RIFLE: An Architectural Framework for User-Centric Information-Flow Security

Neil Vachharajani Matthew J. Bridges Jonathan Chang Ram Rangan Guilherme Ottoni
Jason A. Blome George A. Reis Manish Vachharajani David I. August

Departments of Computer Science and Electrical Engineering
Princeton University
{nvachhar, mbridges, jcone, ram, ottoni, jblome, gareis, manishv, august}@princeton.edu

Abstract

Even as modern computing systems allow the manipulation and distribution of massive amounts of information, users of these systems are unable to manage the confidentiality of their data in a practical fashion. Conventional access control security mechanisms *cannot prevent the illegitimate use of privileged data once access is granted. For example, information provided by a user during an online purchase may be covertly delivered to malicious third parties by an untrustworthy web browser. Existing* information-flow security mechanisms *do provide this assurance, but only for programmer-specified policies enforced during program development as a static analysis on special-purpose type-safe languages. Not only are these techniques not applicable to many commonly used programs, but they leave the user with no defense against malicious programmers or altered binaries.*

In this paper, we propose RIFLE, a runtime information-flow security system designed from the user's perspective. By addressing information-flow security using architectural support, RIFLE gives users a practical way to enforce their own information-flow security policy on all programs. We prove that, contrary to statements in the literature, runtime systems like RIFLE are no less secure than existing language-based techniques. Using a model of the architectural framework and a binary translator, we demonstrate RIFLE's correctness and illustrate that the performance cost is reasonable.

1. Introduction

In modern computing systems, security is becoming increasingly important. Computers store tremendous amounts of sensitive information. Personal and business computers store private data such as tax information, banking information, and credit card numbers. Computers used for military applications store extremely sensitive information where confidentiality is critical. Since these computers are often connected to potentially hostile public networks, such as the Internet, security mechanisms to protect the confidentiality of this data are vital.

Various *discretionary access control* security mechanisms [14] are typically used to protect data. For each data access, a policy (e.g. an access control list or file permissions) is checked to see if the access is permitted, and if so, the data is returned. While these mechanisms prevent unauthorized data accesses, once a program is granted access, the data's owner has no control over how the application uses the data. Thus, access control security mechanisms give data owners only two options: deny access to data altogether or trust programs to keep data confidential.

Unfortunately, deciding which programs to trust is difficult. Web browser embedded applications such as Java applets or Flash applications, for example, have made it possible to download and immediately execute arbitrary programs from untrusted sources on the Internet with little to no user intervention. Even full applications explicitly downloaded by the user from untrusted sources can be problematic as demonstrated by so-called *spyware* programs [17] (programs that typically install with other programs and send data collected from the host computer back to the program's creator).

Unlike access control systems, *information-flow security* (IFS) systems [7, 8, 12, 21, 23] allow untrusted applications to access confidential data while preventing them from leaking this information to other programs or people without explicit authorization from the data owner. The focus of existing work on these systems has been on language-based and static analysis mechanisms for implementing the security policies. In these systems, programs are written in special programming languages that contain security annotations. During compilation, the compiler, assisted by source code annotations, verifies that illegal information leaks as defined by the programmer cannot occur (except possibly through a variety of *covert channels* [21, 28]).

Unfortunately, although these systems can enforce information-flow security policies for programs written in an IFS programming language, the user has no guarantee of this safety since the compiler assures the programmer, not the user, of a program's safety. To the user, a binary for a safe program is indistinguishable from one that is unsafe. Use of a proof-carrying code (PCC) framework [24] can overcome this problem by proving properties of the program to the user. However, PCC frameworks, like the IFS programming languages, suffer from a low adoption rate and from their inability to deal with existing code.

Additionally, even if all programs were written in IFS languages and validated by users with a PCC framework, the user would still be at the mercy of programmer-defined policies. This must occur in language-based systems since policy enforcement occurs at compile-time. It is possible to build run-time (as opposed to compile-time) IFS systems which allow users to establish their own security policies. However, few run-time systems have been studied because run-time systems are believed to be inherently less secure than language-based, static systems [21]. As a result, existing run-time systems are either not practical [27] or restrict the user's freedom in defining policy at compile time [31]. In this paper, we prove that, contrary to these beliefs, run-time systems are no less powerful than static systems, and we reestablish the feasibility of enforcing information-flow security completely from the user's perspective.

With this insight, we present RIFLE, a Run-time Information FLow Engine capable of enforcing user-defined information-flow security policies for *any* program. In RIFLE, information-flow security policies are enforced using a combination of binary translation and a modified architecture. Program binaries are translated from a conventional instruction-set architecture (ISA) to an information-flow secure (IFS) ISA. The translated programs are executed on hardware that aids information-flow tracking, and the executed programs interact with a security-enhanced operating system that is ultimately responsible for enforcing the user's policy. This three part enforcement mechanism ensures that confidential data is not leaked and provides users with complete control over the confidentiality of their data.

The contributions of this work are:

1. The first description of the information-flow problem from the user's perspective.
2. A proof that, contrary to conventional wisdom, language-based information-flow systems are less secure than previously thought.
3. An architectural approach, called RIFLE, that
   (a) is at least as powerful as language-based information-flow systems.
   (b) empowers users with the ability to set policies for their data rather than relying on programmers.
   (c) is language independent, supporting legacy codes.
4. An evaluation of RIFLE for performance on compute intensive benchmarks and a demonstration of its abilities on applications of practical interest.
5. Insight that sets future research directions in user-centric information-flow security including performance opportunities, other uses for information-flow tracking, and the utility of a declassification scheme.

The next section gives context for IFS. Section 3 describes existing language-based IFS mechanisms which have been studied heavily in the literature. Section 4 explores the challenges faced by language-independent, run-time information-flow systems. Section 5 presents RIFLE, and Section 6 evaluates the properties of it. Finally, the paper concludes in Section 7.

2. Information-flow Security

Information-flow security (IFS) mechanisms allow users to control or audit information even after programs have been given access. This is a powerful ability with many applications ranging from supporting compliance with new laws mandating the auditing of medical records to alerting users when spyware delivers sensitive information to a network device. Here, however, the discussion of IFS begins with a simple example.

2.1. An Information-flow Security Example

Consider the average home computer user, call her Alice, who decides to upgrade her computer to Microsoft Windows XP. At the conclusion of the upgrade process, Alice must register her copy of Windows by sending a seemingly random, possibly encrypted, sequence of numbers that was computed by the installation software to Microsoft. Unfortunately, Alice has no idea what these numbers mean or what information they encode. The software claims the signature is computed by hashing Alice's computer hardware configuration, but the sequence could contain sensitive information that was stored on Alice's computer. While it is likely that the signature is benign and contains only the information the installation software claims, the computer and the software running on it provide Alice no mechanism to verify that this is in fact the case.

This situation is not restricted to the installation of Microsoft Windows XP. In general, any program Alice runs will take input data and transform it into output data. Unfortunately, Alice does not know what parts of a program's input are encoded in the various outputs. Since the data transformation effectively occurs in a black box, Alice is unable to make an informed decision about how she can deliver the program data while maintaining the confidentiality of her input data. Instead, she is forced to make her decision on the basis of trust or necessity. In the Windows XP registration process, Alice would probably allow the signature to be sent to Microsoft since she either trusts Microsoft or realizes that her only other option is to delete Windows.

2.2. Ensuring Confidentiality with IFS

Information-flow security allows data confidentiality to be ensured even in the presence of untrusted applications. IFS policies are defined by a set of annotations, called *labels*, that are attached to all values and storage locations in a program, and a set of legal *flows*, pairs of labels, that determine how information can flow. If a value has label l_1 and a storage location has label l_2, the value can be stored into that location only if the flow $l_1 \rightarrow l_2$ is allowed by the policy. To allow access to data while ensuring security, an information-flow security mechanism's goal is to verify that a program only contains legal flows.

Returning to the Windows XP registration example, one could imagine attaching the label hardware to data describing computer hardware and label private to confidential data. When the installation software computes the hardware signature, the label of the signature would inform the user of what data was used to produce it. If the data label is hardware, the user (or the information-flow security mechanism) would send the signature to Microsoft. On the other hand, a label of $\{\mathsf{hardware}, \mathsf{private}\}$ would indicate that both hardware configuration information *and* confidential data were used to produce the signature. In such a case, a user may be less inclined to reveal the signature.

Using simple named labels and explicit enumeration of legal flows as described above is one example of how one could define an IFS policy. More formal methods exist including those which allow distributed declassification of data [10, 22]. However, regardless of the specific labeling methodology, the mechanisms used for verifying an information-flow policy remain unchanged.

IFS does not completely remove trust from the system since a user must trust the enforcement mechanism. Fortunately, the size of the trusted computing base can be small and many techniques exist to establish a trusted computing base. There are various techniques to prove that processors are trustworthy including Intel's LaGrande [13], Microsoft's Next-Generation Secure Computing Base (Palladium) [20] and the Trusted Computing Platform Alliance [30]. Mechanisms to establish trusted memories have also been proposed [18, 19, 34]. Finally, techniques to bootstrap the trusted system to create a usable trusted base from a small set of trusted components also exist [2, 3, 29].

3. Existing IFS Systems

To enforce an information-flow security policy, an IFS mechanism must be able to identify the label of all data being processed by a program and verify that all data flows in it are legal with respect to a specific policy. This section will describe a compile-time approach, used by existing information-flow security mechanisms, for tracking data labels and identifying illegal flows. These systems rely on special information-flow secure programming languages that allow programmers to annotate their programs with labels. From these annotations, the compiler will *statically* compute the labels of all expressions in the program and verify that all information flows are legal. Since these systems verify the security of a program at compile-time, they are in a class we will call *static* information-flow security mechanisms. This section describes the properties of these existing systems and outlines their shortcomings.

3.1. Information Flow via Data Flow

The majority of statements in a program compute some value based on one or more variables and then store the results into some other variable. For example, consider the pseudo-code `c = a + b`. Clearly information is flowing from the variables a and b into the variable c. If a programmer were to annotate the variables a, b, and c with labels (which will be denoted as $\underline{a}$, $\underline{b}$, and $\underline{c}$ respectively), and if the compiler knew the set of legal flows, then it could verify that the flows $\underline{a} \rightarrow \underline{c}$ and $\underline{b} \rightarrow \underline{c}$ were legal.

Rather than check these two flows independently, for simplicity, the static analyses in the compiler define a label for each expression in the program. The compiler then checks if the flow between the expression label and the destination variable label is legal. To find the label for an expression, the compiler computes the *join* ($\oplus$) of all the operands in the expression. For the `c = a + b` example, the compiler would calculate a label for the expression `a + b` by computing $\underline{a} \oplus \underline{b}$. The join operator combines labels and produces the most permissive label that is at least as restrictive as both of its operands. If, for example, labels were sets of users permitted to access some data, then the join operator would be set intersection; only those users allowed to read all data used in a computation should be able to read the result of the computation.

3.2. Information Flow via Control Flow

Unfortunately, verifying the data flow of a program is insufficient to verify that no illegal information flows occur in the program. Consider the program shown in Figure 1. If the variables x and y are both Boolean variables, then at the end of this segment of pseudo-code, the value of y would be equal to the value of x. Clearly, there is information flow between x and y, but this flow does not occur through data

```
1  if (x == true)
2    y = true;
3  else
4    y = false;
5  // Since y == x, the label of y should be
6  // at least as restrictive than the label of x
```

Figure 1. Information flow through control

flow. In the literature, this type of flow is called an *implicit flow* to contrast it with the *explicit flow* seen earlier [28].

To verify that no illegal implicit flows occur in a program, for each statement, the compiler must identify which branches control the statement. In this example, the condition `x == true` controls the statements on lines 2 and 4. Thus when performing the data flow verification for those statements, the compiler must also verify that $\underline{\mathtt{x}} \rightarrow \underline{\mathtt{y}}$ is a legal flow. Just as in the case of statements with multiple source operands, in this example, the compiler can verify the flows $\underline{\mathtt{true}} \oplus \underline{\mathtt{x}} \rightarrow \underline{\mathtt{y}}$ and $\underline{\mathtt{false}} \oplus \underline{\mathtt{x}} \rightarrow \underline{\mathtt{y}}$. In general constants are annotated with the least restrictive label denoted $\perp$ and $\perp \oplus \underline{\mathtt{n}} = \underline{\mathtt{n}}$.

3.3. Flow Verification and Type Checking

Static systems function by having the compiler determine whether the information flows in the program are compatible with the programmer's label annotations. This process of verifying flows by checking labels is very similar to the process of type-checking [21, 22] a program. Since the information-flow labels of variables are conceptually the same as types, performing "type-checking" of the information-flow labels (for a type-safe language) will guarantee that no data with a given label will ever be transferred into a variable with an incompatible label.

3.4. Shortcomings of Static Systems

While static systems do provide information-flow security they have several disadvantages stemming from the programmer-centric approach. First, since policy enforcement occurs at compile-time static IFS systems provide no guarantee of security to users, but instead, provide them to the programmer. Static IFS systems can be embedded into a proof-carrying code framework (PCC) [24], where programmers provide users with compiler-generated security proofs for programs. Unfortunately, the user is at the mercy of the programmer to provide such a proof.

Even if a PCC framework is used, security policy decisions are still made by the programmer. Recall that it was the programmer who labeled program variables, and, since flows were verified during compilation, it was the programmer who defined the set of legal flows. If the security policy chosen by the programmer is too lax or overly conservative for the user, the user must abandon use of the program or take risks with its use.

Finally, static information-flow systems dramatically reduce the space of applications available to users since the security can only be guaranteed for programs written in specific IFS languages. It is possible to extend an existing language to support information-flow security [21], however only those languages with strong type safety guarantees can be extended in this way. Therefore, all legacy applications cannot be checked for security and future applications developed in type-unsafe languages such as C or C++ also cannot be checked for security.

4. The Dynamic Approach

Since policy determination and enforcement occur at compile-time in static IFS systems, users are unable to set and enforce individual policies and are at the mercy of programmers to provide guarantees of security. To provide IFS from the user's perspective, this section reconsiders the fundamentals of run-time solutions which have been largely abandoned by IFS researchers. The run-time approach is the core of RIFLE as described in later sections.

4.1. Tracking Information Flow Dynamically

Dynamic mechanisms track information flow at program run-time rather than during compilation. In a dynamic information-flow approach, instead of statically assigning a label to each storage location, labels act as additional program data that propagate through computation. When an operation is performed, the labels, in addition to the data, are read from the operation's inputs. The join of these labels is computed, and, in addition to the operation's result, the resulting label is stored in the target storage location. Initial data labels are provided by the user along with the program input. Consider the earlier example of `c = a + b`. If this code were executed with dynamic information-flow tracking, rather than verifying that the flow $\underline{\mathtt{a}} \oplus \underline{\mathtt{b}} \rightarrow \underline{\mathtt{c}}$ is legal, the label *assignment* $\underline{\mathtt{c}} := \underline{\mathtt{a}} \oplus \underline{\mathtt{b}}$ would occur.

This data flow mechanism *tracks* information as it flows through the system, but it does *not* provide any level of security. To enforce security, dynamic systems verify flows to output channels (files, shared memory, etc.). Unlike storage locations, output channels have a constant, user-defined label. When a program operation attempts to write data with label l to an output channel C, rather than assigning the output channel a new label, the system verifies that the flow $l \rightarrow \underline{C}$ is legal. If the flow is found to be illegal, the program terminates. Otherwise, if the flow is legal, the data is copied to the output channel and the program continues execution.

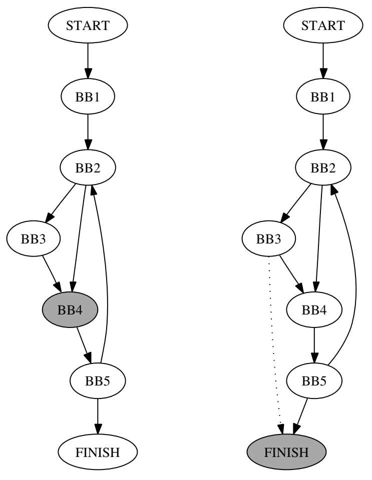
Just as in the static approach, it is necessary for a dynamic mechanism to track information through implicit flows in addition to explicit ones. The naïve approach for handling implicit flows is to directly apply to dynamic systems the control-flow technique used in static systems. Static systems tracked implicit flows by joining the label of *controlling* branches into the label that results from an

```
     int secret_data; int i;

BB1: secret_data = ...;

BB2: for(i = INT_MIN; i <= INT_MAX; i++) {
BB2:   if(i == secret_data)
BB3:     work(); //throws an unchecked exception
BB4:   printf("x");
BB5: }
```

(a) Termination channel attack program



(b) Perceived CFG (c) Actual CFG

Figure 2. Termination channel attack

operation's computation. In the static system, this label was *checked* against the label of the destination storage location, while in a dynamic system, this label would be *assigned* to the destination storage location. While this approach seems secure, Section 4.2.2 will explain why it is inadequate.

4.2. Static vs. Dynamic Mechanisms

Existing work has primarily focused on static information-flow mechanisms because they are believed to be inherently more secure than dynamic systems [22]. This section first shows that this is, in fact, not the case. The section then proceeds to discuss why an effective dynamic scheme must handle certain attacks that are naturally avoided by static systems. Later sections will illustrate how RIFLE prevents these attacks.

4.2.1. Termination Channel Attacks

Static systems are believed to be more secure than dynamic systems because dynamic systems are vulnerable to the following attack. A program is constructed that contains a security violation for certain input sets, but no violation for others. Since security violations (in dynamic IFS systems) cause program termination, observing whether or not this program terminates abnormally provides information about the program input. Static systems avoid this attack since security violations are identified at compile time and are input *independent*.

This type of attack causes information to leak through the *program termination channel* which is an example of a *covert channel* [15, 28]. While static systems avoid abnormal program termination due to security violations, they also have termination channels since the channel exists whenever a program can throw an *unchecked exception* (an exception that causes program termination) such as an out-of-memory exception or a null-pointer dereference. IFS violations are only one example of unchecked exceptions. It is believed, however, that static systems are less vulnerable to termination attacks because they can only leak one bit of information per program execution [21] while dynamic systems can leak an arbitrary amount of information per execution. Unfortunately, this is *not the case*, both systems leak identical amounts of information.

Consider the pseudo-code shown in Figure 2(a). The program in this example will output the value of the variable `secret_data` in unary. For example, if the value of `secret_data` is 7, then the program will output the character 'x' 6 times ("xxxxxx")—the program terminates before the seventh 'x' is printed. The program iterates over all possible `integer` values, in order, outputting an 'x' each time `secret_data` does *not* match the iterator. When a match is encountered the program calls the `work` function which intentionally causes an unchecked exception resulting in program termination.

Assuming that the variable `secret_data` was marked with a label which should not be permitted to flow to the output, then this program ought to be rejected by an information-flow mechanism. Unfortunately, neither static nor dynamic mechanisms will prevent this attack. The data leak can be traced to a subtle, incorrect assumption about the control-flow graph of the program. The naïve control-flow graph for the program is shown in Figure 2(b). The node in grey marks the first node after BB2 that is control independent of the branch in BB2 (the immediate post-dominator). Since BB4, the node containing the statement which prints 'x', is not dependent on the conditional statement in BB2 and since the print statement outputs a constant, it is not dependent on any confidential data. Consequently both static and dynamic information-flow security mechanisms declare the program safe for execution.

Figure 2(c) shows a modified version of the control-flow graph. An edge between BB3 and FINISH has been added (shown as a dotted line). The new graph is a more accurate representation of the *actual* run-time control flow since the unchecked exception in BB3 could cause the program to terminate. In this new control-flow graph, BB4 (which contains the print statement) *is* control dependent on BB3 and transitively on BB2. Therefore, although the print statement is outputting a constant, an information-flow security mechanism would forbid the output due to the control dependence on confidential data.

Unfortunately, since dividing by zero, dereferencing a `null` pointer, indexing an array out-of-bounds, or even exhausting some system resource could cause an unchecked exception, using the correct control-flow graph would cause information-flow security mechanisms to reject most practical programs. It has been suggested that unchecked exceptions be forbidden [8], however such a scheme is also unrealistic. For many programs, there exists no suitable recovery from a `null` pointer dereference, for example. Therefore, even if the exception were checked, the program would have no choice but to explicitly exit. The control-flow graph of such a program is identical to the correct control-flow graph for a program with unchecked exceptions. Consequently, forbidding unchecked exceptions would also cause most programs to be rejected by an IFS mechanism.

Despite this attack and other possible attacks through covert channels, information-flow security is still valuable. This particular attack for example, requires a computation and output size that is exponential in the number of bits leaked. Attacks similar to this one may not output data in a unary format, but will also be similarly rate-limited. Thus, these attacks cannot leak substantial amounts of information in any reasonable amount of time.

4.2.2. Control-Flow Attacks

While both static and dynamic systems can both leak arbitrary amounts of information, the naïve dynamic control-flow management strategy (based on a straight-forward translation of static systems) can do so at a rate linear in the number of bits leaked.

Consider the effectiveness of the straight-forward dynamic control based information-flow strategy described earlier on the program shown in Figure 3. The program takes one Boolean input, a. Assuming $\underline{a} = \mathsf{secret}$ at the start of the program, Table 1 traces the values and labels of all variables for the case when $a = \mathtt{false}$ and when $a = \mathtt{true}$.

When a is `false`, the program follows the solid path in the figure. When executing block W, $c := \mathtt{true}$ and $\underline{c} := \underline{a} = \mathsf{secret}$ since the condition in block V is based on a and W is control dependent on block V. Execution continues in block X where the program falls through to block Z leaving all other variables unaffected. At the end of the execution, b is equal to a, but $\underline{b}$ is less restrictive than $\underline{a}$.

When a is `true`, the program follows the dashed path in the figure. The program falls through from block V to X. From there control is transferred to block Y where $b := \mathtt{true}$ and $\underline{b} := \underline{c} = \bot$ since the condition in block X is based on c and Y is control dependent on block X. The program then completes execution in block Z. Once again, at the end of the execution, b is equal to a, but $\underline{b}$ is less restrictive than the $\underline{a}$.

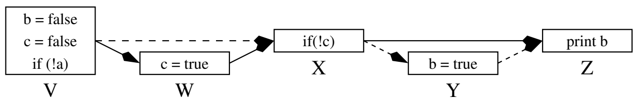


Figure 3. Program demonstrating that dynamic information-flow security mechanism must be insecure or overly restrictive

This program clearly has a flow of information from a to b, but the naïve dynamic information-flow tracking mechanism is not appropriately updating variable labels. If the flow $\underline{a} \rightarrow \underline{\mathtt{stdout}}$ is illegal, then this program will avoid the policy by hiding the information flow. This attack against dynamic information-flow mechanisms was originally proposed by Fenton [9], and it demonstrates that building a safe dynamic scheme is nontrivial.

The attack is possible because the naïve dynamic information-flow tracking mechanism only modifies the labels of storage locations when something is assigned to that storage location. Unfortunately, information flows can occur because instructions are *not* executed. For example, skipping the execution of block W communicates the value of a to c just as executing block W does.

In order to address this problem it seems as though a dynamic scheme would need to analyze the paths of execution *not* taken to see what storage locations *could* potentially be written. However, such schemes are impractical because the size of the not-taken path can grow exponentially if it contains many branches. Further, looking at the not-taken path may not reveal the accessed storage locations because they are hidden behind pointer computation. In general identifying what memory locations could possibly be accessed is undecidable [16, 26].

Given that fully examining and analyzing the full control-flow graph is not feasible dynamically, a workable dynamic scheme must somehow ensure information-flow security in the presence of incomplete information. Any such dynamic scheme must be overly restrictive in that it will consider some secure programs insecure, as shown by the following theorem.

Theorem 1. *Any information-flow security mechanism that cannot know what state is modified by code that is not executed is either insecure or overly restrictive.*

We say that an information-flow security mechanism is *secure* if no information-flow policy is violated (a mechanism *is* secure if violations are detected and prevented

	a = false						a = true					
Block Label	a	$\underline{a}$	b	$\underline{b}$	c	$\underline{c}$	a	$\underline{a}$	b	$\underline{b}$	c	$\underline{c}$
V	false	secret	false	⊥	false	⊥	true	secret	false	⊥	false	⊥
W	false	secret	false	⊥	true	secret						
X	false	secret	false	⊥	true	secret	true	secret	false	⊥	false	⊥
Y							true	secret	true	⊥	false	⊥
Z	**false**	**secret**	**false**	⊥	true	secret	**true**	**secret**	**true**	⊥	false	⊥

Table 1. Execution trace of program from Figure 3

dynamically). An information-flow security mechanism is *overly restrictive* if some *legal* flow is prohibited. With these definitions, the theorem follows directly from the undecidability of determining semantic dependences between program statements [25].

4.3. Dealing with Undecidability

Theorem 1 clearly states that in order for an information-flow mechanism to be secure, it *necessarily* must be overly restrictive. This necessity stems from the undecidability of determining semantic dependence between two statements in a program. While this undecidability does preclude a *perfect* information-flow security system, it does *not* preclude the existence of restrictive systems that rely on *conservative* solutions to the semantic dependence problem. In practice, conservative solutions to many undecidable problems are used in lieu of perfect solutions. For example, compilers regularly use conservative pointer alias analyses to facilitate optimizations. The optimizations are successful, and correct, despite the undecidability of pointer analysis.

The challenge, therefore, in building a dynamic information-flow security system is identifying which approximate solutions to the semantic dependence problem yield secure enforcement mechanisms that are not *too* restrictive in practice. The remainder of the paper will discuss a dynamic information-flow security mechanism which is secure, but *adapts* to information obtained from static analysis to become as permissive as possible. To allow the proposed mechanism to be independent of the language a program is written in, it consists of architectural component to track information flow, a binary translation component to guide the architecture, and an operating system component to determine and enforce policy. Since the mechanism requires *no* annotations from the programmer, it not only supports future programs written in arbitrary languages, but can be directly applied to existing applications.

5. RIFLE

RIFLE works by translating a normal program binary into a binary that will run on a processor architecture that supports information-flow security. To avoid the pitfalls dynamic mechanisms encountered while tracking implicit flows, the binary translation will convert *all* implicit flows to explicit flows. The RIFLE architecture is then responsible only for tracking explicit flows. Since access to all output channels in a program pass through the operating system, the operating system will be augmented to use the labels tracked by the architecture to ensure that no illegal flow occurs. The translation is not intended to eliminate covert channels (such as timing channels) [15, 35], but to identify implicit flows in a program. Solutions found in the literature to address covert channels [35] are directly applicable to this system.

This section will first present an abstract architecture that can track information flow through explicit flows. The section then describes the binary translator which augments programs with new instructions to convert *all* implicit flows into explicit ones. The section then gives a brief explanation of considerations for the operating system.

5.1. Abstract Information-Flow Architecture

To enforce information-flow security, our mechanism will convert programs targeted for a conventional instruction-set architecture (ISA) to programs for an information-flow security (IFS) ISA. The IFS ISA augments all state defined in the base ISA with space to store a label; this includes augmenting both registers and memory. Additionally, for each instruction in the base ISA, there is an instruction in the IFS ISA. The semantics of the IFS ISA instruction are identical to that of the base ISA instruction with respect to the state defined by the base ISA. Converted programs, therefore, will have identical semantics to that of the original program. In addition to these base semantics, each instruction in the IFS ISA will use the augmented state to track explicit information flows. To allow translated programs to track implicit flows, the IFS ISA also defines additional security registers to hold auxiliary labels and instructions to manipulate these security registers and the labels affixed to general purpose registers. Since the original program does not have access to the security state and since the binary translation is trusted, attacks using the security state are impossible.

The augmentations described above can be applied to any conventional ISA. However, for clarity, the remainder of this section will discuss how information-flow security can be implemented with the abstract ISA shown in Table 2. This table shows the base ISA instructions and their semantics. These instructions represent the common instructions found in a general purpose RISC ISA. Notice that all register to register instructions have been condensed into a sin-

Base ISA Instruction	Base ISA semantics	IFS ISA Instruction	Augmented ISA semantics
regop R[a]=R[b],R[c]	$\texttt{R[a]} := \texttt{R[b]}\,op\,\texttt{R[c]}$	$<$S[j],...$>$regop R[a]=R[b],R[c]	$\underline{\texttt{R[a]}} := \underline{\texttt{R[b]}} \oplus \underline{\texttt{R[c]}} \oplus \texttt{S[j]} \oplus \ldots$
load R[a]=[R[b]]	$\texttt{R[a]} := \texttt{Mem[R[b]]}$	$<$S[j],...$>$load R[a]=[R[b]]	$\underline{\texttt{R[a]}} := \underline{\texttt{Mem[R[b]]}} \oplus \underline{\texttt{R[b]}} \oplus \texttt{S[j]} \oplus \ldots$
store [R[a]]=R[b]	$\texttt{Mem[R[a]]} := \texttt{R[b]}$	$<$S[j],...$>$store [R[a]]=R[b]	$\underline{\texttt{Mem[R[a]]}} := \underline{\texttt{R[a]}} \oplus \underline{\texttt{R[b]}} \oplus \texttt{S[j]} \oplus \ldots$
(R[a])branch T	if(R[a]) jump to T	(R[a])branch T	-
-	-	$<$S[j],...$>$join S[a]=S[b],S[c]	$\texttt{S[a]} := \texttt{S[b]} \oplus \texttt{S[c]} \oplus \texttt{S[j]} \oplus \ldots$

Table 2. Abstract machine instructions: R[i] refers to general register i, S[i] refers to a security register i, Mem[a] refers to the memory location specified by the address a, and $\underline{\texttt{X}}$ refer to the label of the data element X

```
 1  // Assume R[1] contains a
 2  // b will be stored in R[2]
 3  // c will be stored in R[3]
 4       mov R[2] = 0
 5       mov R[3] = 0
 6       (R[1]) branch .L1
 7       mov R[3] = 1
 8  .L1: (R[3]) branch .L2
 9       mov R[2] = 1
10  .L2: store [R[5]] = R[2]
```

(a) Program from Figure 3 translated into the base ISA

```
 1  // Assume R[1] contains a
 2  // b will be stored in R[2]
 3  // c will be stored in R[3]
 4              mov R[2] = 0
 5              mov R[3] = 0
 6              mov S[1] = labelof(R[1])
 7              (R[1]) branch .L1
 8       <S[1]> mov R[3] = 1
 9  .L1: <S[1]> mov S[3] = labelof(R[3])
10              (R[3]) branch .L2
11       <S[3]> mov R[2] = 1
12  .L2: <S[3]> store [R[5]] = R[2]
```

(b) Program from Figure 3 translated into the IFS ISA

Figure 4. Example of IFS ISA tracking implicit flows

gle instruction. For each base ISA instruction, the table also shows the IFS instruction and its augmented semantics. Except for the branch instruction, all IFS instructions take additional *security register operands* which are used to help track implicit flows (these operands are listed in angle brackets before an instruction). Finally, the IFS ISA has one additional instruction that computes the join of two labels.

Before describing how automatic binary translation can be used with an IFS ISA to track implicit information flows, consider how the program shown in Figure 3 could be translated to the IFS ISA to prevent information leaks. The program has been translated into the base ISA and is shown in Figure 4(a). The print at the end of the original program has been replaced with a store instruction. The IFS ISA translation is shown in Figure 4(b). Two security register defines have been added to the program on lines 6 and 9. The define of S[3] on line 9 will compute the join $\underline{\texttt{R[3]}} \oplus \texttt{S[1]}$. When the store is performed on line 12, since it uses security operand S[3], the stored data will have the label $\underline{\texttt{R[5]}} \oplus \underline{\texttt{R[2]}} \oplus \texttt{S[3]}$. Since S[3] is more restrictive than S[1] and S[1] contains $\underline{\texttt{R[1]}}$, the flow from R[1] to the memory location will be correctly identified.

5.2. Automatic Binary Translation

Since the IFS architecture only tracks explicit flows, an input program must be converted into a program where all implicit flows are made explicit. As was described in the last section, this can be accomplished by translating an input program binary into a secure binary by adding the appropriate instructions and security operands. This section will describe how these transformations can be performed automatically by a binary translator. The binary translator will leverage static analysis to make the information-flow secure binary as permissive as possible.

5.2.1. Basic Translation

Since all implicit flows occur due to control transfer instructions, the binary translator must first define a security register based on the predicate of the branch. Each branch instruction of the form:

$$(\texttt{R}[a])\texttt{branch}\ T$$

gets replaced with the following pair of instructions:

$$\texttt{join S}[c] = \underline{\texttt{R}[a]}, \perp$$
$$(\texttt{R}[a])\texttt{branch}\ T$$

Prior to this transformation, $\texttt{S}[c]$ should be an unallocated security register that is not related to any state in the base ISA. Additionally, for every instruction which is control dependent on the branch in the original program, the binary translator adds $\texttt{S}[c]$ to the instruction's list of security operands.

If only this transformation were applied, the translated program would be equivalent to the naïve translation from static IFS to dynamic IFS discussed earlier.

```
      cmp.gt R[1] = R[2], R[3] // R[1] := R[2] > R[3]
      (R[1]) branch .end
      store [R[2]] = 7         // Mem[R[2]] := 7
.end: ...                      // Rest of program...
```

Figure 5. Code conditional variable store.

5.2.2. Handling Implicit Flows

Consider the attack (shown in Figure 3) on the naïve translation discussed earlier. Using the currently defined translation, at node X, the register containing the variable `c` would have the label $\underline{\texttt{a}}$ only if the path through node W is taken. As was described earlier, the variable `c` contains the complement of the variable `a` regardless of which path was traversed. Therefore, our translation needs to ensure that the variable `c` has the label $\underline{\texttt{a}}$ regardless of which path is traversed.

Ideally, the binary translator would insert an instruction on the direct path from node V to X to perform a label join restricting the label $\underline{\texttt{c}}$. Unfortunately, if the variable `c` is stored in memory (rather than a register), it may not be known where the variable `c` is stored. For example consider the code shown in Figure 5. In the figure, if the branch is taken, memory is modified at the location given by `R[2]`. If the branch is not taken, the value of `R[2]` must be *different* than the value of the register had the branch been taken. Therefore, inserting code to restrict the label $\underline{\texttt{Mem[R[2]]}}$ if the branch is not taken will modify the label of a *different* memory location than if the branch were taken. If `R[2]` is bimodal (i.e. it only takes on one of two values), then it will still be possible to distinguish between what path was taken by analyzing the appropriate location in memory.

To avoid this problem caused by memory indirection, instead of inserting an instruction along the not-taken path of a branch, the binary translator will append the security register defined by the branch to the list of security operands on all instructions that *potentially* use values defined by instructions control dependent on the branch. For example, recall the example Figure 4. The store instruction on line 10 in Figure 4(a) is control independent of the branch on line 8. However, in the secured program the store on line 12 is annotated with `S[3]`, the security register for the branch on line 9, since the store uses `R[2]` and there is a define of `R[2]` control dependent on the branch.

Conceptually, this strategy restricts a value's label when it is *used* rather than when it is *defined*. This strategy is safe since *all* instructions that could observe the result of a conditional variable assignment, see the data with a restricted label.

Deciding what instructions can observe values defined by other instructions can be easily accomplished for register based instructions using reaching-definitions analysis [1]. For memory load and store instructions more sophisticated memory dependence analysis is necessary. The literature describes various conservative alias analysis algorithms (i.e. algorithms which may identify false dependences, but will not omit any true dependence) which can be used to determine the set of store instructions which write to the same address as a particular load instruction reads [5, 6, 11, 33]. While these analyses operate at the source level, other analyses have been described which operate on program binaries [4]. These analyses are able to reconstruct a program's control-flow graph (in the presence of register indirect jumps) and subsequently perform pointer analysis. Notice that as the quality of memory dependence analysis improves, fewer false dependences will be observed during binary translation making the translated program less restrictive.

5.2.3. Handling Loops

The translation described thus far will work for acyclic code. However, if registers or memory locations are live across the back edge of a loop, the translation for branch instructions described earlier can potentially cause information leaks. This occurs because the security register defined as part of the translation may potentially be used by instructions after the back edge is crossed. If the branch instruction redefines it, then values computed under earlier conditions will be accessible under the new label stored into the security register. Since the new label is potentially less restrictive than the old label, this may cause information leaks. To avoid this potential leak, the security operand defined by each branch instruction should be:

$$\texttt{join S}[c] = \underline{\texttt{R}[a]}, \texttt{S}[c]$$

By defining the security operand before each branch as the join of the branch predicate and the previous value of the security operand, the security operand does not lose the information it previously contained. Therefore this definition will be secure even in the presence of loops. This instruction will cause the security operand `S[c]` to monotonically get more restrictive.

To avoid making security operands overly restrictive, each security operand annotated onto an instruction due to a conditional reaching definition, will be unique to that instruction. At each defining location, this security operand will be set to the label $\perp$. This transformation remains secure, since redefining a storage location destroys all information that could have been learned due to assignments that did *not* occur. Since this information is destroyed, we no longer need to remember the security label of that information. Redundant code introduced by using a unique security registers for security operands will be eliminated through compiler optimizations.

5.3. Security

While a formal proof of security is beyond the scope of this paper, this section will assert a definition of security and then sketch a proof of the soundness of the binary translation with respect to this definition.

Definition 1. *A program is secure if, for any threshold label and any two program inputs which are identical for all data values labeled with some label less restrictive than the threshold label, the program outputs are identical for all data values labeled with some label less restrictive than the threshold label.*

Intuitively, the definition states that a program is secure if nothing can be learned about confidential inputs by looking at public outputs. The binary translation guarantees this through its management of security operands. Consider the execution of the secured binary with the two program inputs used in the definition of a secure program. Any instruction which produces output will be run in both programs or will be control dependent on a label more restrictive than the threshold. The control dependence implies the output will carry a label more restrictive than threshold. If both programs execute the output instruction, the data values are equal or the data value is the result of an explicit or implicit flow of information from inputs with labels more restrictive than the threshold. Since the binary translator converts all implicit flows to explicit ones, and the architecture tracks explicit flows, the label of the data being output must be more restrictive than the threshold. Therefore, the translated program running on a RIFLE architecture is secure.

6. Evaluation

To guide the design of an instantiation of the abstract framework just described, we measure the properties of unmodified assembly files from a type-unsafe language. Itanium 2 assembly files were obtained from the IMPACT C compiler using standard optimizations. These assembly files were then annotated using the binary translation techniques described earlier. Rather than using an arbitrary number of security operands per instruction, the ISA was extended to support only one security operand. Additional join instructions were inserted to combine security operands on instructions that initially had more than one. To reduce the number of inserted join instructions, we performed classical compiler optimizations such as constant folding, constant propagation, copy propagation, common sub-expression elimination, and dead code elimination.

We evaluated the Unix utility `wc`, `thttpd`, and several C benchmarks from SPEC Integer 2000 and MediaBench benchmark suites for correctness, conservatism, and performance.

6.1. Verifying Program Security

To evaluate program correctness, all benchmarks were run on our IA-64 functional simulator augmented with the RIFLE architectural extensions. Each program processes multiple files, each with a different security label. Rather than defining labels for output streams and enforcing a particular policy, output files were annotated with labels at the byte level. Program execution correctness was checked using real hardware. Information-flow correctness was verified by manual inspection.

During initial runs of the programs, we noticed that labels became extremely restrictive. Analysis revealed that this was due to the stack pointer becoming restricted upon the execution of a branch guarding procedure calls. Using additional analysis to show that the stack pointer is the same after the execution of either path (with the exception of program termination dealt with as described earlier), the more aggressive, yet still conservative, system produced labels in the output file as expected. For example, when the `wc` utility was run on a variety of files each marked with different labels, the output describing each file was marked with the file's label, while output describing summary data about all files was marked with the join of all the files' labels.

We will present an analysis of two applications, PGP and thttpd, to illustrate how information-flow security can be used in practice, show that RIFLE is not overly restrictive, demonstrate potential security vulnerabilities, and identify areas of future research.

6.1.1. PGP

Pretty Good Privacy (PGP) is an application for public key cryptography. PGP maintains a public-key ring and private-key ring. Users can request that a file be encrypted, decrypted, or digitally signed using one of the keys in either ring. To test PGP, we created a pair of key rings and labeled each key in the rings with a unique label. We also labeled the input file with a unique label. We then ran PGP with options so that it would encrypt and digitally sign a plain text file. We examined the labels of the resulting output file. We expected the output file to labeled with the join of the input file label, the label of the public key used for encryption, and the label of the private key used for the digital signature. Initial experiments showed the output file being labeled with the labels of not only the key used for encryption and signature, but also of all the keys that appeared before them in the keyrings. Examination of the code revealed this to be the correct behavior. The code that read keys from the keyring had to scan over the keyring until it found the appropriate key. Each encountered key was checked to see if it was the requested key, and if so, exited the loop. This loop structure forced subsequent loop iterations to be control dependent on the exit condition of prior iterations.

To overcome this label creep, we relabeled our keyrings making all key identifiers and key sizes share a single unique label. Key sizes were relabeled since keys are variable size and the size of the current key is needed to locate the subsequent key. With these modifications, we reran PGP and the new output possessed the expected labeling.

6.1.2. thttpd

thttpd is a tiny web server application. For the experiment we configured the web server with two documents. Each document was password protected. We exercised the web server with four requests, two per file. One request per file was submitted with the correct password, the other request per file was submitted with an incorrect password. Each request and document was uniquely labeled. The password file, which consists of user names followed by fixed length passwords, had two unique labels per line; the user name was labeled differently from the password. As expected, running thttpd revealed that the responses to the unauthorized requests were labeled with all the user name labels in the password file (since the whole file was scanned) and the request label. For the authorized requests, the responses were labeled with the the label of the request, the label of the document, the labels of user names up to the authorized user's name, and the label of the password. While this accurately characterizes the information that was examined to produce the response, it may mislead a server administrator to believe that responses are leaking passwords. In actuality, only one bit of information about the password (whether the password was correct) is leaked. Future work will examine how to inform users of how much information is leaked and study methodologies to allow declassification.

6.2. Performance

To evaluate the performance cost of implementing RIFLE, we measured the number of overlapping security register live ranges prior to register allocation and the execution time of secured programs relative to the original unsecured programs. All performance measurements were run on a validated cycle-accurate Itanium 2 model with RIFLE extensions built in the Liberty Simulation Environment [32].

Figure 6 shows the number of overlapping security register live ranges. For a given number of overlapping live ranges, the graph shows the percentage of execution time spent in functions with fewer overlapping live ranges. As is shown in the graph, for most benchmarks more than 70% of the execution time is spent in functions with fewer than 100 security registers live. Future work on additional compiler optimizations to reduce register live ranges may have the potential eliminate register spill and fill code, and therefore to significantly boost the performance of the system.

The execution speed of the secured programs, relative to the base is shown in Figure 7. The first bar for each benchmark shows the secured programs' runtime normalized to the runtime of the unsecured program running on a machine with all of the data caches for Itanium 2 duplicated to store security labels. The second bar for each benchmark shows the normalized runtime when the benchmarks were run on a model whose data cache was partitioned into two equally sized pieces whose total size is equal to that of the Itanium 2. As the graphs indicate, the runtime performance penalty for the system is relatively low. The additional security instructions do not incur a significant performance penalty since these instructions are independent of the original program instructions and therefore can be executed in parallel, if sufficient resources exist. Additional compiler optimizations will reduce this penalty even further.

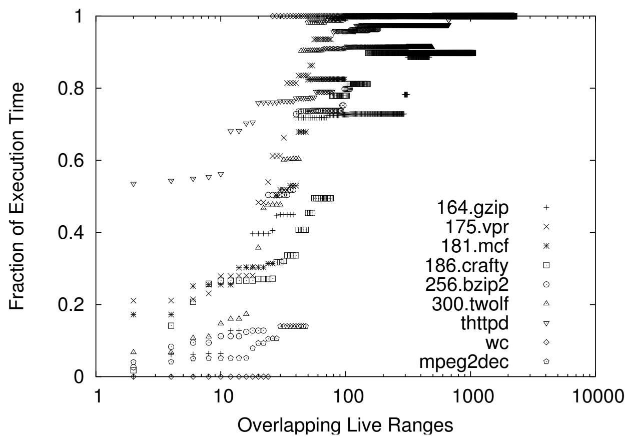


Figure 6. Overlapping security register live ranges

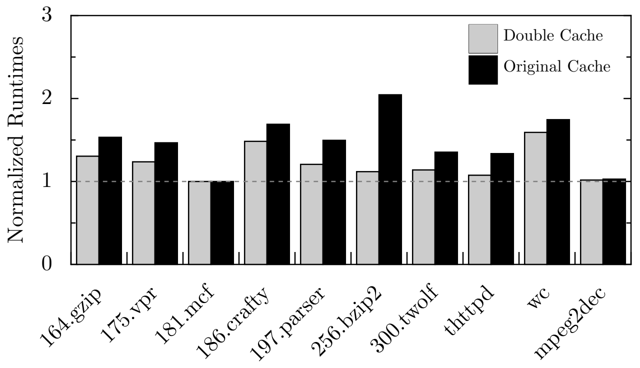


Figure 7. Performance of secured programs

7. Conclusion

Information-flow security allows users to maintain control of their data while still permitting untrusted applications to access the data to perform useful computation. Typically, these information flow security schemes are built as extensions to type-safe languages where security is verified statically by the compiler. In such schemes, control over policy decisions and policy enforcement is in the hands of the programmer. This approach has been the main research focus because static information flow systems were believed to be more secure than dynamic systems.

In this paper, we present RIFLE, a runtime information flow mechanism that is as secure as existing static schemes. However, unlike static schemes, security decisions are in the hands of the user since RIFLE works with all programs (not just those written in special languages) and policy decisions are left to the user not to the programmer. We implemented RIFLE and demonstrate the performance cost for security is reasonable. Our implementation also demonstrates that RIFLE successfully tracks information flow and can be effectively used by end-users to manage their confidential data.

Acknowledgments

We thank David Walker, John Sias, and the entire Liberty Research Group for their support during this work. This work has been supported by the National Science Foundation. Opinions, findings, conclusions, and recommendations expressed throughout this work are not necessarily the views of the National Science Foundation.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 65. IEEE Computer Society, 1997.

[3] W. A. Arbaugh, A. D. Keromytis, D. J. Farber, and J. M. Smith. Automated recovery in a secure bootstrap process. In *Symposium on Network ands Distributed System Security (SNDSS)*, pages 155–167, 1998.

[4] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the 13th International Conference on Compiler Construction*, pages 5–23, 2004.

[5] B.-C. Cheng and W. W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

[6] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 142–153. IEEE Computer Society, 1998.

[7] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[8] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[9] J. S. Fenton. *Information Protection Systems*. PhD thesis, University of Cambridge, Cambridge, England, 1973.

[10] S. N. Foley. A taxonomy for information flow policies and models. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 98–108, 1991.

[11] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 47–58. ACM Press, 2001.

[12] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, *Proceedings of the 25th ACM Symposium on Principles of Programming Languages*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[13] Intel Corporation. Web site: http://www.intel.com/technology/security/downloads/LT_Arch_Overview.pdf, February 2004.

[14] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, 1971.

[15] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[16] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[17] Lavasoft. Web site: http://www.lavasoftusa.com, January 2004.

[18] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 178–192. ACM Press, 2003.

[19] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[20] Microsoft Corporation. Web site: http://msdn.microsoft.com/security/productinfo/ngscb, January 2004.

[21] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[22] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 129–142. ACM Press, 1997.

[23] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[24] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Langauges*, pages 106–119, Paris, Jan. 1997.

[25] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, 1990.

[26] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

[27] H. J. Saal and I. Gat. A hardware architecture for controlling information flow. In *Proceedings of the 5th annual symposium on Computer architecture*, pages 73–77. ACM Press, 1978.

[28] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[29] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171. ACM Press, 2003.

[30] Trusted Computing Platform Alliance. Web site: http://www.trustedcomputing.org/, February 2004.

[31] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 179–193. IEEE Computer Society, 2004.

[32] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO)*, pages 271–282, November 2002.

[33] R. P. Wilson and M. S. Lam. Effective context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[34] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[35] J. C. Wray. An analysis of covert timing channels. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 1991.