Program Fracture and Recombination for Efficient Automatic Code Reuse

Peter Amidon
School for Independent Learners
Los Altos, CA 94022

Eli Davis, Stelios Sidiroglou-Douskos, and Martin Rinard
MIT EECS and MIT CSAIL
Cambridge, MA 02139

***Abstract*—We present a new code transfer technique, program fracture and recombination, for automatically replacing, deleting, and/or combining code from multiple applications. Benefits include automatic generation of new applications incorporating the best or most desirable functionality developed anywhere, the automatic elimination of errors and security vulnerabilities, effective software rejuvenation, the automatic elimination of obsolete or undesirable functionality, and improved performance, energy efficiency, simplicity, analyzability, and clarity.**

The technique may be particularly appropriate for high performance computing. The field has devoted years of effort to developing efficient (but complex) implementations of standard linear algebra operations with good numerical properties. At the same time these operations also have very simple but inefficient implementations, often with poor numerical properties. Program fracture and recombination allows developers to work with the simple implementation during development and testing, then use program fracture and recombination to automatically find and deploy the most appropriate implementation for the hardware platform at hand. The benefits include reduced implementation effort, increased code clarity, and the ability to automatically search for and find efficient implementations with good numerical properties.

I. Introduction

We present a new technique, *program fracture and recombination*, for automatically locating and transferring computations between multiple applications. This technique promises to significantly advance our ability to more productively leverage the enormous amount of existing software. Even more, program fracture and recombination holds out the promise of automatic program improvement and evolution without the need for any developer or potentially even any human involvement. Starting with multiple programs, program fracture and recombination operates as follows:

- **Fracture:** Fracture the programs into *shards* — pieces of the programs that implement a computation or functionality. The granularity of the fracture determines the size of the shards. Potentially useful granularities include functions, procedures, classes, abstract data types, modules, loops, and program slices. Program fracture typically includes the encapsulation of each shard into its own separately invocable program for testing, analysis, and exploration.
- **Characterization:** Characterize the behavior and characteristics of each shard. Examples include running the encapsulated shard on automatically generated inputs to obtain example input/output pairs, recording input/output pairs for the shard as invoked in context by executions of the program in which it was originally embedded, static analyses which partially or completely characterize the semantics of the shard, abstractions of the semantics obtained by generalizing the recorded input/output pairs, and specifications, either inferred or provided by the developer.
- **Shard Matching and Replacement:** One potential application replaces an original shard with a better replacement shard. There are many axes along which the replacement shard may be better — it may be more efficient, have better numerical properties, be simpler to understand, be endowed with additional capabilities such as the ability to execute successfully in parallel or distributed environments, have more error checking code, be more secure or better preserve privacy, or be missing undesirable or irrelevant functionality.
- **Shard Insertion:** Another potential application transfers functionality from one program to another. In this application a shard is taken from a donor program and inserted into a recipient. Vertical transfers take place between different versions of the same project. Horizontal transfers take place between independent projects (or independent forks of the same project). The donor and recipient can even be the same system. Scenarios include transfering functionality across software systems and correcting errors. CodePhage, which automatically locates and transfers security checks across multiple applications, implements a form of shard insertion [1].
- **Shard Rejuvenation:** Software projects often include obsolete optimizations for extinct hardware platforms. The resulting code complexity can defeat optimizing compilers and hinder the understandability, extensibility, analyzability, and maintainability of the system. Shard rejuvenation can replace the complex, obsolete version of the code with a simpler version.
- **Shard Removal:** As software evolves, previously desirable functionality can often become irrelevant. Potential drawbacks include difficulty analyzing the program and residual errors and vulnerabilities left over in the now irrelevant code [2], [3]. Shard removal can automatically eliminate the now undesirable code and functionality. It can also eliminate functionality that should never have been introduced into the application at all.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

Given the ease of obtaining sample inputs and outputs, either by automatically generating the input or by recording inputs presented to shards in context during executions of the enclosing programs, we expect input/output driven shard identification and transfer to play a prominent role. Input/output driven approaches can also promote the replacement of shards with other shards with different semantics — for example, the replacement of shards with errors or incomplete implementations with shards that have fewer errors or implement more cases. One straightforward approach to implementing this kind of shard replacement with analysis/semantics-based approaches would involve a concept of specification ordering and desirability.

Benchmark	Size	Original Time	Shard Replaced Time	Ratio
Matrix Multiply	100x100	0.1	0.1	1
Matrix Multiply	250x250	16.1	11.1	1.45
Matrix Multiply	500x500	167.6	88.7	1.89
Matrix Multiply	750x750	982.7	336.5	2.92
Matrix Multiply	1000x1000	2014.8	794.9	2.53
Linear Solver	100x100	0.4	0.7	0.57
Linear Solver	250x250	6.3	5.3	1.19
Linear Solver	500x500	56.2	36.9	1.52
Linear Solver	750x750	247.1	119	2.08
Linear Solver	1000x1000	784.3	269.4	2.91

TABLE I. PERFORMANCE RESULTS FOR CASE STUDIES. TIMES IN MILLISECONDS AVERAGED OVER TEN RUNS. NUMBERS COLLECTED ON A SURFACE PRO 2 512G WITH AN INTEL CORE I5-4300U PROCESSOR (2 CORES, 4 THREADS, 1.9 GHZ PROCESSOR BASE FREQUENCY).

II. CASE STUDIES

We next discuss two case studies that highlight the potential benefits of code fracture and recombination for high performance computing. These case studies are based on our LLVM implementation of code fracture and recombination (our technical report presents two more case studies [4]).

A. Motivation

Many computations have straightforward basic implementations but very complex maximally efficient implementations. Particularly prominent examples include sorting, linear algebra, and linear programming. The availability of powerful but difficult to program hardware such as graphics accelerators can further increase the distance between basic and maximally efficient implementations.

Program fracture can enable developers to write a basic implementation, in some cases by simply copying several lines from a textbook. Fracturing their program can expose the basic implementation as one of the shards. Fracturing other programs can expose replacement shards that implement the same functionality (or even approximate versions of the functionality). Replacing the original basic shard with the more efficient shard can automatically improve the performance (and, for many numerical computations, properties such as stability and accuracy) without the need for the developer to manually investigate the performance, find substitutes, or refine the implementation.

High performance linear algebra subcomputations comprise a particularly compelling application of this basic idea. The basic matrix multiply computation, for example, can be expressed as a simple triply nested loop. But the field has devoted decades of effort to obtaining efficient matrix multiply implementations. Standard techniques include parallelization, blocking for multiple levels of the memory hierarchy, and specialized implementations for GPU platforms.

Program fracture and recombination can provide one way to automatically search for and find these efficient implementations. In addition to serving as a code search mechanism, this approach can reduce the implementation effort and increase the clarity, conciseness, and maintainability of the source code.

B. Code Fracture and Recombination Implementation

Our current LLVM implementation first instruments the program to record the values of the parameters passed into each function on entry and the values of the parameters and the return value on exit. We also use a modified implementation of the C memory allocation routines (malloc, calloc, free) that records the sizes of allocated blocks of memory. The instrumentation uses these sizes to determine the number of elements in dynamically allocated memory blocks passed as parameters to each function. The implementation also records the type signature of each function.

Our implementation then runs the program on provided representative inputs and records the resulting parameter values on function entry and exit. To avoid generating an overwhelming amount of data, the current implementation caps the number of recorded calls at five.

The implementation then searches its database of candidate shards. It first searches the database to find shards with compatible types for the invoked functions. It then executes each candidate shard on the recorded parameter values to determine if the shard is (if desired, approximately) semantically compatible with one of the invoked functions. For each candidate shard it records timing information and uses that information to find the shard that performs best on the recorded inputs.

The final step is to substitute the candidate shard with the best performance in for the corresponding original function. Our current implementation supports both shard substitution at the compilation stage (leaving the source code intact), or shard substitution that changes the source code to explicitly invoke the replacement shard. Our current shard library is populated with shards that invoke implementations of linear algebra operations from the BLAS and LAPACK libraries.

It is also possible to create an interactive shard selection tool that would enable the developer to explore properties of various matching candidate shards, then (if desired) select a replacement shard. It is also possible to recombine shards dynamically, with the shard choice driven by dynamic values such as problem size or other characteristics. It is even possible to deploy shards speculatively — for example, first trying a more efficient shard that may handle fewer cases, then backing off to a less efficient but more reliable shard if the first fails.

C. Matrix Multiply

Matrix multiply is a standard computational kernel in many scientific computations. The textbook dense matrix multiply algorithm consists of a simple triply nested loop (Figure 1). But

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

```
#define index(M,width,i,j) M[i*width+j]

void multiply(double *A, int nA, int mA, double *B,
    int nB, int mB, double *C, int nC, int mC) {
  for (int i = 0; i < nA; i++) {
    for (int j = 0; j < mB; j++) {
      index(C,mC,i,j) = 0;
      for (int k = 0; k < mA && k < nB; k++) {
        index(C,mC,i,j) +=
          index(A,mA,i,k) * index(B,mB,k,j);
      }
    }
  }
}
```

Fig. 1. Simple Matrix Multiply Source Code

```
#define index(M,width,i,j) (M[i*width+j])

void mult(double *A, int mA, int nA, double *b,
    int nB, int i, double c) {
  for (int j = 0; j < nA; j++) {
    index(A, nA, i, j) *= c;
  }
  b[i] *= c;
}

void add_mult(double *A, int mA, int nA, double *b,
    int nB, int source, int dest, double mult) {
  for (int j = 0; j < nA; j++) {
    index(A, nA, dest, j) +=
      (index(A, nA, source, j) * mult);
  }
  b[dest] += b[source] * mult;
}

void solve(double *A, int mA, int nA, double *b,
    int nB, double *x, int nX) {
  for (int idx = 0; idx < nA; idx++) {
    double a = index(A, nA, idx, idx);
    int row = idx;
    mult(A, mA, nA, b, nB, row, 1.0/a);
    for (int i = row + 1; i < mA; i++) {
      double f = index(A, nA, i, idx);
      add_mult(A, mA, nA, b, nB, row, i, -f);
    }
  }
  for(int idx = nA - 1; idx > 0; idx--) {
    for (int i = idx - 1; i >= 0; i--) {
      double f = index(A, nA, i, idx);
      add_mult(A, mA, nA, b, nB, idx, i, -f);
    }
  }
  for (int i = 0; i < nB; i++) {
    x[i] = b[i];
  }
}
```

Fig. 2. Simple Linear Equation Solver Source Code

the field has invested substantial time and effort in developing complex, heavily optimized matrix multiply implementations. In this case study we consider reusable shards from the BLAS and LAPACK linear algebra packages, which include implementations of a variety of linear algebra operations including dense matrix multiply.

We start with a program that uses the standard triply nested matrix multiply loop (source code in Figure 1). Using two 1000x1000 matrices as input, we apply program fracture and recombination, which finds the BLAS 3 matrix multiply implementation and replaces the triply nested loop version with the BLAS version. Table I presents the performance results for a variety of matrix sizes. In general, the larger the matrix size, the larger the performance increase. At larger matrix sizes the shard replaced version runs over two times faster than the original version.

We note that the BLAS implementation may not be the most efficient implementation available (alternative, potentially more efficient implementations may be obtained, for example, by autotuning [5]). The goal is to start with a simple but potentially inefficient implementation, then identify the best available shard in the shard library (in this case the BLAS implementation). Populating the shard library with more efficient shards would enable larger performance improvements.

D. Linear Solver

Our linear solver benchmark generates and solves a system of linear equations determined by a 1000x1000 matrix and a 1000x1 vector. The original program implements a simple dense linear solver based on Gaussian elimination without pivoting (Figure 2). The replacement shard invokes the LAPACK linear solver. Table I presents the resulting performance results for a variety of matrix sizes. In general, the larger the matrix, the larger the performance increase available via the LAPACK implementation. The shard replaced version runs over two times faster than the original version for larger matrix sizes. Because it is more extensively engineered, the shard replaced version also exhibits superior numerical properties — we have found matrices for which the original version generates a solution vector of NaNs, while the replaced version generates an accurate solution.

E. Discussion

These two case studies illustrate how code fracture and recombination can enable performance increases and improve numerical properties. We are in effect proposing a new code search mechanism — instead of reading documentation or typing queries into a search engine, a developer would implement simple versions of the desired computations, then use code fracture and recombination to automatically find and replace the simple implementations with more complex implementations that feature better performance and numerical properties.

Researchers have spent significant effort developing optimized implementations of linear algebra operations. Specific techniques include explicit blocking for the cache hierarchy (with the goal of performing $O(N^2)$ computation out of the cache for $O(N)$ communication between the adjacent levels of the memory hierarchy) [6], cache-oblivious algorithms that use divide and conquer algorithms that naturally perform the majority of the computation out of the cache [7], and autotuning algorithms that explore an automatically generated space of implementations to find the one that (empirically) performs the best on the current hardware platform at hand [5].

Our goal is not to obtain new algorithms with better performance or other characteristics than algorithms obtained by these existing means. Our goal is instead to develop a system that helps programmers automatically find and use such sophisticated and efficient versions starting from a simple but inefficient version.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

III. Potential Uses

Our case studies emphasize the use of code fracture and recombination for improving performance (and potentially numerical properties) of software systems. We next discuss a variety of other uses for code fracture and recombination.

A. Software Transparency and Simplicity

One drawback of optimized code is its complexity and opacity — an efficient autotuned implementation of a basic operation such as FFT can include dynamic code generation and compilation, exploration of multiple alternatives, and multilingual implementations [8]. The opacity can make it difficult to understand the semantics of the code, how it operates, and the role that it plays in the computation. The complexity can induce extraneous dependences that impair portability and the ability to operate on specialized platforms. Systems that dynamically generate and compile different versions of the code (autotuners are a prominent example) may require the presence of a compiler, a significant requirement that not all computing platforms can satisfy. Appropriate fractures could automatically find a simpler, more transparent replacement with fewer dependences. Enhanced code transparency and portability are but two of the potential benefits.

B. Software Aging and Shard Rejuvenation

Of course, optimization can also become counterproductive over time — as the characteristics of the computational platform change, optimizations targeted at previous platforms often persist in the code even though they have no remaining purpose. Too often the result is a complex block of code optimized for an obsolete hardware platform. Another source of counterproductive software aging is obsolete functionality that remains (potentially disabled) in the current version of the software. Like obsolete optimizations, obsolete functionality can add complexity and obscure the purpose and operation of the software. Both obsolete optimization and obsolete functionality are simply examples of how undesirable software changes accrete over time, increasing software complexity and making it difficult to understand or modify the software.

This accretion has known negative consequences — as software systems age, they are known to become increasingly difficult for human developers to maintain [9]. At some point, the code becomes so difficult to change that the probability that a given change will introduce new unacceptable behavior exceeds the probability that the change will accomplish its desired goal.

Program fracture and recombination can help promote software rejuvenation. Identifying more modern, transparent, and relevant replacement implementations for specific shards, then replacing the aging original shard with the younger replacement shard can help extend the useful lifetime of the system.

C. Error Handling and Corner Case Code

Error handling and corner case code can be notoriously difficult to develop and get right, in part because the situations in which the code is relevant can be difficult to envision. Program fracture and recombination supports a model of development in which programmers write code that is only designed to handle the common case and elides the error handling and corner case code that can obscure the structure, purpose, and operation of the software. The system can then automatically discover and replace the original shard with a more elaborately developed shard that includes error handling and corner case code. It is also possible to leave the core computation itself in place, then enhance the computation with automatically located and transferred checks from other programs [1].

Note that this approach can support quickly engineered prototype components that are designed to work only in limited contexts or for limited use cases. A linear program solver taken directly from a textbook, for example, may work only for very small linear programs, with phenomena such as numerical instability crippling the solver for larger problems. But a textbook linear program solver can define the desired behavior for small problems, making it possible to identify and use a more intensively engineered replacement linear program solver. Given that developing a robust solver can take years or even decades [10], the potential productivity enhancements are substantial.

It is even possible to work with shards that implement only a few hard-coded cases or rely on humans to provide the desired functionality.

D. Multiple Code Views

One drawback of including error handling and corner case code is that it obscures the structure and purpose of the software, making the software more difficult to understand. One way to attack this problem is to replace the original complex code with a simpler shard that focuses on the common case. The concept is to automatically generate simpler, easier to understand code views with multiple perspectives designed to support multiple different purposes. Instead of a single monolithic code base that includes all of the code and functionality, whether relevant or not, this perspective promotes a fluid set of views designed to satisfy different needs and goals. One particularly noteworthy aspect is the fluid view of program semantics and the recognition that technically incorrect or incomplete programs are, for many purposes, more useful than more correct or complete programs. We have focused here on the elimination of potentially confusing error handling or corner case code as a way to enhance the ability of a developer to identify and work with the important core functionality of the program. But excess error handling and corner case code can also impair performance or introduce undesirable dependences on other software or system components.

E. Shard Analyzability

Complex code (whether resulting from optimization, aging, the inclusion of error handling and/or corner case functionality) can significantly impair the static analyzability of the program. Replacing this complex code with simpler shards can improve the analyzability of the program and promote the automatic extraction of useful information that can provide insight into the software and its properties. The success of accurate analysis stubs in enabling the successful information flow analysis of Android applications provides some indication of the potential for improved static analyzability that shard replacement can deliver [11].

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

One particularly intriguing aspect of program fracture and recombination is the potential it offers to simplify not just the code, but also the analysis results. Simpler code can be easier to analyze, enhancing the precision and scalability of the overall analysis. Sound static analyses reflect the complete semantics of the program. Substituting shards with simpler semantics can simplify the analysis results and enhance compositionality and scalability. Devising static analyses that analyze the program along several different axes (for example, effects on different parts of the program state and different aspects of the semantics) and using the axes to place the resulting analysis result within a projectable space of analysis results can promote the effective elimination of irrelevant analysis components. In this way the system can deploy even sound program analyses and use the analysis results to drive shard replacement algorithms that work with compatible but not semantically equivalent shards.

F. Enhanced Capabilities

Like optimization, capabilities such as distribution or the ability to operate safely in parallel contexts can require significant development effort. Program fracture and recombination can enable developers to write simple, less capable versions of core functionality, then automatically replace this functionality with more extensively engineered versions that can operate successfully in new contexts or exploit resources available in specialized contexts.

G. Functionality Elimination via Shard Removal

Programs may often contain components that have undesirable functionality. Security vulnerabilities [2], the ability to process a larger than desired set of inputs, logging or error reporting code that interacts with an obsolete subsystem, or even certain features such as support for SNMP that may no longer be desirable [3] can all be seen as undesirable functionality. Replacing the shard that implements the undesirable functionality with the null shard that does nothing can improve the program by eliminating the undesirable functionality. It may also be possible, of course, to find other replacement shards that implement the desired functionality but not the undesired functionality.

H. Functionality Enhancement via Shard Insertion

It is also possible to transfer shards directly into programs to obtain software hybrids with the combined functionality of both programs. This insertion would take a shard or shards from a donor and insert them into a recipient at an appropriate insertion point. There would be no replacement shard — the new shard or shards would enhance the recipient with new functionality not present before the insertion of the shard. CodePhage implements this shard transfer capability [1]. μScalpel transplants developer-identified procedures from a donor into a host [12].

I. Shards as Specifications and Code Search

Program fracture and recombination promotes the view of shards as specifications. It is often straightforward to code up enough of a computation to enable the program fracture and replacement system to find an appropriate replacement shard. In this way shards can serve as specifications that enable effective code search. Our two case studies demonstrate how this capability can be used to find sophisticated implementations of standard linear algebra operations.

J. Dependence Elimination

One of the primary goals of program fracture is to obtain shards that are freed from dependences they may have had on parts of the original program that should not be transferred with the shard. One way to realize this goal is to break the program into shards in such a way that each shard omits these dependences. In some cases this may require an intelligent fracture process that explicitly finds dependences on undesirable parts of the program, then eliminates the code that creates the dependence. The fracture may simply create a stub that breaks the dependence while still enabling the rest of the code with transitive dependences to execute, it may trace out and remove code that transitively depends on the excised dependence, or it may employ some combination of the two techniques.

IV. Compatible Shards and Adapters

Our current implementation works with compatible shards that have the same type signatures. There are several ways to generalize this technique to include more replacement shards. In general, we expect the automatic generation of adapters and shims that enable replacement shards that are not immediately precisely compatible. These techniques can also enable the automatic insertion of shards into contexts that require some adaptation before the shard can be automatically inserted [13].

Explicit Polymorphism: Some programming language constructs immediately support this kind of replacement. The use of explicitly polymorphic constructs such as C++ templates, for example, promotes the construction of specializable shards that can work in a variety of contexts. The use of parameter order canonicalization (by, for example, defining a total order on program types and working with adapters that expose the shard interface in that total order) can enable parameter reordering adapters that again enable the discovery and replacement of shards with otherwise incompatible parameter orders.

Data Structure Translators: The next step is data structure translation adapters. One example is adapters that perform array index translations to enable replacement shards to work with different array indexing schemes. One class of adapters changes the way the arrays are stored in memory, leaving the indexing code of the replacement shard in place. Another class leaves the storage scheme in place but statically analyzes and transforms the code of the replacement shard to work with the storage scheme of the program into which it is inserted.

It is also possible to leverage abstract data types — many abstract data types offer implementations of equivalent semantics with different performance or other tradeoffs. In some cases the developer may have used only a subset of the functionality of one abstract data type that could be implemented more efficiently by a less general but more efficient other abstract data type. In all of these cases the recombination can use the data structure semantics to recognize compatible shards from different data structures, with each shard in this case consisting of subsets of the abstract data types. This

978-1-4673-9286-0/15/$31.00 ©2015 IEEE

example shows that shards do not need to consist of single procedures or methods — they can be modules, related groups of code, or even arbitrary otherwise disconnected pieces of code extracted from single or multiple applications.

Specification-Based Compatibility: It is also possible to base compatability on specifications, either provided by the developer as part of the software development process or automatically inferred by an analysis (either static or dynamic). Shards with the same specification are interchangable. Factoring the specifications along different axes, where each axis captures a different aspect of the behavior of the shard such as side effects on different parts of the system, input or output effects, logging effects, or various aspects of the semantic properties of the shard.

V. External Dependences

Shards may often have external dependences on functionality present in their original context but not present in their new context. Examples include invoked modules and externally visible state such as global variables. These dependences may require importing into the recipient and/or conflict resolution along with appropriate initialization to operate properly.

It is possible to use program analysis to automatically identify and extract accessed global variables. The global variable may be initialized to an appropriate value either by program analysis or by recording values that appear in executions of one of the programs. Note that the program fracture and recombination system may need to pull in data structures such as hash tables and trees that store auxiliary data upon which the shard depends. Another approach is to trace the code in the original program that initializes the global variables or other state upon which the shard depends, then transfer and insert that code as well. Note that the process may be recursive — the initialization code may access files or other external resources. In that case the program fracture and recombination system can transfer the files or external resources, simulate the effect they have on the system via a spoofing or record/replay system, or even simply record the data structures that are constructed as a consequence of interacting with the external resources. In general, there is a chain of dependences from the resource interaction to the end result on the system, and it is possible for the program fracture and repair system to intercept the dependences anywhere along the chain and reconstruct (or alternatively potentially transitively excise) the effects.

The shard itself may also have a direct dependence on configuration files, normal files, or other external resources. It is possible to use the same techniques described above to deal with these dependences.

A more challenging situation occurs with dependences mediated via the operating system or some other opaque system. For example, some packages or system calls require initialization to operate successfully, but the dependences are controlled by data structures stored in the operating system or some other opaque system. Here the program fracture and recombination system may need some external knowledge of the dependence relationship so that it can, for example, find and replay the appropriate initialization calls, synthesize an appropriate initialization sequence, or do a search to find the sequence in the original program.

The shard may also have dependences on specific hardware components unavailable in other contexts or other unmovable resources. In this case the program fracture and recombination system can simply excise the dependence. In general, such dependence excision can be incorporated into arbitrary parts of the system and not just for hardware dependences. One of the goals of program fracture and recombination is to free useful pieces of code from dependences on the original context in which it appeared. Dependence excision, potentially following the dependences to completely eliminate the effects when appropriate, is therefore a key element of program fracture and recombination.

VI. Conclusion

We now have decades of investment in software systems, with desired functionality often implemented multiple times at varying levels of quality, performance, and capabilities. In principle, the vast majority of the functionality that most programs need has already been implemented. Our ability to profitably find and combine relevant pieces of functionality, as opposed to our ability to develop new software, may ideally become the limiting factor in software development. Program fracture and recombination promises to significantly enhance our potential in this critical area.

References

[1] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *PLDI*, Jun. 2015.

[2] M. Rinard, "Manipulating program functionality to eliminate security vulnerabilities," *Advances in Information Security*, vol. 54, Jul. 2011.

[3] H. H. Nguyen and M. C. Rinard, "Detecting and eliminating memory leaks using cyclic memory allocation," in *Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, Montreal, Quebec, Canada, October 21-22, 2007*, 2007, pp. 15–30.

[4] S. Sidiroglou-Douskos, E. Davis, and M. Rinard, "Horizontal code transfer via program fracture and recombination," Tech. Rep. MIT-CSAIL-TR-2015-012, Apr. 2015. [Online]. Available: http://hdl.handle.net/1721.1/96585

[5] R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *PPSC*, 1999.

[6] M. Lam, E. Rothberg, and M. Wolf, "The cache performance and optimizations of blocked algorithms," in *ASPLOS*, Apr. 1991.

[7] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, 1999.

[8] M. Frigo, "A fast fourier transform compiler," in *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, 1999, pp. 169–180.

[9] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 225–252, 1976.

[10] B. Murtagh and M. Saunders, "MINOS 5.51 user's guide," Tech. Rep. SOL-83-20R, 2003.

[11] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, "Information-flow analysis of Android applications in DroidSafe," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[12] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *ISSTA*, 2015.

[13] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2015.

978-1-4673-9286-0/15/$31.00 ©2015 IEEE