Particionamento Seguro de Código

Luiz Gonzaga Mota Barbosa
UECE,Brasil
gonzaga.dd@gmail.com

Leandro Jales Martins
UECE, Brasil
le.jales@gmail.com

Pablo Ximenes
UPRM, USA
pablo@ximenes.info

André dos Santos
UECE, Brasil
andre@dossantos.org

Resumo

Program slicing é uma técnica utilizada para dividir um programa grande em fatias (slices) menores, que são programas executáveis e independentes e, quando executadas em uma certa associação, reproduzem o comportamento do programa original. Essa técnica tem sido utilizada com diversas aplicações tais como depuração de erros (debugging), manutenção e entendimento de software entre outras.

Nossa proposta é aplicar essa técnica em smart cards, de forma que as fatias são produzidas do lado de fora do dispositivo e carregadas para ele sobre demanda. Porém os métodos de particionamento existentes são vulneráveis a ataques de canal lateral (side channel attacks), nos quais o atacante deriva informações importantes relacionadas ao programa observando seu comportamento. Nesse trabalho nós mostramos uma técnica de particinamento seguro de código compatível com tal plataforma e quaisquer outras que necessitam de resistência a tais ataques.

Esse trabalho é patrocinado pelo U.S. Army Research Office (ARO).

1. Introdução

Com o rápido crescimento dos setores que necessitam de cálculos seguros, veio o aumento na demanda por transações seguras. O uso de dispositivos embarcados também tem aumentado acompanhando essas necessidades. *Smart cards* são importantes dispositivos embarcados que oferecem resistência à adulteração e um ambiente seguro. Um contexto típico de sua utilização é carregar aplicações para dentro deles e executá-las em seu interior. Para um atacante, fica impossível adivinhar o que acontece dentro do cartão devido à sua propriedade de resistência à adulteração, dessa forma, o dispositivo se assemelha a uma caixa preta na qual ninguém é capaz de adivinhar o que há em seu interior[1]. Smart cards têm empregado técnicas interesantes para manter a resistência à adulteração em termos de ataques de temporização (*timing attacks*) ou ataques de potência (*power attacks*) [4, 5].

Devido sua grande restrição de memória (8KB – 64KB), smart cards tem sido utilizados apenas com uma simples aplicação residindo em seu interior, realizando funções simples como cartões telefônicos e cartões de identificação de assinante. Com isso, surge um grande interresse em *smart cards* multiaplicação, os quais podem suportar várias aplicações executando em seu interior[1].

A fim de permitir que múltiplas aplicações residam no cartão, alguém deve quebrar os programas em partes menores capazes de serem carregadas para dentro dele sobre demanda e que mantenham o comportamento do programa original. Dessa forma, o cartão manteria em seu interior apenas as fatias atualmente em execução ou aquelas prontas para executar. Mas particionar um programa, consiste em parte do programa estar fora do cartão. Durante a transmissão desses sub-programas para o dispositivo, alguém pode observar o comportamento do programa e, a partir daí, lançar um ataque malicioso à aplicação. Nesse ponto, alguém deve garantir que nenhuma informação relevante ao comportamento do programa poderá ser obtida por um observador.

Nesse trabalho, apresentamos um novo método de particionamento de código no qual nenhuma informação relacionada ao comportamento do programa pode ser observada durante a transmissão de cada fatia, carregada uma por uma, para o dispositivo embarcado. As fatias são geradas e ordenadas em ordem não crescente de tamanho. As maiores são, se possível, representadas por fatias menores necessárias para sua composição. As dependências são ordenadas em níveis e

as fatias são escolhidas seguindo certas propriedades.

2. Motivação

A motivação principal desse artigo é o desenvolvimento de uma técnica para particionar programas de maneira que não escape nenhuma informação sobre o controle de fluxo da aplicação residente no cartão. Nossa técnica de fatiamento de código é totalmente diferente das já existentes [2]. Em nosso esquema de particionamento, códigos móveis são primeiramente particionados no lado do servidor, o qual gera fatias executáveis e independentes que, associadas produzem o comportamento do programa original. Tais fatias são carregadas, uma por uma, sobre demanda para o cartão e executadas uma atrás da outra, assim que uma fatia termina sua execução, a próxima fatia é requisitada ao servidor.

O foco desse artigo está sobre o desenvolvimento de uma técnica de particionamento seguro de código resistente ao vazamento de informações relevantes ao comportamento do programa. Essa técnica será empregada particularmente em smart cards a fim de permitir que várias aplicações residam no mesmo dispositivo.

3. *Program slicing* (Fatiamento de código)

Program slicing é uma técnica utilizada para particionar um programa grande em fatias (*slices*) menores. Tais fatias são partes, independentes e executáveis, do programa maior. Quando executadas de maneira associada, produzem o comportamento do programa original. As fatias são formadas seguindo um critério de fatiamento sobre a variável *v* na instrução *n*, representado pela seguinte notação: <n, v> [2]. Esse critério está relacionado com quais instruções do programa afetarão ou serão afetadas, direta ou indiretamente, pelo valor da variável *v* na instrução n. Se estivermos interessados em quais instruções serão afetadas pelo valor da variável *v* na instrução *n,* então usaremos *forward slicing* (fatiamento adiante). Caso contrário, se estivermos interessados em quais instruções do programa têm influência sobre o valor da variável *v* na instrução n, então usaremos backward slicing (fatiamento retrógrado). O fatiamento pode ser também estático ou dinâmico. No primeiro, as fatias são tomadas em relação a qualquer entrada possível para o programa. No segundo, estas são formadas em relação à uma entrada específica, a qual é especificada no critério de fatiamento (<variável, instrução, entrada>)[2, 3, 6]. A técnica utiliza grafos de controle de fluxo como representação intermediária, para auxiliar no cálculo das fatias[2].

Program slicing tem sido empregada na depuração de programas [7, 8, 9, 10, 11, 12, 13, 14], testes [25, 26, 27, 28, 29, 30, 31, 32, 33], manutenção [19, 20, 21, 22, 23, 24], entendimento [15, 16, 17, 18], métricas de softwares [34, 35, 36, 37] entre outras aplicações.

4. Fatiamento seguro de código

O fatiamento de qualquer dado programa gerará subprogramas que podem ter códigos executáveis sobrepostos, desde que certas instruções podem influenciar o comportamento de muitas variáveis diferentes e então seriam inclusos em todas as fatias que referenciam estas variáveis. Se alguém gerar todas as fatias para um dado programa, produzirá, inevitavelmente, um conjunto de subprogramas que farão exatamente as mesmas ações do programa original, mas de forma que executará novamente muitas instruções em diferentes fatias.

Nossa hipótese é que essa natureza de sobreposição de fatias de programas possa ser tomada como vantagem para resolver o problema de particionamento de progamas móveis. Nós definimos o problema do particionamento seguro de programa móvel como um problema de divisão de um programa que é maior que a plataforma de execução remota a fim de carregar as subpartes (resultantes do particionamento) para dentro da plataforma remota para execução de maneira que não revele qualquer informação para um observador que possa ser utilizada para derivar informação de controle de fluxo relativo ao programa original.

Dessa maneira, nós acreditamos que é possível particionar um programa com objetivo de esconder sua informação de controle de fluxo através da montagem de grafo de dependência de fatias complexas e mais simples de forma que fatias mais complexas possam ser montadas como uma representação de fatias menores e mais simples (e geralmente sobrepostas). A idéia principal é produzir um grafo de relacionamento entre fatias, no qual uma fatia mais complexa seria representada como referências à fatias mais simples. Por exemplo, dado o progama abaixo, nós extrairemos uma fatia complexa e a representaremos como fatias mais simples.

Programa com trocas e adições:

```
1-  a = 1
2-  b = 30
3-  c = 37
4-  d = 10
5-  temp = a
6-  a = b
7-  b = temp
8-  d=d+1
9-  c = c + d
10-  print a,b,c
```

Fatia 0: <linha:10, var: c>

```
3-  c = 37
```

4- d = 10
8- d=d+1
9- c = c + d
10- print a,b,c

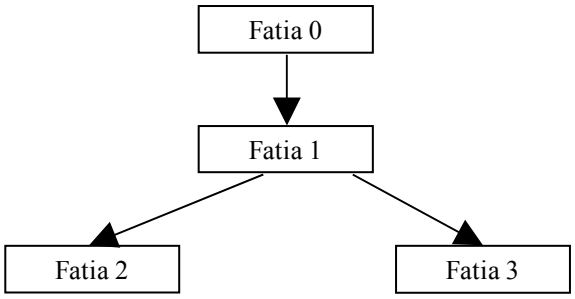
Fatia 1: <linha:9, var: c>
3- c = 37
4- d = 10
8- d=d+1
9- c = c + d

Fatia 2: <linha:8, var: d>
4- d = 10
8- d = d + 1

Fatia 3: <linha:3, var: c>
3- c = 37

De acordo com nossa descrição, Fatia 0 poderia ser representada como uma associação das fatias 1,2 e 3, onde a fatia 1 também seria derivada a partir de 2 e 3.

Dessa forma, o grafo de relacionamento seria:



Como podemos ver, Fatia 0 teve sua complexidade simplificada na forma de fatias menores que podem ser associadas a fim de extrair o mesmo comportamento em relação as variáveis de interesse com o programa original. Adicionalmente, podemos notar que certas fatias (2 e 3), têm códigos (instruções) sobrepostos que serão executados em ambas as fatias (isto é, duas vezes). Ao final, somente as fatias 2 e 3 serão executadas, mas os resultados de seus cálculos podem ser utilizados para derivar o resultado para a fatia 0. Essa propriedade junto com uma atribuição estratégica de fatias sobrepostas serão usadas para proteger o fluxo do programa contra observadores no problema de particionamento de programas móveis.

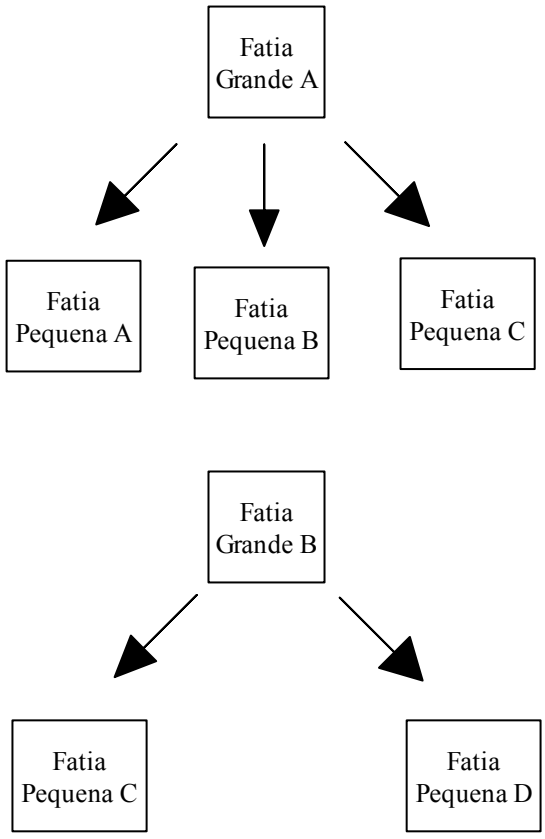
Portanto, nós definimos um algoritmo geral para produzir o grafo de dependência e assim gerar o código particionado. Esse algoritmo trabalha como se segue:

1. Gere todas as fatias do programa (fatiamento retrógrado – *backward program slicing*).

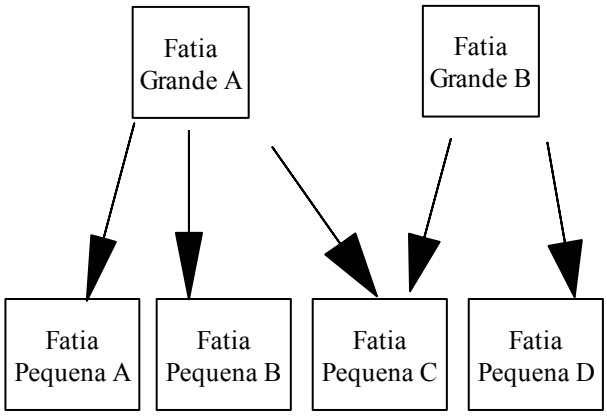
2. Ordene as fatias em ordem não decrescente de tamanho, ou seja, da menor para a maior.

3. Represente as fatias maiores como uma associação das menores sempre que possível.

4. Crie o grafo de dependências das fatias mostrando as fatias menores necessárias para produzir as maiores. A fim de produzir o grafo de dependências, primeiro nós mostramos os descendentes de uma fatia grande em uma estrutura de árvore. Os descendentes de uma fatia grande são as fatias menores que são usadas para decompor a maior. Como a mesma fatia pequena pode ser descendente de diferentes fatias grandes, essa associação de árvores individuais gerará, ocasionalmente, um grafo que desfaz a estrutura de árvore. Esse é o motivo de chamarmos grafo de dependências (e não árvore de dependência). A figura abaixo exemplifica esse processo mostrando como duas árvores individuais são fundidas em um grafo de dependências.



Árvores Individuais



Grafo de Dependências
(Árvores Fundidas)

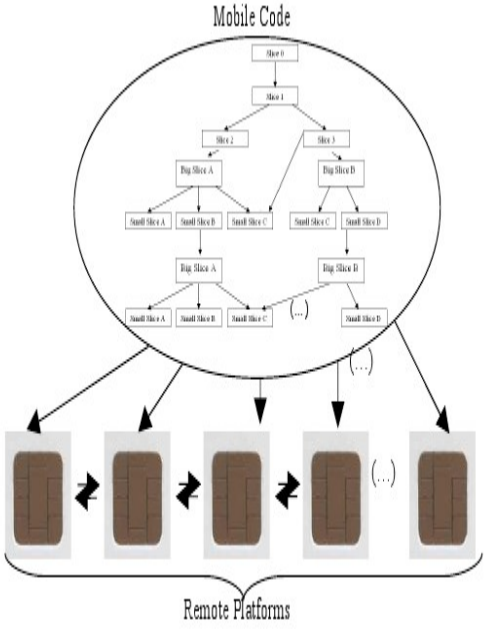
5. Organize o grafo de dependências em níveis, onde fatias no mesmo nível não possuem qualquer relacionamento de dependência entre cada uma (podem ser executadas independentemente) e podem depender apenas daquelas fatias que estão em um nível inferior.

6. Estabeleça a natureza de sobreposição de cada fatia.

7. Escolha os nós no grafo de maneira aleatória para produzir partições que devem obedecer essas propriedades:
   a) A computação de todas as partições individuais em associação devem gerar a computação do programa todo.
   b) As partições devem ter um certo nível de comportamento de sobreposição que nem é tão pequeno para torná-las independentes, nem tão grande para degradar, significantemente, a performance do programa todo.
   c) A maior partição gerada nesse processo deve ser pequena o suficiente para ajustar-se à plataforma remota.

O algoritmo para essa escolha aleatória no passo 7, se for julgado como simples o suficiente, pode ser feito durante o tempo de execução, ao passo que, no caso desse algoritmo ser muito complexo para o tempo de execução, um conjunto de muitos cenários de execução podem ser calculados durante o tempo de compilação. Isso significa o cálculo de uma trajetória de fatias para a execução do código móvel. Isso garantirá não somente as restrições de adequação de tamanho para as fatias, mas também tornará mais difícil adivinhar qualquer relacionamento das estruturas de fluxo de controle que podem ser usadas para adivinhar informações secretas.

As partições resultantes serão compostas por informações de cabeçalho que normalizarão cada partição a fim de torná-las indistinguíveis umas das outras. Adicionalmente, os cabeçalhos incluirão códigos hash para verificar integridade e outras informações de controle que possa ser utilizada para execução correta do código. Cada fatia pode incluir instruções do código do programa e referências para outras partições. Como o estado das variáveis é guardado no lado de dentro da plataforma remota, não é provável que aconteça a sobreposição de fatias referenciadas, exceto em casos de exaustão de memória. Outra prorpiedade importante da técnica apresentada é que cada partição pode ser executada independentemente desde que as fatias são, por definição, partes independentes do programa. Isso permite a execução paralela de programas através do uso de mais de uma plataforma remota. No caso particular de execução paralela, o código móvel particionado está mais sujeito a apresentar sobreposição de fatias referenciadas. Isso pode ser evitado projetando-se um protocolo de troca de mensagens entre a plataforma individual remota a fim de evitar cálculos de fatias referenciadas replicadas. A ferramenta geral para o modelo de execução segura do código móvel que é proposta é descrita na seguinte figura:



5. Análise de segurança

A fim de expressar o nível de garantia expressa pelo esquema proposto em termos mais concretos, nós analisamos sua segurança. Nós concentramos o problema sobre o nível de dificuldade que um atacante tem durante a tentativa de derivar a informação do canal lateral considerando o entendimento do fluxo de fatias dentro do programa. Em outras palavras, quão difícil será para um atacante apontar laços (*loops*), chamadas à funções, etc; apenas através da observação de como as

fatas individuais são carregadas para dentro da memória. Nessa análise, nós não consideramos outras informações de canal lateral que não estão relacionadas com o carregamento e descarregamento de fatias (isto é, temporização, potência, etc).

Condições básicas

Um atacante a fim de obter qualquer dado significante para montar qualquer tipo de ataque que o permitiria derivar informação de controle de fluxo do programa, ele precisaria observar, no mínimo, dois fatos básicos:

1. Para um ataque de execução simples de programa, o atacante precisará encontrar, no mínimo, uma fatia repetida, uma fatia que está sendo requisitada mais de uma vez para ser carregada para dentro da memória. Isso significa que se nenhuma fatia é carregada mais de uma vez, o atacante não está apto a derivar informação de controle de fluxo a partir de uma execução individual do programa, desde que nenhum relacionamento entre as fatias pode ser determinado dessa forma. Essa condição básica está ilustrada na figura 1.

2. Mesmo que não hajam fatias repetidas em uma execução individual do programa, o atacante poderia executar o programa várias vezes sob diversas condições a fim de obter múltiplas imagens do fluxo do programa. Dessa forma, o atacante precisará identificar uma fatia fixa, que é qualquer fatia que mantém sua posição relativa na ordem de carregamento. O atacante estará apto a estabelecer que a fatia fixa possui controle de fluxo de alguma forma e assim iniciar seu ataque a partir desse fato. Esta condição básica está ilustrada na figura 2.

Se nenhuma dessas duas condições básicas são encontradas, o atacante não estará apto a montar qualquer ataque significativo contra o esquema proposto. Nós mostraremos como o esquema proposto está completamente protegido contra a primeira condição e como o esquema proposto invalida a segunda condição mesmo quando ela é encontrada.

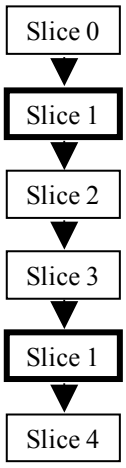


Figura 1. Fatia repetida

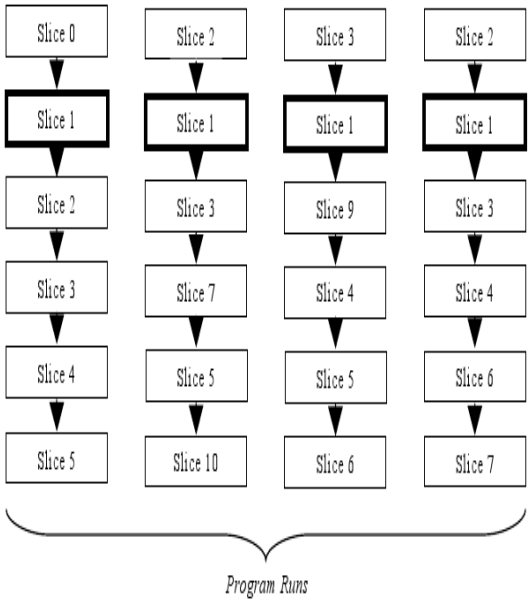


Figura 2. Fatia fixa.

Caso 1: Repetição de fatias

No esquema proposto, quando um programa está em execução, o que realmente acontece é que pequenos subprogramas independentes estão executando em nome do software original. Esses subprogramas são as fatias calculadas do programa que servirão para calcular o estado do programa original em uma dada situação. As fatias que serão carregadas a fim de serem executadas dessa forma, serão escolhidas baseadas no gafo de dependências explicado anteriormente. Toda vez que uma fatia é executada, a informação de estado é armazenada. Informação de estado significa os resultados da execução da fatia na forma de variáveis de

saída. Dessa forma, se uma fatia é necessária novamente, nenhuma re-execução será necessária, sendo suficiente apenas os resultados das variáveis de saída. É importante notar que fatias de programa da forma que estão sendo propostas são diferentes de chamadas à funções ou saltos condicionais. No caso de chamadas à funções e saltos condicionais, as variáveis de saída podem resultar em diferentes valores para diferentes chamadas, mas como as fatias não são apenas código, mas código colocado em um ponto específico do fluxo do programa (critério de fatiamento), isso garante que quando a fatia é necessária, ela será chamada apenas uma vez e que fatias repetidas não serão encontradas por um atacante.

Caso 2: Fatia fixa

Como demonstrado anteriormente, o esquema proposto deriva seu caminho de execução a partir do grafo de dependências que co-relaciona fatias em uma ordem de hierarquia onde fatias no mesmo nível são independentes umas das outras. Dessa forma, o que realmente será carregado para dentro da memória são os nós do nível mais baixo do grafo de dependências, todos do mesmo nível. Como as fatias estão no mesmo nível do grafo de dependências, elas são todas independentes umas das outras e assim não tem a exigência de serem carregadas em uma ordem específica. O fato da ordem de carregamento não ser importante no esquema proposto, garante que não haverão fatias fixas em qualquer ponto do fluxo de execução do programa. Essa propriedade importante protege o esquema proposto contra ataques que analisam múltiplas execuções do programa, desde que cada execução gerará diferentes imagens do fluxo de execução, sem pontos de sobreposição.

6. Conclusões

Esse trabalho apresentou um esquema de particionamento seguro de código resistente a ataques de canal lateral a ser utilizado particularmente em smart cards. Nós mostramos como particionar o programa seguindo um grafo de dependências entre todas as fatias geradas. Fizemos também uma análise de segurança do esquema, mostrando que ele está protegido contra as duas condições básicas necessárias para um ataque de tal natureza , tais condições foram citadas nesse trabalho.

Como trabalho futuro esperamos realizar a análise de desempenho do modelo proposto e implementá-lo em sua versão protótipo.

7. Agradecimentos

Esse trabalho é patrocinado pelo Escritório de Investigação do Exército dos Estados Unidos da América (U. S. Army Research Office – ARO), processo número Grant # W911NF-07-1-0271.

8. Referências

[1] T. Zhang, S. Pande, A. dos Santos, F. J. Bruecklmayr, "Leakage-Proof Program Partitioning," Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems ? CASES 2002, October 2002.

[2] David W. Binkley, Keith Brian Gallagher, "Program slicing".

[3] Mark Harmam, Robert M. Hierons, "A overview of program slicing".

[4] P.C. Kocher, Timing attacks on implementations of Die-Hellman, RSA, DSS, and other systems, Proceedings Crypto'96, LNCS 1109, Springer-Verlag 1996, 104-113.

[5] P. Kocher, J. Jaffe and B. Jun, Differential Power Analysis, Crypto99.

[6] M.Weiser. Program slicing. In Proceeding of the Fifth International Conference on Software Engineering, pages 439-449, May 1981.

[7] H. Agrawal. Towards automatic debugging of computer programs. Technical Report SERC-TR-40-P, Purdue University, 1989.

[8] M. Kamkar. Interprocedural Dynamic Slicing with Applications to Debugging and Testing. PhD thesis, Linkoping University, S-581 83 Linkoping, Sweeden, 1993.

[9] B. Korel and J. Laski. STAD - A system for testing and debugging: User perspective. In Proceedings of the Second Workshop on Software Testing, Veri cation and Analysis, pages 13 20, Ban , Alberta, Canada, July 1988.

[10] J. R. Lyle. Evaluating Variations of Program Slicing for Debugging. PhD thesis, University of Maryland, College Park, Maryland, December 1984.

[11] J. R. Lyle and M. D. Weiser. Experiments on slicing-based debugging aids. In Elliot Soloway and Sitharama Iyengar, editors, Empirical Studies of Programmers. Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[12] J. R. Lyle and M. D. Weiser. Automatic program bug location by program slicing. In Proceeding of the Second International Conference on Computers and Applications, pages 877 882, Peking, China, June 1987.

[13] T. Shimomura. The program slicing technique and its application to testing, debugging and

maintenance. Journal of IPS of Japan, 9(9):1078 1086, September 1992.

[14] M. Weiser. Programmers use slices when debugging. CACM, 25(7):446 452, July 1982.

[15] F. Cutillo, R. Fiore, and G. Visaggio. Identification and extraction of domain independent components in large programs. In Proceedings of the Working Conference on Reverse Engineering, pages 83 92, June 1993.

[16] F. Lanubile and G. Visaggio. Function recovery based on program slicing. In Proceedings of the Conference on Software Maintenance - 1993, pages 396 404, September 1993.

[17] E. Merlo, J. Girard, L. Hendren, and P. De Mori. Multi-valued constant propagation for the reengineering of user interfaces. In Proceedings of the Conference on Software Maintenance - 1993, pages 120 129, September 1993.

[18] M. Pleszcoch, P. Hausler, A. Hevner, and R. Linger. Function theoretic principles of program understanding. In Proceedings of the Twenty-third Annual Hawaii Conference on System Sciences, volume 4, pages 74 81, 1990.

[19] G. Ramalingam and T. Reps. A theory of program modi cations. In S. Abramsky and T.S.E. Maibaum, editors, Proceedings of the Colloquium on Combining Paradigms for Soft ware Development, pages 137 152. Springer-Verlag, 1991.

[20] M. Plato and M. Wagner. An integrated program representation and toolkit of the maintenance of c programs. In Proceeding of the Conference on Software Maintenance, October, 1991.

[21] K. Ono, K. Maruyama, and Y. Fukazawa. Applying a veri cation method and a decomposition method to program modi cation. Trans. IEICE, J77-D-I(11), November 1994. JAPAN.

[22] G. Ramalingam and T. Reps. Modifcation algebras. In Proceedings of the Second Intenational Conference on Algebraic Methodology and Software Technology. Springer-Verlag, 1992.

[23] J. Laski and W. Szermer. Identifcation of program modifications and its application in software maintenance. In Proceedings of the Conference on Software Maintenance – 1992, pages 282 290, November 1992.

[24] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. IEEE Transactions on Software Engineering, 17(8):751 761, August 1991.

[25] H. Agrawal, R. DeMillo, and E. Spafford. A process state model to relate testing and debugging. Technical Report SERC-TR-27-P, Purdue University, 1988.

[26] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages,. ACM, 1993.

[27] D. Binkley. Using semantic diferencing to reduce the cost of regression testing. In Proceedings of the Conference on Software Maintenance - 1992, pages 41 50, November 1992.

[28] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In IEEE International Conference on Software Maintenance, 1995.

[29] K. B. Gallagher. Using program slicing to eliminate the need for regression testing. In Eighth International Conference on Testing Computer Software, May 1991.

[30] M. Kamkar, P. Fritzson, and N. Shahmehri. Interprocedural dynamic slicing applied to interprocedural data flow testing. In Proceeding of the Conference on Software Maintenance -93, pages 386 395, 1993.

[31] B. Korel and J. Laski. STAD - A system for testing and debugging: User perspective. In Proceedings of the Second Workshop on Software Testing, Veri cation and Analysis, pages 13 20, Ban , Alberta, Canada, July 1988.

[32] J. Laski. Data ow testing in stad. The Journal of Systems and Software, 1989.

[33] G. Rothermel and M.J. Harrold. Selecting tests and identifying test coverage requirements for modi ed software. In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 169 84, August 1994.

[34] L. Ott and J. Bieman. E ects of software changes on module cohesion. In Proceedings of the Conference on Software Maintenance - 1992, pages 345 353, November 1992.

[35] L. Ott and J. Thuss. The relationship between slices and module cohesion. In International Conference on Software Engineering, May 1989.

[36] H. Longworth. Slice based program

metrics. Master's thesis, Michigan Technological University, Houghton, Michigan, 1985.

[37] H. Longworth, L. Ott, and M.Smith. The relationship between program complexity and slice complexity during debugging tasks. In COMPSAC 86, 1986.