugging of detected invalid flows. It also supports unstructured control flow, in the form of break and return statements, but this is true of most dynamic analysis, since control flow information for such code is more precisely handled during runtime execution. The analysis requires a static analysis phase that computes the control flow graph.

This information is used in computing the direct dynamic control dependence and makes the algorithm more precise compared to other dynamic slicing algorithms. They also use a stack to store the predicates, within the dynamic scope, that have a direct influence on the secret statement in question. It can optionally detect implicit flows through a static analysis phase, that transforms the detected implicit flows into explicit ones. However, this might have higher chances of false positives. Moreover, this method would not work well with the broader adversary model we consider in this survey paper, since flows enabled by the adversary from low variables to high security variables – through possible mutations – are not handled in this technique.

Al-Saleh et al [14] research work focused on using dynamic information tracking for solving the second task of information flow control, namely integrity. As can be seen of all the work that has been surveyed until now, research on IFC has focused on solving the confidentiality task, but focus on integrity has been very less, i.e., solutions do not work on protecting against the broader adversary model, where the attacker is an active one. The authors come up with a preliminary way to tackle dynamic information flow tracking and make it applicable to intrusion detection systems. It is important to note that this work leverages tracking of dynamic information flow, rather than analysis of these flows. The tracking is done using tainting, and the metric used to quantify the dependencies is mutual information between sources and sinks. However, the taint tracking mechanism is developed in the architectural level, and hence the storage overhead associated with the tags is quite high. Moreover, as the authors admit, mutual information between sources and sinks could falsely quantify the actual information flow between them. Finally, the accuracy of the system has not been quantified with respect to false positives and false negatives. This makes it very questionable with respect to its application for IDS, where reducing the false alarm rate is a major requirement for a good IDS.

As an extension to the prior work done by Masri et al [13], they use the tool they built to enhance the capabilities of an application based IDS [27]. They use the concept of an attack signature to define some subset of flows that are explicitly not information leaks, but they are a subset of the flows that define the attack. These attack signatures can be identified by their tool. To support the detection of unknown attacks using anomaly based IDS, they use the information flow profiles along with cluster filtering, to identify the attacks. Cluster filtering clusters together the profiles that are similar, and this helps in reducing the false positive rate. They also suggest profiling the detected attacks offline, and seeing if they actually represent an attack. If it is so, recovery procedures are initiated, and that particular attack signature is added to the set of insecure information flow patterns.

However, like all IFC techniques that try to improve on intrusion detection techniques, the inherent issues with dynamic information flow analysis makes the solution not viable in IDS that have much stronger adversary models than even the broad adversary model we consider in this survey paper. We would need more advancement in dynamic information flow analysis to make it a viable solution for IDS.

## VIII. Program Slicing

Static analysis can be extended with program slicing to improve precision in terms of reducing the false positives. Similar to the case of intrusion detection, if the false positive rate is quite high, the technique would be rendered infeasible. This survey paper looks into how program slicing is leveraged to solve smaller sub problems within information flow analysis, like refactoring and transformation.

Cavadini [16] developed a technique called secure slicing that has the important advantage of being able to transform insecure code to secure code at static time, thereby avoiding another issue with static analysis, i.e., the rejection of whole programs because of small regions of code being insecure. However, this technique focuses only on the confidentiality aspect of secure IFC. An invalid flow is defined as flow of secure data information into public channels, without the explicit permission for declassification. The solution can be split into two phases – computing the program slice and performing program transformation on the secure slice obtained in the previous phase. Program slicing is done by computing the backward slice of a variable and checking if there is a flow from a variable marked as high or secret into a low output channel or variable.

However, due to refactoring in terms of program transformation, there is no strict guarantee that the semantics of the program would remain unchanged. Hence, we believe this technique cannot be used in critical legacy applications where more precise techniques combining dynamic analysis and tainting should be used. In terms of program transformation, they consider two different approaches:

1) instrument code with *skip* statements instead of insecure public output statements (this is similar to No-Ops in the architectural level)
2) Separate insecure flows into direct (data dependent flows) and indirect flows (control dependent flows), and place *skip* statements in sinks of indirect flows, while keeping some informational output or *skip* statements at the sink of direct flows. This makes sure that even if the adversary has access to the refactored source code of the program, he cannot figure out the values of the secret input variables. The informational output statements can be print statements that displays to the user that he cannot see the value of secret variables. This explains why informational outputs are placed only in the case of direct flows.

The technique also handles code that has been intentionally declassified, since the technique requires the previous phase to specify the set of invalid flows (through the PDG). The author claims the scalability of the technique in Java. However, annotation is required for other languages which do not have tools for handling dependence analysis. The annotation required would include specifying the high variables and variables that can be declassified. We cannot say whether this technique would actually scale to real world applications since it was tested on case studies like double-blind peer-review and another case of intentional declassification.

Smith and Tober [15] proposed a technique to refactor code into high and low security components using program slicing. The technique consists of three stages, namely identifying sensitive or secret code, refactoring them into high security component and adding declassification statements for handling explicit declassification from the high component into the low component part of the program. The first stage is handled by a static program slicer that considers an IO centric approach, i.e., IO channels have to be initialized to either belonging to the high or low category. The technique however does not consider integrity since only the forward slice of a high security input is considered. In other words, the information flow or backward slice of low security inputs are not handled. Another disadvantage of this technique is that it requires the programmer to explicitly handle declassification after the code has been factored into high and low slices. Declassification is a known hard problem, especially since it requires careful analysis and understanding of the code. This could also lead to possible leakage of information, if the declassification is not handled correctly. The authors specify some principles to help in correct declassification, but as we know, relying on programmers to declassify, might lead to more vulnerabilities, than already existing ones.

At the end of the refactoring phase, the high security component releases only high output information, and the declassification to the low security component is done through public access methods. Annotations required from the programmer include properly labeling data at the input points, placing public methods for declassification, and placing checks to make sure that high outputs are only within the high component. The primary issue with refactoring code automatically is that it would result in code that would be more difficult to understand or maintain. It is obvious that as code becomes more complex, the chances of vulnerability increases. One advantage of this technique is that it handles possible information leaks through covert channels like debug or log output channels, which might end up inadvertently revealing enough information about the sensitive inputs. This solution of refactoring will not scale when considering more than two security levels, i.e., the different levels according to the security lattice. In terms of general scalability when considering only the two levels of high and low, the technique will not work well, since it requires a lot of annotation and careful understanding of code.

In terms of precision, these dynamic techniques would be more precise, since it prunes away more invalid flows that program slices, which are an extension over the static analysis. With respect to the adversary models, both the techniques would definitely fail against the broader adversary model, since both techniques do not consider integrity. Moreover, the second technique's threat model assumes that the IO channels are secure, which does not work against the adversary who uses the IO channel as a covert channel. It can be inferred that both the techniques would need advancements in automating the handling of declassification, which is a hard problem.

Shaffer at al [17] proposed a solution using static analysis and dynamic program slicing, to handle information flow leakage as used by the covert adversary model. The solution handles both covert channel and overt flaw. An overt flaw occurs either by a direct data dependence or an implicit control dependence, which transfers the value from a high variable to a low variable. The implicit flow is handled using dynamic slicing techniques.

The covert channels they specifically consider are the timing and the storage channels. The point of interference of a system, is some internal resource that is viewed by an adversary for two purposes:

1) knowledge of whether the second adversary has communicated some information
2) the information that the second adversary has communicated

A program is transformed manually into a base program, which is an abstraction of the original program and is written in a domain specific language called Implementation Modeling Language (IML). Apart from the fact that this raises huge concerns on scalability, there is also the issue of accuracy, since as the authors mention, there might be some covert channels that are lost in the transformation.

An invariant model that should be satisfied by the program under all conditions should also be manually stated by the programmer. The invariant model and the base program is compiled into a language called the Alloy Specification language, and the representation formed is called the domain model. It is used in formulating the dynamic slice. The dynamic slice is limited by the scope variable, that says upto which variable, the slicing needs to be done. The slicing done is a dynamic backward slice.

Also, the domain model contains the implementation model, which specifies how the timing and storage covert channel attacks can be caught. For example, if a variable definition occurs before another variable definition, this is captured using a system clock and a keyword called `before`, which can be used to check which definition occurs first.

This is different from the side channel timing attack, since the timing attack is used to get a high variable value which has leaked inadvertently due to some timing differences. Similarly, using an example of the storage covert channel attack, they show how the specifications can be written to avoid information transfer using a common file, that is marked full by one adversary, and the other adversary observes the value of the variable `full` to figure out what the first adversary was communicating.

However, apart from the obvious issue of scalability, another problem with this technique is that the onus of understanding the covert channel is placed upon the programmer or security expert, after the framework has detected a covert channel. It can be inferred that as the covert channel is used to transfer more bits, for example through looping constructs, the complexity of the checks to capture these covert channels also increases. Moreover, it does not handle the integrity of the system, namely it does not shield the system against malicious flows from low variable to high variables. We believe this technique might be one step forward in the direction towards formalizing solutions that handle the covert adversary model. However, there still exists a lot of work in terms of handling real world programs, and taking care of many more corner cases.

Another work characterizing information flow in object oriented programs [25], with Java as the reference, also uses program slicing technique. The author developed the concept of a comprehensive Object-oriented Program Dependence Graph (OPDG), that has edges representing polymorphic flows, various class and interface dependencies, and other concepts like inheritance and encapsulation. With respect to Java, there are four kinds of information flows that are identified – statement-level, method-level, class-level and package-level. The analysis involved identifying the basic components and computing the backward and forward slices, using the OPDG. However, this work is mostly theoretical analysis, and no concrete implementation in the form of a usable tool exists. This work can be used as a general guideline when characterizing information flow in OO programs. Also, the work seems to be more applicable in the context of error propagation, and testing, rather than satisfying adversary models. We think this work can be modified to suit the needs of secure information flow, by using the metrics that can be computed using this technique – width of information flow and correlation coefficient.

## IX. Tainting and Static Techniques

Tainting essentially marks untrusted inputs as tainted and tracks the flow of taints through the system. This is the reverse of the information flow confidentiality criterion, which checks if information flows from secret variables to public variables. Hence tainting tries to enforce the second criteria of secure information flow, namely integrity. However, while tainting has numerous applications, dynamic tainting has the significant disadvantage of high overhead since it requires tracking at runtime the taint of the secret values, and for doing so, the usually followed technique is at the finest granularity – byte level tracking. This is because, for unsafe languages like C, alias analysis is not very precise, and hence increasing the precision of the taint mechanism requires resorting to fine grained granularity. A term common to tainting is stability, which is a metric quantified as the variables in the program that are tainted during the tainting process. Hence, a tainting technique with extremely low or weak stability is one that taints almost all variables in the program, thereby introducing false positives. The solutions surveyed in this section try to negate these problems associated with dynamic tainting, by reducing the overhead using static analysis.

Chang et al [20] proposed a solution to detect invalid flows based on some predefined security policy, using static dataflow analysis and dynamic tainting. They cover more attacks that traditional taint-based methods, and at a much lower runtime overhead. This is because, in case the static dataflow analysis manages to find that there are no invalid flows, there is no runtime overhead involved. However, this is at the cost of increased annotation required by the programmer who needs to thoroughly understand the nuances and the design of the code, and a security expert who can specify the security policy this application should adhere to at all times.

Another disadvantage of this system is that the instrumentation added to the code, to check if the security policy is invalidated, guards the system only against that specific security policy. The security policy is specified in a preexisting annotation language called Broadway, and the authors claim that all the required policies can be added by some security expert initially. This seems to be analogous to the signature based intrusion detection system, with the policies being the signatures. Moreover, it should be noted that the overhead associated with the instrumented code is not directly proportional to the complexity of the security policies. The authors claim this to be an advantage, stating that it actually depends on various factors like whether the instrumented code is on the critical path, and other runtime properties. However, we believe that this could be a disadvantage since determining the overhead in systems that are critical to such metrics, might be hard to do, unless the programmer has a good understanding of the underlying system, along with how the instrumentation and static analysis works. In other words, for such systems, the programmer would have the additional burden of figuring out the entire working of the application being protected, and the static and dynamic analysis involved. Finally, this system does not handle implicit flows and side channels through timing attacks. Hence, it would not protect this system against the broader adversary model that we consider.

In terms of low level implementation details, the underlying pointer analysis is client-driven, and the client here is the dataflow annotations, which use the results of the pointer analysis. Also, the taint system is tag based at byte granularity. This would have higher overhead compared to other coarse granularity schemes. The authors admit increasing the size of the trusted computing base (TCB), by including the compiler, the annotated security policies, and the instrumented code that is added as runtime checks to validate that the application adheres to the security policies. It seems reasonable to consider the compiler as part of the TCB, from a pragmatic standpoint.

Tripp et al [19] developed a taint analysis mechanism for web applications, specifically written in Java. They handle the scalability issue and runtime overhead associated with taint analysis, to the point that it was evaluated against real industrial benchmarks. They introduce a much smaller slicing algorithm, namely the hybrid thin slicing algorithm – flow-insensitive dataflow analysis through the heap, and context- and flow- sensitive dataflow analysis through the local variables, thereby reducing the variables tainted during the analysis. They also take care of the dual purpose of confidentiality and integrity, while other taint analysis solutions have primarily focused on integrity. Another interesting aspect about the paper is that they come up with general models to handle complex flows, when importing libraries and other cases of pointer analysis. This is specifically to handle scalability issues.

However, they do not consider implicit flows and the broader adversary model where the active attacker could compromise the system, if the attacker can perform side channel attacks like the timing attacks. The results of this paper are very implementation specific and geared towards web applications written in Java. It would be interesting to know why the server side was chosen, when there are a number or information flow leak vulnerabilities in client side scripting like PHP and Javascript. Moreover, it is not clear how their technique evaluates with respect to stability, since they only look at false and true positives.

## X. Static and Dynamic Techniques

We now look into solutions that leverage a collection of techniques for characterizing information flow leakage – static and dynamic analysis, to improve preciseness and reduce runtime overhead. We specifically analyze the application of information flow, as applied to web application security and Javascript.

Lam et al [21] developed a comprehensive solution with static and dynamic information flow analyses, and model checking, to capture a wide range of vulnerabilities plaguing web applications. The invalid information flow needs to be specified by the programmer or security expert in a high level language called Program Query Language (PQL), which can capture most invalid flows easily, and the programmer can also specify the corrective actions to take if the invalid flow is detected.

The static analysis is done using context-sensitive, flow-insensitive pointer analysis. It should be noted that all invalid flows captured, depends on the flows included in the program specifications. The authors give an example of the PQL specification for the SQL injection attack. However, we think the fundamental limitation of this approach is that it can only capture the flows explicitly mentioned by the programmer. Hence, as the security design is only *as strong as its weakest link*, it is obvious that new information flow attacks, or even covert channel attacks cannot be captured.

The static analysis is done to reduce the runtime overhead associated with dynamic analysis, and since it is conservative, there are no chances of missing out on potential invalid flows.

The runtime instrumentation for dynamic analysis is done only on program points identified by the static analysis. The program is checked to see if the errors are corrected, at the end of each of the phases. This is effectively pruning away the false positives that are generated. Moreover, as the authors claim, this would help the relatively naive programmer to capture invalid flows. However, the catch here is that the programmer should make decisions regarding which flows are invalid, and this would require an understanding of the threat model and the system.

They also develop a model checker which is used in the final phase, and it is designed specifically for applications using Apache Struts and JSP. However, the inherent issue with model checking is scalability, since it runs for each possible representative input. The model checker is used to display the attack vector that can cause the vulnerabilities identified by the previous phases, thereby alleviating the programmer's work. The main advantage of this technique over other systems is the effective pruning of false positives, and providing concrete examples of attack vectors that can cause the particular information leak.

Just et al [22] proposed a hybrid analysis to track information flow in Javascript, and this is different from other solutions, since it deals with various object oriented concepts like dynamic dispatch, the Javascript `eval` function, exceptions etc. They also handle unstructured control flow along with explicit flows, and inter- and intra-procedural dynamic slicing. The static analysis finds the immediate postdominators to take care of blocks with more than one successor, and the runtime analysis involves placing the postdominators and the associated statements in a runtime stack, every time any function or control statement is executed.

An immediate postdominator is the statement to which control will flow, no matter what path has been taken (for example, the statement immediately after an if-else block). It is obvious that runtime exceptions and interprocedural analysis will not satisfy this rule. However, the workaround followed by the authors is having an EXIT node to take care of such runtime exceptions. The control flow of the program when it encounters runtime exception, goes to the EXIT node.

While the analysis is sound, the solution for exception handling has not been implemented, and hence the precision of the technique with respect to false negatives cannot be evaluated.

Moreover, it is obvious that the technique will not work

|  | **Metrics** |  |  |
|---|---|---|---|
| **Technique** | **Precision (FN: False Negatives, FP: False Positives)** | **Scalability** | **Practicability** |
| **Static** | No FN, high FP | Low scalability | requires good knowledge of some specification language |
| **Dynamic** | FN in strict control dependency [11] | runtime instrumentation overhead | very less annotation required |
| **Program Slicing** | Program semantics may change [16], or technique can introduce FN due to incorrect declassification [15] | scalable specific to Java [16], low scalability due to manual effort involved [15] | low since it requires code understanding and more annotations [15] |
| **Tainting and Static** | Possible FN – it does not consider implicit flows [19], or it is bound to the predefined security policy specified [20] | highly scalable to production code [19] | moderate – security policies specified using annotation language [20] |
| **Static and Dynamic** | Low FP but FN possible due to missing security policy [21], | scalability issues for large Javascript code [22] | requires annotation in PQL [21] |

*TABLE I:* Summary of various techniques with respect to the metrics considered. Precision considers only confidentiality, i.e., it is defined with respect to the first adversary model

for large Javascript programs, for example Javascript code in GMail which uses various other libraries.

There would be runtime overhead, in the form of pushing and popping from the stack. Moreover, their labeling technique for static analysis involves initially assigning a security label for each statement in the form of long integers represented using bit vectors, and then tracking information flow using a bitwise `OR` operation. It is not very clear if the authors are restricting to the two standard levels of security, namely `low` and `high`.

With respect to the adversary models, this system handles implicit flows and works for the limited adversary model, but does not consider integrity. Moreover, there is no protection offered against covert channel attacks.

## XI. Discussion and Future Work

There are many interesting observations that we can infer from the research work done in secure information flow using compiler based techniques. Research in this field has spanned various domains – either by using the technique on its own, or by some combinations of techniques to enhance the metrics considered. Information flow characterization has many applications in building secure systems. However, its actual applicability in such systems has been hindered by its performance with respect to some particular metric – precision, scalability, or practicability. The results inferred from the survey are summarized in table I. Another interesting observation is the research effort spent on characterizing information flow specifically for Java programs [19] [25] [16]. We think that this might be because Java is being used in large scale industry projects, while C and C++ are used for low level system coding. However, it is important to characterize secure IFC in C and C++, or confirm the compatibility of the techniques above with these languages, since they are inherently unsafe due to pointers.

It is obvious that applications have different metrics that they are most concerned about. Pragmatically speaking, it is nearly impossible, to handle all the metrics that we have considered with equal importance, and to come up with a flexible solution. For example, for intrusion detection systems, precision in terms of reducing false alarms is very important. Hence, all research in this area have focused on achieving integrity and improving precision, by using combinations of static analysis, dynamic tainting etc.

Also, these techniques which we have considered tend to fall into well defined categories, when dealing with the two aspects of IFC – confidentiality and integrity. Static analysis inherently tries to solve the problem of confidentiality, while dynamic analysis handles integrity. While some solutions claim to support both confidentiality and integrity aspects of IFC [9] [19], they do not perform well with respect to some metric like precision or scalability.

Program Slicing as a technique for handling information flow had some interesting research work where smaller subproblems within information flow were tackled. The techniques for secure information flow using program slicing and static analysis, does not seem practicable, since it requires a solid understanding of some concepts of characterizing information flow, for example when to declassify information. However, there are other techniques which used program slicing as one stage in the entire analysis framework. This lead to an improvement in precision, while not having significant impact on scalability.

Moreover, there are many solutions proposed [21] [20] [17], that have good precision, but depend on the programmer to specify the invalid information flow or patterns in some custom specification language. Apart from the pragmatic concerns in terms of programmer understanding and annotations required, the weakest link in this technique is the human involved in the security framework, and efforts should be made to automate this process.

Dynamic information flow analysis does not seem to hold much promise when it comes to information flow control. This is because apart from scalability issues, there has not been a breakthrough in applying this analysis to confidentiality. When solving the problem of integrity, we need a stronger adversary model, and the techniques surveyed in this paper with respect to dynamic analysis, cannot handle that model yet. We believe it can be better applied to the field of input fuzzing, testing and identifying critical variables in a program.

Nowadays, energy being a big concern in computing, we believe there is a lot of potential in research in applications that can tolerate some information leakage, as long as it does not lead to catastrophic consequences. Let us call this class of applications – leakage tolerant applications. We think that it might be interesting to explore the trade off space between precision and energy. This is because they usually tend to have an inverse relation with each other. The work on strict control dependence in dynamic information flow tracking, is one step in analyzing flows in such applications. The research question is *Can we dynamically configure the solution for leakage tolerant applications, such that they consume lesser energy in performing the analysis, at the cost of leaking some amount of secret data?*

The work involved in this would be to identify dynamically configurable parameters in the fundamental analysis that is done. This would mean developing some heuristics that can prune the analysis, depending on the energy metric. Moreover, fundamental static analysis like pointer analysis, and other expensive analysis would also need to be finetuned, by first deciding if there are avenues for approximation in these analyses, that can help in significant energy savings.

Another possible future work is in the area of static analysis. There seems to be a lot of research potential with respect to static solutions for information flow as it is still in its nascent stages. The inherent issue with static analysis is that it tends to be overly conservative, thereby limiting precision, but is more scalable than dynamic techniques, since there is no runtime overhead involved. The research question we should address is *Can we reduce the false positives incurred in static analysis using some well defined heuristics?* Current research work combines static analysis with dynamic information flow tracking to improve the precision, but this is at the cost of added runtime overhead. The key point here is that we are aiming for a reduction in false positives, not complete elimination. Hence, this is useful in applications that can tolerate some degree of false positives, but requires scalability and speed.

As a first step, it would be interesting to quantify the results of static analysis, and use heuristics to prune away information flows that do not fall within a specified threshold. The work done in handling information leaks in mobile OS [8] [10] seems very promising with respect to using static analysis in an industrial setting. There might be other avenues where static analysis would prove to be a very viable solution for secure information flow.

## XII. Conclusion

We have surveyed and critically evaluated current research work done in the field of information flow security, using principles in compilers like program analysis – static and dynamic, program slicing, and tainting. Usually work related to information flow presents solutions to handle some specific cases, but they do not explicitly mention their adversary model. Most of the research work, generally tends to focus more on providing elegant solutions that works well with respect to some metric, i.e., scalability, precision or practicability. The techniques surveyed in this paper have been critically evaluated against three specific adversary models, where the adversary has varying levels of capabilities.

Moreover, these solutions were constantly reviewed to verify that it satisfies the ten principles of computer security. We found that some work does not keep the design simple – by having a larger trusted computing base, and some others have questionable assumptions about their threat model, especially solutions targeting intrusion detection systems. We have found that while most of the research has focused on protecting confidentiality of applications, research for protecting integrity by characterizing information flow as invalid, is relatively at its nascent stage. There has been decades of research on formally verifying information flow in systems using type based systems and programming constructs. While these solutions offer soundness, they are even more conservative than static analysis, and also require annotations from the programmer or security expert. We believe there is a lot of research potential and growing industry relevance in using compiler techniques like program analysis and tainting for secure information flow.

## References

[1] W. Masri and A. Podgurski, "Using dynamic information flow analysis to detect attacks against applications," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–7, May 2005.

[2] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, pp. 236–243, May 1976.

[3] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *Proceedings of the 18th IEEE workshop on Computer Security Foundations*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 255–269.

[4] G. Smith, "Principles of secure information flow analysis," in *Malware Detection*, ser. Advances in Information Security, M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Eds. Springer US, 2007, vol. 27, pp. 291–307.

[5] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331.

[6] M. Pistoia and U. Erlingsson, "Programming languages and program analysis for security: a three-year retrospective," *SIGPLAN Not.*, vol. 43, pp. 32–39, February 2009.

[7] C. Hammer, "Information flow control for java based on path conditions in dependence graphs," in *In IEEE International Symposium on Secure Software Engineering*, 2006.

[8] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications." in *NDSS*. The Internet Society, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2011.html#EgeleKKV11

[9] Y. Liu and A. Milanova, "Static analysis for inference of explicit information flow," in *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '08. New York, NY, USA: ACM, 2008, pp. 50–56.

[10] X. Xiao, N. Tillmann, M. Fahndrich, J. de Halleux, and M. Moskal, "Transparent Privacy Control via Static Information Flow Analysis," Microsoft Research, Tech. Rep., 08 2011.

[11] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu, "Strict control dependence and its effect on dynamic information flow analyses," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 13–24.

[12] S. McCamant and M. D. Ernst, "Quantitative information flow as network flow capacity," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 193–205.

[13] W. Masri, A. Podgurski, and D. Leon, "Detecting and debugging insecure information flows," in *In ISSRE04: the 15th International Symposium on Software Reliability Engineering*, 2004, pp. 198–209.

[14] M. I. Al-Saleh and J. R. Crandall, "On information flow for intrusion detection: what if accurate full-system dynamic information flow tracking was possible?" in *Proceedings of the 2010 workshop on New security paradigms*, ser. NSPW '10. New York, NY, USA: ACM, 2010, pp. 17–32.

[15] S. F. Smith and M. Thober, "Refactoring programs to secure information flows," in *Proceedings of the 2006 workshop on Programming languages and analysis for security*, ser. PLAS '06. New York, NY, USA: ACM, 2006, pp. 75–84.

[16] S. Cavadini, "Secure slices of insecure programs," in *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ser. ASIACCS '08. New York, NY, USA: ACM, 2008, pp. 112–122.

[17] A. B. Shaffer, M. Auguston, C. E. Irvine, and T. E. Levin, "A security domain model to assess software for exploitable covert channels," in *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, ser. PLAS '08. New York, NY, USA: ACM, 2008, pp. 45–56.

[18] C. Hammer, R. Schaade, and G. Snelting, "Static path conditions for java," in *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, ser. PLAS '08. New York, NY, USA: ACM, 2008, pp. 57–66.

[19] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 87–97.

[20] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in *Proceedings of the 15th ACM conference on Computer and communications security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 39–50.

[21] M. S. Lam, M. Martin, B. Livshits, and J. Whaley, "Securing web applications with static and dynamic information flow tracking," in *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, ser. PEPM '08. New York, NY, USA: ACM, 2008, pp. 3–12.

[22] S. Just, A. Cleary, B. Shirley, and C. Hammer, "Information flow analysis for javascript," in *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, ser. PLASTIC '11. New York, NY, USA: ACM, 2011, pp. 9–18.

[23] J. Heusser and P. Malacaria, "Quantifying information leaks in software," in *Proceedings of the 26th Annual Computer Security Applications Conference*, ser. ACSAC '10. New York, NY, USA: ACM, 2010, pp. 261–269.

[24] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *International Journal of Information Security*, vol. 8, pp. 399–422, 2009.

[25] B. Li, "A technique to analyze information-flow in object-oriented programs," *Information & Software Technology*, vol. 45, no. 6, pp. 305–314, 2003.

[26] A. Rountev, B. G. Ryder, and W. Landi, "Data-flow analysis of program fragments," in *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. ESEC/FSE-7. London, UK, UK: Springer-Verlag, 1999, pp. 235–252.

[27] W. Masri and A. Podgurski, "Using dynamic information flow analysis to detect attacks against applications," in *Proceedings of the 2005 workshop on Software engineering for secure systems building trustworthy applications*, ser. SESS '05. New York, NY, USA: ACM, 2005, pp. 1–7.