Boat: Automatic Generation of Policy Code for Flume

Bill Harris
wrharris@cs.wisc.edu

Matt Elder
elder@cs.wisc.edu

May 16, 2008

Abstract

Distributed Information Flow Control (DIFC) is a system-level security mechanism that can simplify program security but requires that a programmer supply policy code in any programs it helps to secure. Writing this policy code is not a simple matter, and is thus likely to consume significant programmer attention and introduce bugs.

Thus do we implement Boat, a means to simplify DIFC programming. Boat reads a program's C source and a policy specification, generates from these a system of constraints, and solves those constraints to generate instrumentation for the original C source. Thus, Boat augments the original program with added code to ensure that its DIFC policy matches the programmer's specification.

1 Introduction

Security-conscious systems and applications implement information flow control systems as mechanisms for ensuring that sensitive data does not reach undesired parts of the system or the outside world. Traditional Mandatory Access Control (MAC) [8] implements security policies directly in the system. MAC is effective but inflexible, as only a system administrator can define a security policy for the system to hold.

Distributed Information Flow Control (DIFC) systems [1, 10, 3] provide an attractive alternative, as they permit each application to define and enforce its own security policies by detailing what data is sensitive and how that data may be propagated. Such systems are flexible and powerful, but complex. For every system object such as a process or file, the system maintains a security *label* indicating what data the process may receive and transmit, and privileges indicating how the process may adjust its label. Programmers must be ever aware of these mechanisms to ensure that their code is both functionally correct and secure. Even the simplest policies can quickly become difficult to program and it is difficult to ensure that such code matches the programmers high-level desired policy.

To solve this difficulty we propose *Boat*, a high-level specification language for security policies and a supporting source-to-source compiler. A Boat specification consists of a policy specification file and a small number of annotations in the source program. The programmer directly specifies the functionality and security requirements for their program in our high level language. Boat then analyzes the program and specification, represents both as constraints over Flume labels, and generates Flume code that enforces the desired policy.

We have implemented Boat as an extension to CIL [5], a C parser and front-end. Our extension takes as input both the source programs to be analyzed and the specification, generates and solves constraints, and carries out a source-to-source translation. This translation instruments the code with the appropriate calls to the Boat runtime library, ensuring that it follows the specified policy. We have used Boat to analyze FlumeWiki, which is itself a version of MoinMoin Wiki [4]. We are able to succinctly express a desired policy of FlumeWiki in Boat's specification language and to use Boat to generate all necessary code for setting labels before all relevant calls.

2 Related Work

Numerous systems have implemented DIFC at the granularity of processes and system objects. Asbestos [1] maintains two labels for every process: a receive label R and a send label S. A label is a set of *compartments*; a compartment can be any value in the set $[0, \ldots, 3]$. An IPC send from process p to q succeeds if and only if p_S is a subset of q_R and every compartment value in p_S is less than or equal to the corresponding compartment value in q_R. If the send succeeds, q_S is then automatically updated by the system to be "at least as high" as p_S. In this way, receive labels encode policies while send labels encode current states.

Flume [3] simplifies and extends Asbestos. It simplifies by maintaining for every process a single label and reduces IPC checks to simple set inclusion, not individual value comparison. It extends Asbestos by allowing the application to maintain for each IPC channel an endpoint. The label of an endpoint may differ (within constraints) from the label of the process, and Flume makes IPC checks on the labels of endpoints. Flume also maintains an *integrity label* for every entity. Integrity labels allow applications to succinctly specify policies such as *Process p can only take input that originated from these entities:* $\ldots$. More practically, Flume is implemented as a user-space process accompanied with a Linux Security Module, allowing it to work atop existing systems. In the current version of Boat, we focus on generating security labels and enforce that the security labels of all endpoints always equal that of the process. However, we plan to employ integrity labels and endpoints in future work.

Dataflow analysis [6] is a standard static analysis technique that we employ for reasoning about relevant program points.

In [7], Rehof and Mogenson define *definite* systems of constraints over lattice elements, present an algorithm for solving all definite constraint systems, prove that any system that is not definite is NP-hard to solve, and give necessary and sufficient conditions for a system to be definite. We apply their work to show that the constraint systems generated by our analysis are not definite and extend their algorithm with heuristics to derive an algorithm that runs reasonably well on our test cases. Constraint solving has been used as the basis for many program analyses; for example, subtyping constraints and lattice inequalities are used to infer type qualifiers in CQual [2].

3 Flume

Boat defines an annotation language that accompanies C programs. Programmers express their high-level security policy as a specification file and a few small program annotations. Boat then generates calls to the Flume API that enforce the policy at runtime. Thus, some background in Flume is necessary to discuss Boat.

Flume is a complete system for enforcing DIFC. It is a primarily user-space program (called the *reference monitor*) that interposes itself on communication between a security-critical program and the environment, including other processes and system resources. We here summarize key Flume functionality; the interested reader should examine [3] for more details.

3.1 Labels and Tags

Flume maintains for every process and file a set of *tags*, called a *label*, along with information on what tags a process may add and remove from its label. A process may request from Flume a fresh tag, which is simply a random unsigned long integer. Flume grants the tag to the process along with privileges to add and remove the tag from its label. The process may then add the tag to the labels of processes, and possibly grant them the privilege to add or remove the tag from their own labels.

3.2 Pipes

The primary aim of Flume is to allow applications to enforce security policies as they see fit. To do so, it cannot merely provide facilities for manipulating tags. Flume must ensure that these tags are respected during IPC. To do this, Flume maintains a notion of what processes are *confined* versus which are *confined*. Flume allows an unconfined process to communicate with the outside world as it wishes, but assumes that any data sent to an unconfined process is effectively leaked outside of the system. In contrast, Flume carefully monitors all IPC of a confined process. It employs a kernel module written in the LSM [9] framework to monitor all relevant system calls and prevent any communication that sidesteps the Flume API. A confined process P may thus communicate with its child process Q over a pipe as follows:

1. P calls `flume_socketpair(myfd, theirtok, ...)` to obtain a file descriptor (`myfd`) and a token (`theirtok`). Flume creates a label for `myfd`; by default, the label is initialized to the current label of the process. If the file is opened for reading, then $S_{fd} \subseteq S_p$ must hold, if it is open for writing, $S_p \subseteq S_{fd}$ must hold, and if it is open for both, then $S_{fd} = S_p$ must hold.

2. Flume provides `spawn`, its wrapper to the traditional UNIX `fork` call. `spawn` takes as one of its arguments a vector of descriptor tokens to grant to the child at its birth. P thus passes the token to Q when it spawns it.

3. Q calls `flume_claim_token` to exchange its token for the file descriptor that is the other end of the pipe held by P.

4. The parent and child may then communicate. However, the file descriptors they use now actually point to the Flume reference monitor. When P with descriptor $fd1$ attempts a write to Q with $fd2$, Flume will check if
$$S_{fd1} \subseteq S_{fd2}$$
holds. If so, the message is passed. If not, the message is buffered but will be issued if and when the security labels satisfy the above condition. Notably, unless the labels are equal, Flume cannot return an accurate error code, as this could be used illegally by P to glean information about Q when the labels dictate that this should not be allowed. Thus, it is the programmers responsibility to be sure that the corresponding labels exhibit the correct relation, which they must accomplish without reliable dynamic feedback from the resource monitor.

3.3 Files

Flume includes a file system monitor that allows tags to persist on files. When the Flume reference monitor detects a file system request like `open` or `mkdir`, it forwards the request to a file system monitor. The file system monitor stores persistent label information about each file in the system; when access to a file is requested, it checks the requester's label against both the file's label and the requested action to determine if it will allow the access.

4 Constraint Generation

Flume thus reduces its decisions to evaluating subset expressions over labels. We propose that a programmer using the Flume system will have two desires that can thus also be expressed as subset constraints over labels:

Security constraints: At some points in the program, the programmer expects that data from one system object cannot reach another system object.

Functionality constraints: At some points in the program, the programmer expects data from one system object to reach another system object without being blocked by Flume.

The programmer must ensure that their code adheres to these constraints along with the structural requirements of a valid Flume program. One of our key contributions is the notion of a static representation and solution of these constraints.

4.1 Intraprocess Flow Constraints

The intraprocess structure of a single program implies a set of constraints that encode how the process may change its own labels. At any point of execution, a process has its current security label S_p, a label S^+ containing all tags that it may add to S_p, and a label S^- containing all tags that it my remove from S_p. Flume's structure implies that, between two points of interest a and b, the labels at b can be no *greater* than their value at a augmented by all labels in S^+ and no *less* their value at a with all labels in S^- taken away.

More formally, we know a list of functions relevant for Flume communication such as `spawn` and `write`. We generate a control-flow graph (CFG) of the program. For simplicity, we assume that every node in the graph corresponds to a single instruction. For a program point p in the CFG that contains a call to a relevant function, we generate a label variable S_p that represents the security label of the process when it reaches this program point. Using a standard dataflow analysis, we compute the set $\bar{p}$ for every relevant program point p. The set $\bar{p}$ is the set of all relevant program points that can be the last relevant point visited before visiting p on some execution path. For all

$q \in \bar{p}$, we then generate the following two constraints:

$$S_q \subseteq S_p \cup S^+$$
$$S_p \subseteq S_q \cup S^-$$

These constraints encode the relationship between the labels at a program point and the labels at its predecessors.

4.2 Specification Constraints

To use Boat, the programmer must annotate each `spawn` program point with a unique *family name*, and name the destination family for each `write`. To permit this, the first argument to `spawn` and `write` is a literal string containing the required family name. That is, the spawn corresponding to the process family "Foo" looks like `spawn("Foo", arglist...)`, and a write to a process in the family "Foo" looks like `write("Foo", arglist...)`.

Boat uses these annotations to link the source code to the programmer-defined specification file. A Boat specification has two parts: a list of process families, and a set of family relationship statements. The list of process families is presumed to contain all used families and to be ordered by trustworthiness. Thus, Boat will prefer to grant privileges to families listed earlier in the list.

Each relationship statement is: a family name, an operator, an optional family modifier, and another family name. The possible operators are `:`, `->`, `<-`, and `<->`; an arrow may be preceded by a `!` for negation. The statement `Foo:Bar` means that processes in the family Foo may spawn processes in the family Bar. That is, Foo is Bar's parent family.[1] An arrow operator from a source family to a target family means that a source process may write to a target process; `Foo<->Bar` is a contraction of `Foo->Bar; Foo<-Bar`. A negated arrow means that a source process may not write to a destination process; Flume will disallow their communication.

The right-hand family name may be preceded by a family modifier: the child operator `*`, or the niece operator `\`. So, the statement `Foo->*Bar` means that a Foo process can write to any Bar process that it spawned, and the statement `Foo!->\Bar` means that a Foo process cannot write to any Bar process that it did not spawn. Finally, the statement `Foo->Foo` means that any Foo process can write to any other Foo process, `Foo!->Foo` means that no Foo process can write to any other Foo process.

[1]This probably should be determined directly from the source code. We mean to add this feature soon. [[Add this to Future Work.]]

Between parsing the policy specification and producing policy constraints lies R-process generation. Representative processes, or *R-processes*, are the pseudo-processes on which we state our constraint satisfaction problem. R-processes are made necessary by the generally unbounded nature of each process family: any single spawn point may spawn an arbitrarily large number of processes, so not all distinct processes can be directly represented. At a high level, R-processes provide a way to embed some form of universal quantification in our constraint satisfaction system.

We build a forest of R-processes such that:

1. There exists in the forest at least one R-process for each process family, and

2. For each parenthood relation `Parent:Child` in the specification, every R-process of family Parent has two child R-processes of family Child.

We accomplish this by first finding each process family which will be a root of some tree in this forest; that is, any family which is not the child of any other family. For each root family, we generate one R-process; we generate other R-processes via simple recursive decent down parent-child relationships. For example, Figure 1 shows the tree generated from the parenthood relations `Foo:Bar`, `Bar:Baz`, and `Foo:Frotz`.

So, each R-process knows its process family name, its parent R-process (unless it is a root), and its child R-processes (unless it is a leaf). We can now piece together the entire constraint system:

1. For each process family Foo, and for each R-process P in Foo, generate a copy of the Foo family intraprocess flow constraints specific to P.

2. For each pair of R-processes P and Q that the specification allows P to write to Q, generate the constraint

   $$S_p \subseteq S_q$$

   for each program point $p \in P$ and $q \in Q$ where p might write to Q and q might read from P.

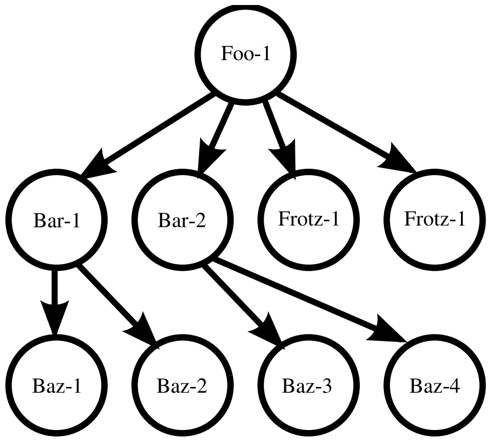

Figure 1: A forest of R-processes, with links between parents and children.

3. For each pair of R-processes P and Q that the specification require P not to write to Q, generate the constraint
$$S_p \not\subseteq S_q$$
for every pair of program points $p \in P$ and $q \in Q$.

5 Constraint Solution

We have now generated the full set of constraints that we would like to solve. The variables in each constraint are sets of security tags, at a particular program point in a particular R-process. Each constraint is in one of the following forms: $A \subseteq B$, $A \subseteq B \cup C$, or $A \not\subseteq B$.

Rehof and Mogensen have given necessary and sufficient conditions for a finite semilattice constraint satisfaction problem to be NP-complete, along with an algorithm (Algorithm D) to solve in polynomial time any such problem that is not NP-complete. [7] Among constraint satisfaction problems on finite sets, a problem is NP-complete if its relations are not closed under set intersection. For example, the constraint form $A \subseteq B$ is closed under set intersection, because for any finite sets A, B, C, and D, if $A \subseteq B$ and $C \subseteq D$, then $A \cap C \subseteq B \cap D$.

However, the other forms in our constraint system are not closed under set intersection, as illustrated by the following examples:

$$\begin{aligned} & \{1\} \subseteq \emptyset \cup \{1\} \text{ and } \{1\} \subseteq \{1\} \cup \emptyset, \\ \text{but } & \{1\} \cap \{1\} \not\subseteq (\emptyset \cap \{1\}) \cup (\{1\} \cap \emptyset)\,. \\ & \{1\} \not\subseteq \emptyset \text{ and } \{2\} \not\subseteq \emptyset, \\ \text{but } & (\{1\} \cap \{2\}) \subseteq \emptyset \end{aligned}$$

Thus, our general constraint system is NP-complete, and the vanilla Rehof-Mogensen algorithm is not guaranteed to work on our system.

5.1 Solution Algorithm

Fortunately, we can apply knowledge specific to our application domain to arrive at a solution efficiently. We first give an overview of Algorithm D presented in [7]. D solves a set of set constraints of the form $E \subseteq s$ where E is an arbitrary set expression and s is a set variable or constant. D employs a simple worklist approach: it initially maps all variables to the empty set value and maintains a worklist of all constraints left to be solved. In a given iteration, it selects a constraint from the list and evaluates both sides of the constraint. If the value of E is a subset of the value of s, then the constraint is already satisfied and D moves on to the next constraint. If the constraint is not satisfied, then the value bound to s is raised to be equal to the value for E and all constraints for which s appears in the left hand side are added to the worklist. D continues in this fashion until it reaches either an empty worklist (in which case it returns its solution) or a contradiction (in which case it returns failure).

Our algorithm maintains the same basic form as D, but our constraints lie outside of the domain of constraints that D can handle. As we iterate over our worklist, we do nothing if the constraint is already satisfied, as in D. If a constraint is not already satisfied, we implement different behaviors in different cases:

$A \subseteq B$ These constraints can be solved by D; we apply the same approach.

$A \subseteq B \cup C$ We need only to raise one of B and C. For all constraints of this form, exactly one of either B or C is a variable denoting representative tags at a program point, and the other is a variable denoting a set of representative privileges. In the interest of granting programs least privilege, we prefer to raise the value of the variable representing the program point.

$A \not\subseteq B$ For such constraints, recall that each variable represents a label, which is simply a set of tags. The application is free to generate new tags at will. So, for a constraint of this form we can simply create a new representative tag a, add a to the current solution for A, and note that a must never be allowed to be in B.

6 Instrumentation

If the constraint solver terminates with a solution, then we have a mapping from label variables to sets of representative tags, or *R-tags*. We must now generate Flume code that corresponds to such a solution. To do so, we partition the R-tags into two sets:

Per-child tags are R-tags that are held in exactly one child variable in a family. We maintain a set of such per-child tags for each family.

One-time tags are all other R-tags, which need only be generated once. We maintain a global set of one-time tags.

Next, we generate run-time code that maintains a mapping from R-tags to sets of Flume tags. It works as follows:

- At initialization, the code allocates each one-time tag and maps its R-tag to a singleton set.
- At an invocation of the form `spawn("Foo", arglist)`, the code notes all per-child R-tags of `Foo`. For each per-child R-tag, it allocates a new Flume tag T, adds T to the new label that will serve as the child's initial label, and adds T to the set of tags to which the parent's R-tag maps. Boat replaces the annotated call with the normal Flume call `spawn(arglist)`.
- At any other invocation of a relevant function, say one of the form `write("Foo", arglist)`, the code examines the set of R-tags that must be in the label at this point, sets the label to hold all of the corresponding Flume tags, and then calls `write(arglist)`.

Program module	LOC
program analysis (boat.ml)	714
constraint solver (boat.ml)	102
instrument.ml	273
spec.ml	205

Table 1: Lines of OCaml code per module

```
./bin/cilly --doboat
--boat-spec=boat_wiki.boat cgilaunch.c
```

Figure 2: The command for applying Boat to FlumeWiki

7 Implementation

We have implemented our system Boat as an extension of the CIL parser and front end for C. The implementation is organized into the following modules:

- A program analysis module that carries out the intraprocess analysis on all given source files and generates intraprocess constraints.
- A specification parser that parses an input specification file and generates specification constraints.
- A constraint solver that maps every variable in the constraints to a set of tags.
- An instrumenter that rewrites the code such that the labels at each program point match the values determined by the constraint solver. The implementation of this module is at the time of publication only partially complete, generating initialization and pre-`write` code. We instrument remaining code, such as pre-`spawn` code, by hand.

8 Application

We applied Boat to FlumeWiki (described in [3]), a Flume implementation of the wiki MoinMoin. The high-level organization of the wiki is given in Figure 3. As described in [3], when an HTTP request is received over the network, a new `httpd` process is spawned, which in turn launches an instance of `wikilaunch`. `wikilaunch`

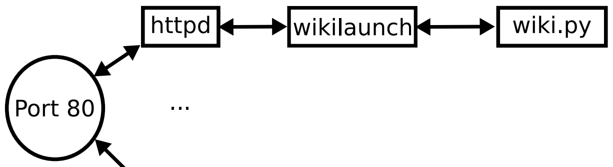

Figure 3: Structure of FlumeWiki

```
rc = flume_socketpair
      (&input, &fdhandles->val[0],
       ...);

/* setup CGI's labelset */
S_label = label_alloc (1);
rc = label_set (S_label, 0,
                S_handle);
labs = labelset_alloc ();
rc = labelset_set_S(labs,
                    S_label);

/* spawn the CGI */
rc = flume_spawn_legacy
      (labs, fdhandles, ...);
/* send form information to cgi */
if (cgl_form_len ()) {
    rc = write (input, ...);
}
```

Figure 4: cgilaunch.c

```
wikilaunch wiki;
wikilaunch:wiki;

wikilaunch -> wiki;
wiki !<-> wiki;
```

Figure 5: The specification for BoatWiki

```
/* setup CGI's labelset */
rc = flume_socketpair
      (&input, &fdhandles->val[0],
       ...);
rc = flume_spawn_legacy
      ("wiki", fdhandles, ...);

/* send form information to cgi */
if (cgl_form_len ()) {
  rc = write ("wiki", input, ...);
}
```

Figure 6: cgilaunch.c with Boat annotations.

is a small module that maintains the entire security policy of MoinMoin. It was created for [3] by factoring all security-aware code from the considerably large (approx. 91,000 lines of code) and complicated MoinMoin code into a single security critical module. Each instance of `wikilaunch` launches its own unconfined child process `wiki.py` and communicates with it.

A simplified version of the relevant code snippet from `cgilaunch.c` (the actual home of the `wikilaunch` code) is included in Figure 4. This code is located inside of the function `do_fork()`, which is called whenever the server wishes to create a cgi process to handle a request. At a high level, the code creates a socket pair. The second block of code creates a new tag, adds it to a new label, and passes that label as the initial security label in the subsequent call to `spawn`. `cgilaunch.c` may then later write to its child processes.

FlumeWiki is designed to adhere to multiple security policies. One is that the data from no instance of `wiki.py` should be able to reach another instance of `wiki.py`. We focus on the specification of this policy. We can specify this policy in the boat specification file found in Figure 5 and the added annotations given in Figure 6 that relate the process families in the specification to the spawn sites in the code. The specification directly corresponds to our desired security policy. The first line declares the process family names. The second declares that processes in `wikilaunch` are parents of those in `wiki`. The third declares that `wikilaunch` should be able to communicate with `wiki`. The last line states that processes in the `wiki` family should not be able to talk to each other.

9 Future Work and Conclusions

Given the complexity of properly securing a system, even with powerful underlying systems, tools like Boat are probably prerequisite to the common occurrence of truly secure software. Of course, in the current rough state of Boat (and, frankly, Flume), these tools are inadequate for production use. They require much more polish and design.

For example, the current version of Boat requires an uncomfortable amount of repeated information. A polished version of Boat would not require parenthood statements in its specification file, as those relationships can be unearthed from the annotated source code. Clever program analysis could even do away with the family annotations on every `write` by tracking the values of file handles and file handle tokens.

Boat doesn't yet address integrity tags or the Flume file system monitor. Generating constraints for integrity tags is not particularly difficult, but finding desirable ways to solve integrity constraints and Boat's current constraints might.

Our application of Boat involved a fairly large program, but it involves only a small set of process families and specification statements. It would be worthwhile to test and improve the scalability of Boat. In particular, we know that the specification constraint generator can produce a number of R-processes exponential in the number of process families, and that this leads to an exponential increase in the size of the set of constraints. If Boat were to support a very large ecology of software, these scalability issues would be a frequent hassle. Solving this threat requires an improved algorithm. The constraint solver itself runs in polynomial time on the size of its inputs, but it is unclear that its actual performance is optimal. The algorithm employed by Boat is a nice hack, but it probably leaves room for improvement.

We judge our method for converting R-tags to Flume tags to be another nifty hack, but it's rather reliant on limitations of the specification language. Probably, any generalizations in the specification language that grant greater flexibility to expressible policies will break this algorithm. We may require a more general solution.

Although this work is preliminary, we have seen that the analysis in Boat is capable of instrumenting large, complex programs with high-level policies. The construction of such systems appears to be quite difficult in general, but useful tools can be carved out of this space. The sort of tools programmers would need to comfortably work with DIFC appear to be feasible.

References

[1] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39(5):17–30, 2005.

[2] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 192–203, New York, NY, USA, 1999. ACM.

[3] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 321–334, New York, NY, USA, 2007. ACM.

[4] MoinMoin. The moinmoin wiki engine, December 2006.

[5] G. Necula, S. McPeak, S. Rahul, and W. Wimer. Cil: Intermediate language and tools for analysis and transformation of c programs, 2002.

[6] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999.

[7] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semilattices. In *SAS '96: Proceedings of the Third International Symposium on*

Static Analysis, pages 285–300, London, UK, 1996. Springer-Verlag.

[8] J. H. Saltzer and M. D. Schoeder. The protection of information in computer systems. In *Proc. IEEE*, volume 63, pages 1278–1308, September 1975.

[9] ”C. Wright, C. Cowan, J. Morris, and S. Smalley G. Kroah-Hartman. Linux security modules: general security support for the linux kernel. In *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pages 213–226, 2003.

[10] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI ’06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.