Secure Dynamic Program Repartitioning

Rene R. Hansen and Christian W. Probst
Informatics and Mathematical Modelling
Technical University of Denmark
2800 Kongens Lyngby, Denmark
{rrh,probst}@imm.dtu.dk

July 16, 2005

Abstract

Secure program partitioning has been introduced as a language-based technique to allow the distribution of data and computation across mutually untrusted hosts, while at the same time guaranteeing the protection of confidential data. Programs that have been annotated with security types are automatically partitioned by the compiler. The main drawback in this setting is that both the trust hierarchy and the set of hosts are fixed once the program has been partitioned. This paper suggests an enhanced version of the partitioning framework, where the trust relation still remains fixed, but the partitioning compiler becomes *a part of the network* and can recompile applications, thus allowing hosts to enter or leave the framework. We contend that this setting is superior to static partitioning, since it allows redistribution of data and computations. This is especially beneficial if the new host allows data and computations to better fulfill the trust requirements of the users. Erasure Policies ensure that the original host of the redistributed data or computation does not store the data any longer.

Keywords: Secure Program Partitioning, Erasure Policies, Distributed Computation.

1 Introduction

Secure Program Partitioning [ZZNM2001, ZZNM2002] has been introduced as a language-based technique for distributing confidential data and computation across a distributed system of mutually untrusted hosts. The program to be distributed is annotated with security types that constrain permissible information flow. The resulting confidentiality and integrity policies are used to guide the partitioning across the network. The resulting communicating sub-programs not only implement the original program, but at the same time satisfy all security requirements of principals, including trust relations to other principals as well as hosts. The results reported in [ZZNM2001, ZZNM2002] with respect to the performance of the distributed code suggest that this is a feasible way to obtain secure distributed computation.

The main drawback in Secure Program Partitioning (SPP) is that the framework contains two fixed components—the *trust relation* and the *hosts in the network*. While we contend that a static trust relation essentially is necessary to ensure system integrity, the second restriction not only seems to be superfluous, but also hinders the system from moving data or computations to newly available hosts that might allow a better fulfillment of principal's requirements. Thus, by allowing the partitioning to be adjusted after a host joins or leaves the network, the overall trust of the principals in the partitioning and thus in the security of the distributed system can be increased.

After data and computations have been redistributed in the enhanced network, in an ideal world hosts should not store any references to items that have been moved to another host. The recently introduced mechanism of Erasure Policies [CM2005] allows to design systems where programmers annotate their data with policies that describe exactly this kind of behavior. Erasure Policies state explicit erasure and declassification requirements.

The contributions of this paper are:

- We extend the framework for Secure Program Partitioning to allow hosts to enter and (under certain conditions) leave the network.
- By adding Erasure Policies to SPP, the extended system assures that information is erased or made inaccessible after a node has left the network.

The rest of this paper is structured as follows. Section 2 gives an overview of related work, including Secure Program Partitioning and Erasure Policies. This is followed in Section 3 by the description of our proposed framework for dynamic repartitioning in the case of hosts entering the network. The emphasis here is on making the partitioning an *active* component of the framework, as well as using Erasure Policies to ensure that hosts make inaccessible any data that has been redistributed to another host. Section 5 concludes the paper with an overview of future work.

2 Related Work

This section gives an overview of Secure Program Partitioning [ZZNM2001, ZZNM2002] and Erasure Policies [CM2005], which form the foundation for the framework described in the rest of this paper. This Section largely follows the description in the cited papers.

2.1 Secure Program Partitioning

Information-flow policies have been used for specifying confidentiality and integrity requirements. Their success is mostly based on the ability to specify how information may be used in the system as opposed to which principals may access or modify the data. Security-typed languages [Sch2000, ABHR1999, HR1998, Mye1999, PC2000, SV1998, VSI1996, ZM2001] have been used to implement these policies in programming languages. By annotating data in programs, the programmer explicitly specifies how the flow of information allowed by the

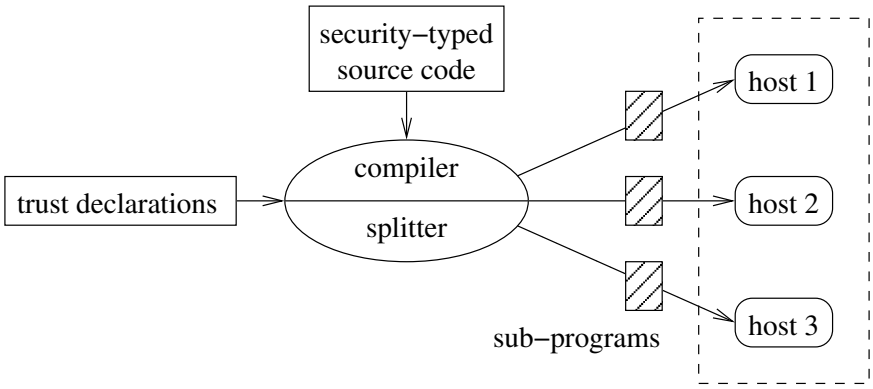

Figure 1: Structure of the Secure Program Partitioning Framework as introduced by [ZZNM2001, ZZNM2002].

language semantics should be constrained. The benefit of these explicit annotations is that programs that violate the restrictions will be rejected either during compilation or during execution. Thus, the program itself does not have to be trusted—instead, only the reused components (compiler and run-time system) must be trusted.

The benefit of SPP is that participants do not need to fully trust each others hosts to enable the distributed execution of a program dealing with data of the principals. As Figure 1 depicts, in the original framework introduced by [ZZNM2001], the compiler receives two inputs—the program source code, which uses a security-typed language, and the trust declarations of all participants. These declarations state each principal's trust in hosts and other principals. They are used to guide the compilation and splitting of the security-typed program into sub-programs that are executed on (some of) the hosts in the network. By communicating, these sub-programs perform the same computation as the original program, however, the splitter ensures that all trust and security policies are fulfilled. As the authors state in [ZZNM2001], the splitter ensures that if a host h is subverted, only the confidentiality or integrity of data owned by principals that trust h is threatened.

The main component in the programming model used in SPP is the *principal*, who can express confidentiality or integrity concerns with respect to data. Principals can be named in information-flow policies and also define the authority possessed by the program being executed. Security labels [ML2000] express confidentiality policies on data. A label $l_1 = \{o : r_1, r_2, \cdots, r_n\}$ means that data labeled with l_1 is owned by a principal o and that o permits readers $r_1, \cdots, r_n$ (and o) to read the data. Data can also have multiple owners, each expressing its concerns with respect to the data. E.g., $l_2 = \{o_1 : r_1, r_2; o_2 : r_2, r_3;\}$ expresses that owner o_1 allows readers r_1 and r_2 to read data labeled with l_2, and owner o_2 does so for readers r_2 and r_3. Of course each annotation must be obeyed by the system, that is only r_2 will be allowed to access data labeled with l_2. Additionally, labels may specify integrity. $l_3 = \{? : p_1, \cdots, p_n\}$ specifies which principals trust the data labeled with l_3.

To allow the splitter to partition the program across the hosts of the target network, it must know the trust relationship between the principals and the hosts. This information is

```
1   public class OTExample {
2     int{Alice:; ?:Alice} m1;
3     int{Alice:; ?:Alice} m2;
4     boolean{Alice:; ?:Alice} isAccessed;
5
6     int{Bob:} transfer{?:Alice} (int{Bob:} n)
7     where authority(Alice) {
8       int tmp1 = m1;
9       int tmp2 = m2;
10      if(!isAccessed) {
11        isAccessed = true;
12        if(endorse(n, {?:Alice}) == 1)
13          return declassify(tmp1, {Bob:});
14        else
15          return declassify(tmp2, {Bob:});
16      }
17      else return 0;
18    }
19  }
```

Figure 2: The source code for the Oblivious Transfer example taken from [ZZNM2002], based on the Oblivious Transfer Problem [Rab1981]. Alice has two values (stored in fields *m1* and *m2*), exactly one of which Bob is allowed to learn. However, Bob does not want Alice to know, which value he has requested. Fields and methods have been annotated with labels to specify confidentiality and integrity requirements of the principals Alice and Bob.

specified by two components per principal and host. The *confidentiality label* $\mathtt{C}_{(h,A)}$ specifies the upper bound on the confidentiality of information that A allows to be sent to host h. Accordingly, the *integrity* label specifies whether the principal trusts information received from host h. For the example shown in Figure 2, the principals Alice, Bob, and Charlie specify the following confidentiality and integrity labels for the four hosts in the network. Alice does trust her own host A as well as the hosts T and S and also believes in the integrity of data received from these hosts. Bob trusts his own host B as well as hosts T and S, but only believes in the integrity of data received from his own host. Finally, Charlie trusts the host T. This results in the following sets $\mathtt{C}$ and $\mathtt{I}$:

$$\begin{array}{ll}
\mathtt{C}_{(A,\text{Alice})} = \{\text{Alice} :\} & \mathtt{I}_{(A,\text{Alice})} = \{? : \text{Alice}\} \\
\mathtt{C}_{(B,\text{Bob})} = \{\text{Bob} :\} & \mathtt{I}_{(B,\text{Bob})} = \{? : \text{Bob}\} \\
\mathtt{C}_{(T,\text{Alice})} = \{\text{Alice} :\} & \mathtt{I}_{(T,\text{Alice})} = \{? : \text{Alice}\} \\
\mathtt{C}_{(T,\text{Bob})} = \{\text{Bob} :\} & \mathtt{C}_{(T,\text{Charlie})} = \{\text{Charlie} :\} \\
\mathtt{C}_{(S,\text{Alice})} = \{\text{Alice} :\} & \mathtt{C}_{(S,\text{Bob})} = \{\text{Bob} :\}
\end{array}$$

For a host h, the sets $\mathtt{C}_h$ and $\mathtt{I}_h$ are computed as the union of the according sets $\mathtt{C}_{(h,\_)}$

and $\mathtt{I}_{(h,\_)}$, respectively. Along the same lines one can derive confidentiality and integrity labels for fields and expressions in a security-typed language. Generally, the sets $\mathtt{C}$ and $\mathtt{I}$ are unified into a single label $\mathtt{L}$ and the two functions C and I are used to extract each of the subsets.

For the splitting of an application onto the hosts available in a network, the authors in [ZZNM2002] specify constraints for fields and statements in the application. For a field f, the generated constraints are

$$C(\mathtt{L}_f) \sqsubseteq \mathtt{C}_h \text{ and } \mathtt{I}_h \sqsubseteq \mathtt{I}(\mathtt{L}_f)$$

The constraints for a statement S result from performing a simple definition-use analysis on all data used or defined in the statement. Let $U(S)$ and $D(S)$ be the set of values used and locations defined by S. Then

$$C\left(\sqcup_{v \in U(S)} L_v\right) \sqsubseteq \mathtt{C}_h \text{ and } \mathtt{I}_h \sqsubseteq I\left(\sqcap_{l \in D(S)} L_l\right)$$

For a list of additional constraints, including those regarding the program counter, refer to [ZZNM2002]. The essential property for this work is that in principal they all have the above form.

2.2 Erasure Policies

Erasure Policies (EPs) as introduced by Chong and Myers [CM2005] impose strong end-to-end requirements to enforce that information is either erased or made less accessible. They are based on a lattice of security levels. The simplest kind of a policy is a *label* l that limits how the labeled data may be used. In the setting of SPP this would be a set of confidentiality and integrity requirements. Additionally, *erasure policies* have the form $l_1 \overset{c}{\nearrow} l_2$, where l_1, l_2 are policies and c is a condition specifying that l_1 must be enforced on the labeled data, and once condition c is fulfilled, l_2 must be enforced as well, independent of the future evaluation of c. Finally there are *declassification policies* $l_1 \overset{c}{\rightsquigarrow} l_2$ stating that l_1 must be enforced on the labeled data, but once condition c is fulfilled, the data may be declassified. From thereon policy l_2 must be enforced, again independent of the future evaluation of c.

3 Dynamic Repartitioning

This section introduces our extension to the Secure Program Partitioning framework as introduced in Section 2.

Dynamic Repartitioning essentially shares all the properties of SPP as described above. The main achievement is that hosts may join or leave the network after an initial partitioning of the application has been found. This initial partitioning is important, since it ensures that the program can also be partitioned across a bigger set of hosts. We define the set of hosts used for constructing the initial partitioning as H_{init}. Hosts in this set are never allowed to leave the network, since they are needed to ensure the existence of a partitioning. Hosts that

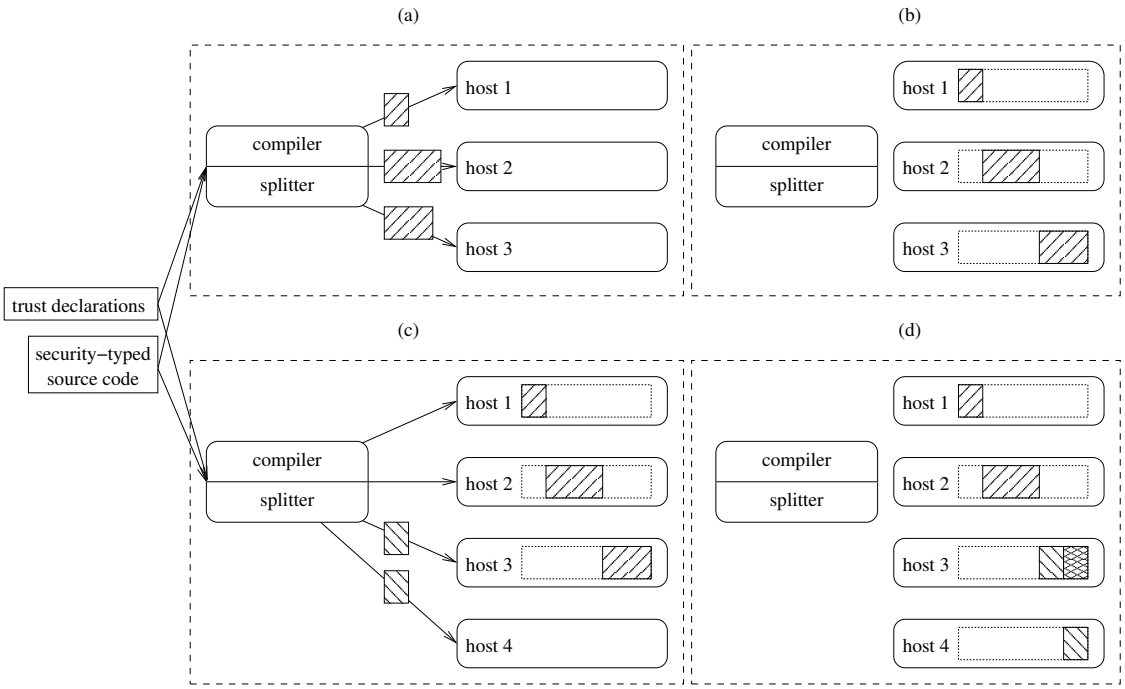

Figure 3: Secure Dynamic Program Repartitioning. The main change in contrast to SPP is that the Compiler/Splitter component is part of the network (a). The initial distribution of application code is done just like in the original framework (b). Once a new node enters the network (c), the Compiler/Splitter tries to find a new distribution that better fulfills the trust requirements specified by the principals. If the new host enables such a repartitioning, the new partitions are distributed to the hosts (d). Otherwise, the system remains unchanged. In a last step, hosts must erase the data that has been partitioned to another host (or make the data inaccessible). In (d) this is the hatched area of host 3.

join the network later are part of the set H_{join}. Hosts in this set may freely join or leave the network since they are not needed for the initial partitioning. In contrast, in the case of hosts from the set H_{init} *leaving* the network no guarantee can be given that the source code can be partitioned across the remaining hosts.

The effect of having an additional host in the network is that some of the constraints introduced in Section 2 may be resolved to an element that is smaller with respect to $\sqsubseteq$ than the original solution. This is true for all constraints where the confidentiality (integrity) sets for the new host n ($\mathtt{C}_n$ and $\mathtt{I}_n$) are smaller (bigger) than those of the originally chosen host h. Of course there is no *guarantee* that the new host will be chosen for data or computations, just like the host S in the Oblivious Transfer example.

3.1 Enforcing Erasure of Data

If data has been repartitioned to another host as result of a host joining the network, principals will want to be assured that their data has been erased or made less accessible on the

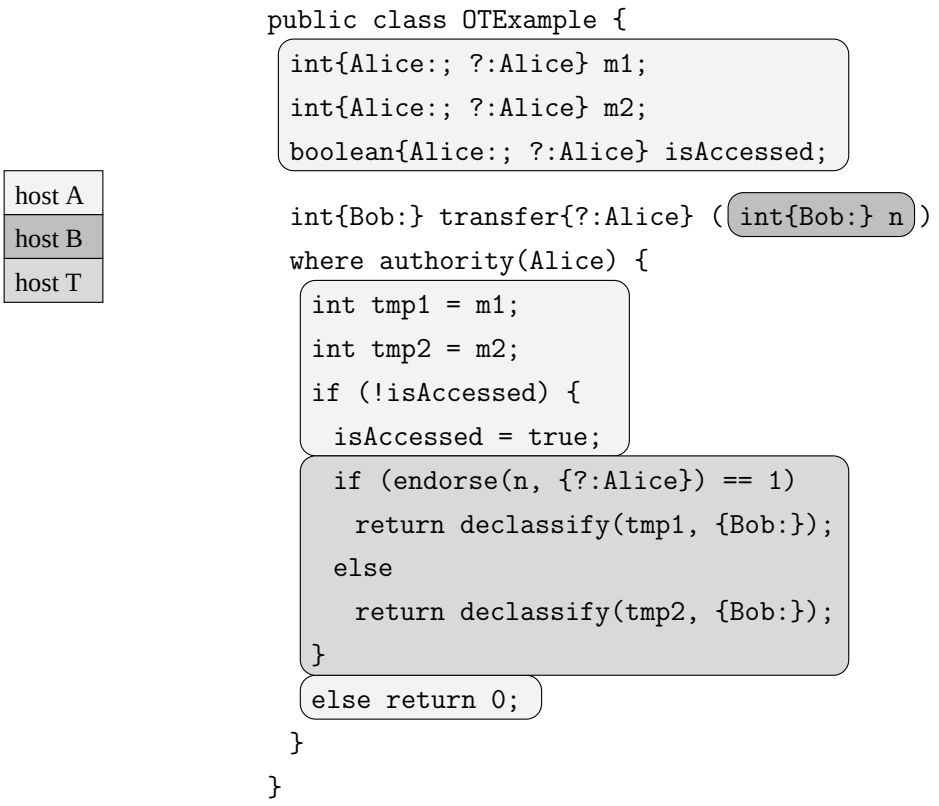

Figure 4: The Oblivious Transfer example including the partitioning of code pieces across three hosts according to [ZZNM2002].

original host. Using the framework of Erasure Policies as described in Section 2, this effect can be ensured automatically *without* additional annotations by the principals.

To do so, we introduce two conditions *rem* and *loc* that model the event of data being rescheduled to a remote host and data being available locally. We assume that during the redistribution of an application no data ever is reused. As the result of a host n joining the network, data might be repartitioned from host h to the new host n. If n leaves the network again, depending on the other hosts that joined or left the network, the data or parts of it could be repartitioned to be stored on h again. In this scenario, a fresh copy of the data would be created and stored on h. To make sure that data stored on hosts are not reused once they have been partitioned on another host, we use an erasure policy using the *rem* condition. The policy for some data d that is partitioned on a host h and in the original framework would have label l then becomes $l^{rem}\nearrow\top$. This ensures that after the d has been partitioned onto another host it can no longer be accessed on host h.

4 An Example

This section returns to the Oblivious Transfer example presented in Figure 2. Using the annotations to the source code and the confidentiality and integrity labels specified by the principals, the code can be partitioned on three hosts A, B, and T. The labels for the four available hosts are

$$\begin{array}{ll} \mathtt{C}_A = \{\text{Alice} :\} & \mathtt{I}_A = \{? : \text{Alice}\} \\ \mathtt{C}_B = \{\text{Bob} :\} & \mathtt{I}_B = \{? : \text{Bob}\} \\ \mathtt{C}_T = \{\text{Alice} :, \text{Bob}, \text{Charlie} :\} & \mathtt{I}_T = \{? : \text{Alice}\} \\ \mathtt{C}_S = \{\text{Alice} :, \text{Bob} :\} & \mathtt{I}_S = \{? :\} \end{array}$$

Figure 4 shows how the application again and how to partition it on three of the hosts, A, B, and T.

Assume that later on a host N joins the network for which the principals have specified $\mathtt{C}_N = \{\text{Alice} :, \text{Bob}\}$ and $\mathtt{I}_N = \{? : \text{Alice}\}$. Repartitioning data and computations that in Figure 4 have been partitioned to T will allow to fulfill the requirements of both Alice and Bob better than in the original partition, since the set $\mathtt{C}_T$ is larger than necessary to fulfill all constraints generated for the program (since the labels in the program do not reason about Charlie).

5 Conclusions and Future Work

We have introduced a framework for Secure Dynamic Program Repartitioning. The work presented in this paper builds on prior work on Secure Program Partitioning, a framework working with a fixed sets of hosts. Our framework inherits all properties from SPP, but additionally allows hosts to join and (under certain conditions) leave the network, possibly causing a re-partitioning of the application, and uses Erasure Policies to ensure that data is made inaccessible on hosts that no longer store the data after repartitioning has taken place.

We see two limitations for our approach. On the on hand, for the time being like SPP we only investigate single-threaded programs. Repartitioning of applications can only occur when the program is not being executed. Thus, no additional communication primitives are needed on top of those introduced by [ZZNM2002]. On the other hand, theoretically the system does not know the confidentiality and integrity sets of principals for hosts joining the network. This could be avoided by having a list of hosts potentially joining the network and principals specifying policies for each of them or by specifying certain properties of a host like operating system or owner to deduct policies for each user based on preferences. While these approaches are inherently inelegant, we currently see no real alternative.

We are currently working on the formalization of the extensions described here. It will be especially interesting to model fully dynamic networks based on the available confidentiality and integrity, that is allow hosts to leave the network again based on the trust of principals into the hosts remaining in the network. In order to ensure that applications can still be partitioned in this case, the system will need to ensure that (instead of forcing the initial

set of hosts to remain in the network) the hosts remaining in the network provide the same confidentiality and integrity as the initial set. Another interesting problem is to change the token model introduced by [ZZNM2002] so that repartitioning can be guaranteed to be safe also while the application is executed.

References

[ABHR1999] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 147–160, New York, NY, USA, 1999. ACM Press.

[CM2005] Stephen Chong and Andrew C. Myers. Language-Based Information Erasure. In *CSFW '05: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 241–254, Washington, DC, USA, 2005. IEEE Computer Society.

[HR1998] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with security and integrity. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, 19–21 January 1998.

[ML2000] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4), October 2000.

[Mye1999] Andrew C. Myers. JFlow: practical mostly-static information flow control. In ACM, editor, *POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, ACM SIGPLAN Notices, pages 228–241, New York, NY, USA, 1999. ACM Press.

[PC2000] François Pottier and Sylvain Conchon. Information flow inference for free. *ACM SIGPLAN Notices*, 35(9):46–57, September 2000.

[Rab1981] M. Rabin. How to exchange secrets by oblivious transfer. Technical Report TR-81, Harvard Aiken Computation Laboratory, 1981.

[Sch2000] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.

[SV1998] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*, pages 355–364, New York, January 1998. Association for Computing Machinery.

[VSI1996] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, December 1996.

[ZM2001] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. *Lecture Notes in Computer Science*, 2028:46–??, 2001.

[ZZNM2001] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In Greg Ganger, editor, *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 1–14, New York, October 21–24 2001. ACM Press.

[ZZNM2002] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, August 2002.