Data Flow Logic: Analyzing Information Flow Properties of C Programs

David Greve
Rockwell Collins
dagreve@rockwellcollins.com

Abstract

Understanding and analyzing information flow is crucial in the evaluation of security critical software systems. Data Flow Logic (DFL) is a domain specific language under development at Rockwell Collins for use in specifying and verifying dynamic information flow properties of such systems. The language employs C source code annotations to enable concise, consumable, abstract specifications of crucial information flow properties. A static analysis engine is also being developed to allow substantial portions of such specifications to be verified automatically. This paper highlights the motivation, methodology and status of the DFL tool suite.

1 Background

Rockwell Collins has substantial experience applying formal methods to the certification of information flow properties of high assurance computing systems. An early example of this is our work in formally verifying and certifying the AAMP7G. The AAMP7G is a microprocessor designed for use in embedded systems that provides a novel architectural feature, intrinsic partitioning, that enables the microprocessor to enforce an explicit communication policy between applications. Our verification of the AAMP7G involved formalizing the information flow properties expected from a separation kernel and verifying that these properties hold for the AAMP7G. A substantial challenge in doing this was modeling and reasoning about the pointer-rich, heap resident data structures manipulated by the kernel microcode[1]. In 2005 the AAMP7G was certified by the NSA as a MILS device, capable of simultaneously processing UNCLASSIFIED through TOP SECRET code word information[3].

Rockwell Collins also performed a semi-formal analysis of the Green Hills INTEGRITY-178B operating system. INTEGRITY-178B is a commercially available,

Research sponsored by Space and Naval Warfare Systems Command, Contract N65236-08-D-6805
Distribution Statement A: Approved for public release; distribution is unlimited.

high-assurance, partitioned operating system designed for safety and security critical systems[4]. The analysis involved formal descriptions of complex, heap resident data structures and semi-formal proofs supporting the claim that the operating system implementation, written in C, adhered to an information flow specification. This analysis satisfied the formal methods requirements for a Common Criteria EAL6+ certification, which was awarded in 2008[2].

Our involvement in the certification of these high-assurance separation kernels led to the identification of several challenge areas where improvements were possible in the specification, maintenance, analysis and evaluation of such systems. These challenge areas included modeling, policies, specifications and analysis. In addressing these challenges Rockwell Collins is developing a domain specific annotation language called Data Flow Logic (DFL) that supports information flow modeling and analysis of source code expressed in the C programming language. As an annotation language[5,6], DFL augments the C programming language with domain specific assertions that can be extracted from the source code and serve as a program specification. DFL extends the GCC attribute specifier construct rather than utilizing specialized comments or changing the C language syntax. DFL's domain specific assertions pertain to the manner in which information is communicated and shared during program execution. DFL makes extensive use of the C type system to allow concise, easily consumable specifications of crucial separation properties in a manner that can be modeled and reasoned about formally. Both the design of DFL and methodology it codifies were strongly influenced by our previous certification experiences. The following sections overview the operation of DFL in the areas of modeling, policies, specifications and analysis.

2 Modeling

A formal model of a computational system is a mathematical representation of that system. The computational system is the target of the evaluation and is represented by the source code making up that system. The creation of a formal model comprises a translation or interpretation that describes how the target works, starting with a non-mathematical description (the source code) and producing an appropriate mathematical representation (the formal model).

Previous Rockwell Collins modeling methodology involved the manual construction of a formal model from the original source code. The DFL framework supports automated model generation directly from C source code using the C Intermediate Language (CIL) compiler framework. CIL is a mature collection of C processing tools developed at the University of California at Berkley that includes a parser and a variety of transformation capabilities[9]. CIL performs several semantics preserving simplifications on the source code to produce a representation of the original program in a “clean” subset of C amenable to formal analysis.

The CIL framework represents the source code internally in an abstract-syntax tree (AST) format typical for language compilers. The DFL framework includes an extension to CIL that emits this internal representation as an ACL2 data structure. ACL2 stands for A Computational Logic for Applicative Common Lisp and it is the name of a theorem proving system whose underlying logic is a subset of the Common Lisp programming language[8]. Our ACL2 framework has been augmented to efficiently read, process and write such data structures. Within the ACL2 framework it is also possible to give formal semantics (meaning) to the AST representation of the source code. The ACL2 AST representation is therefore our formal model of the source code.

Leveraging this framework allows properties of the formal model (and thus the original source code) to be verified using program reasoning techniques from previous Rockwell Collins verification efforts. It also enables the construction of the DFL analysis framework within the programming logic of the ACL2 theorem prover. The fact that DFL is implemented in the logic of the ACL2 theorem prover means that it is possible to prove the correctness of the static analysis capabilities of DFL, endowing them with the same formal pedigree as a proof about the original source code itself.

3 Policies

Formal policies are expressions of the important operational properties of a computing system. Data flow policies, for example, describe how software is allowed to move information within the system. Good policies are simpler to understand than the implementation model they describe. This is accomplished through the use of specifications that abstract away implementation details and unwind optimizations. The

result is a policy that helps developers and evaluators focus on overall system behavior rather than on implementation details.

Policies also act as contracts, imposing obligations on procedure implementations and providing guarantees at procedure boundaries. The obligation that an implementation must satisfy its contract requires that the contract be verifiable. Verifiable means that it is possible to prove that the source code model satisfies the policy. The fact that a contract guarantees certain behaviors enables other system components to leverage those claims to satisfy their own obligations. The challenge of crafting a good policy is one of balancing these competing interests. Policy descriptions need to be abstract enough for developers and evaluators to understand but detailed enough to serve as useful and verifiable contracts.

Security policies are often expressed in terms of the kinds of information flow allowed between different security domains. Analyzing information flow in secure systems therefore requires identifying the security domains of interest and then classifying the state of the system according to those domains. DFL provides a mechanism for giving names to security domains. These names can then be used to articulate security policies. Domains in DFL are represented as simple program variables. Identifying a program variable as a domain is as simple as declaring the variable using the DFL_DOMAIN macro. The code snippet in Figure 1 is an example of using the DFL_DOMAIN macro to declare a domain named *Secret*.

```
DFL_DOMAIN  Secret ;
```

Figure 1: Simple Domain Declaration

In DFL the most general policy description for a given procedure involves the specification of two additional procedures: one to express preconditions and another to express postconditions. A precondition procedure is a void procedure whose body consists of a sequence of assertions that may be used to restrict the set of states that must be considered when attempting to show that the implementation satisfies the contract.

A postcondition procedure is a void procedure whose body consists of a sequence of assertions that describe the policy implemented by the procedure. A specialized

assertion, DFL_DEPENDS(x,y), is used to declare information flow relationships between domains. In DFL the specification for the information flow policy is combined with the other post conditions into the post condition procedure body.

4 Specifications

A specification can be used to map the behavior of the system into terms and idioms appropriate to the application domain. For secure systems, information flow is a key property. Essential to specifying the information flow properties of a system are descriptions of the security domains resident in the system, how data is stored in memory and how it maps into the relevant security domains, the kinds of data structures used in storing data, and the invariants and relationships that are required to ensure secure operation. In many systems, pointer rich heap resident data structures constitute the bulk of the system state. Analyzing information flow in such systems requires that these data structures be classified according to the security domains in which they reside.

Domains can be thought of as collections of program variables, portions of heap allocated data structures and sometimes other domains. Security domains may encompass entire data structure hierarchies or they may be as precise as specific fields within a particular data structure. Mapping a program variable into a domain can be accomplished using the DFL_WITHIN macro. In Figure 2 the DFL_WITHIN annotation asserts that variable *key* resides within the *Secret* domain.

```
int key DFL_WITHIN((Secret)) ;
```

Figure 2: DFL_WITHIN Example

Often a discontinuity exists between the architectural view of a system and the implementation of that system. Such discontinuities arise for a variety of reasons including the need for implementation efficiency and the desire to preserve modularity. In DFL, the process of unwinding optimizations and determining which variables to assign to which domain under what conditions is called *classification*. The classification process captures an understanding of the system and provides a context within which

information flow policies can be expressed and verified. The end result of the classification process is a *classification procedure*.

DFL supports two kinds of classification procedures. A *global* classification procedure is used to classify global variables. Global classification procedures can be identified using the DFL_GLOBAL_CLASSIFICATION macro. The body of a global classification procedure consists of a sequence of global variable declarations and associated attributes. Every declaration appearing in the body of the procedure extends, and thus must match, an existing global declaration (modulo DFL attributes). Figure 3 illustrates a global classification procedure, *GClass*, that classifies a single global variable, *key*, as residing in domain *Secret*.

```
int key;

DFL_DOMAIN Secret;

DFL_GLOBAL_CLASSIFICATION GClass() {
  int key DFL_WITHIN((Secret));
};
```

Figure 3: Global Classification Procedure Example

A *heap* classification procedure is used to group into domains pointers and the heap objects they identify. Heap classification procedures are identified using the DFL_HEAP_CLASSIFICATION macro. The first argument to a heap classification procedure is a void pointer. The body of a heap classification consists of a sequence of possible type declarations for the void pointer named in the first argument of the classification. All type declarations in the body of the procedure must match an existing type declaration (modulo DFL attributes) because they extend the original declaration. Heap classifications are *initiated* by following non-null pointers. We refer to the process of classifying heap-oriented, pointer laden data structures as *crawling* the data structures. This behavior is analogous to the crawler functionality employed in earlier specification efforts[1]. Attaching a specific heap classification procedure to a pointer is done using the DFL_CRAWL macro. Figure 4 illustrates a heap classification procedure that maps every field in every element in a linked list defined by the structure *list* into the domain *Data*.

```
typedef struct list {
  int val;
  struct list *next;
} list;

DFL_DOMAIN Data;

DFL_HEAP_CLASSIFICATION HClass(void *x) {
  struct list {
    int val            DFL_WITHIN((Data));
    struct list *next DFL_WITHIN((Data))
                       DFL_CRAWL((HClass(next)));
  } *x;
};
```

Figure 4: Heap Classification Procedure Example

5 Analysis

Software verification is the process of checking software systems for conformance with a given set of properties. There are many mechanical means of checking software properties[6]. The most comprehensive and rigorous form of verification is called formal verification. Historically Rockwell Collins has used theorem proving systems to formally verify information flow contracts for secure software systems. One important outcome of our previous certification efforts was the development of a useful, mathematical formalization of security policies. This formalization, referred to in the literature as the GWV[10] theorem (named for its original authors Greve, Wilding, and Vanfleet), ensures such critical security properties as an absence of exfiltration, infiltration, and mediation. An important quality of the GWV theorem is that it can both specify a given implementation and be used as a contract for that implementation in the context of a larger system.

Our use of a theorem proving system to verify information flow properties is justified by the fact that some of the properties being checked are undecidable. Nonetheless, we believe that 90% of all information flow properties can be decided statically, leaving only 10% that require more powerful reasoning techniques. A significant challenge, therefore, is to make the analysis effort proportional to the difficulty of the task at hand.

To leverage this opportunity, DFL has been augmented with a static information flow analysis capability that allows it to decide many information flow policies automatically.

We believe that this will save both time and effort, focusing the analysis efforts on the portion of the problem that actually requires human ingenuity to solve.

Analysis in DFL is initiated by way of a contract statement. A contract binds a postcondition policy statement to a procedure under a set of preconditions. Associating a contract with a procedure obliges an implementation of that procedure to satisfy the terms of the contract in any state that satisfies the preconditions. Conversely, an analysis engine may appeal to the terms of a procedure's contract following an invocation of that procedure on any state satisfying the preconditions. Contracts therefore provide both rigorous guidelines and useful behavioral information for the developers and integrators of software systems. A contract statement called *BadBoy_contract* for the procedure named *BadBoy* with precondition *BadBoy_requires* and postcondition *BadBoy_provides* might appear as in Figure 5.

```
DFL_CONTRACT void BadBoy_contract (list *p1, list *p2)
  DFL_INSTANCE(BadBoy(p1,p2))
  DFL_REQUIRES(BadBoy_requires(p1,p2))
  DFL_PROVIDES(BadBoy_provides(p1,p2))
  ;
```

Figure 5: BadBoy Contract Example

The DFL static analysis engine has been shown to be capable of correctly analyzing information flow contracts for C programs, including several procedures selected from the open source operating system Minix[11,12]. It has detected faulty contracts and verified correct contracts. Additionally, the kinds of contracts analyzed by the engine are beyond the scope of many existing static information flow analysis tools such as SPARK Ada since they involve information flow between heap allocated objects accessed via pointers. Figure 6 provides a pictorial representation of a portion of one of the procedures verified using the DFL static analyzer. The different colors in the picture correspond to different domains. Note that the use of pointers and heap resident data structures introduces subtle information flows that must be appropriately accounted for in such analysis. The very subtle nature of such dependencies supports the need for rigorous automated analysis of such systems.

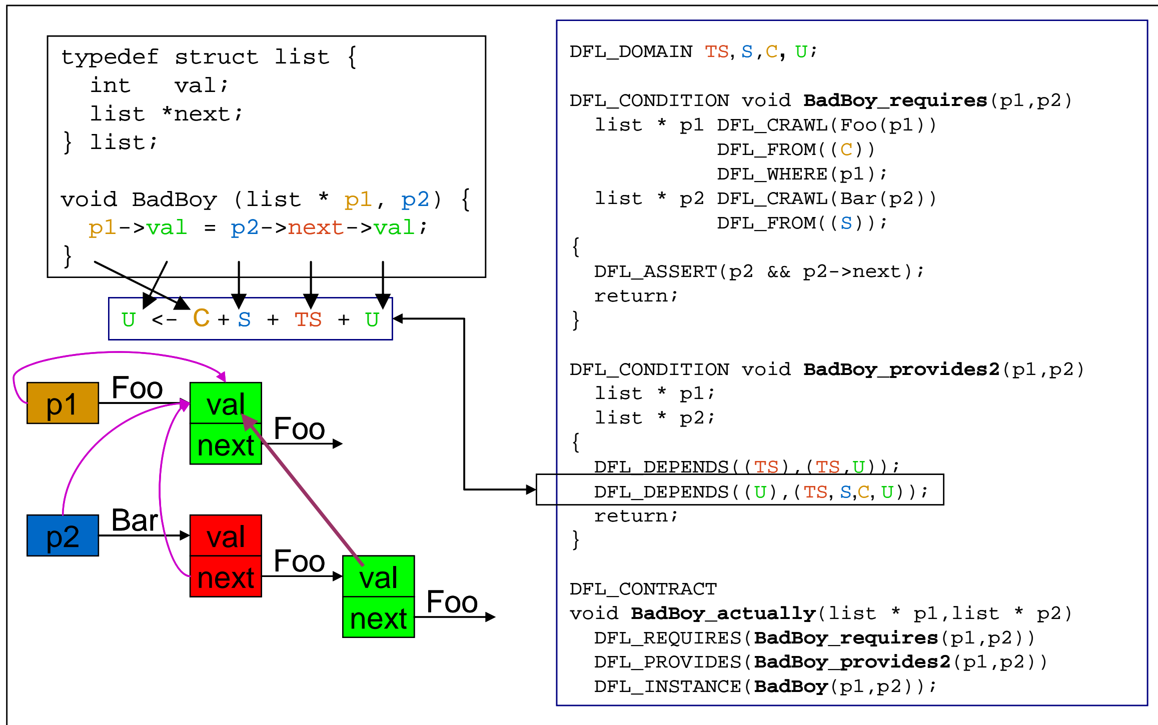

Figure 24: BadBoy Verified Contract

6 Conclusion

The DFL tool suite is currently in a demonstrable prototype phase. Although not fully featured, it has already been used to analyze procedures that manipulate heap resident objects as well as a small collection of kernel procedures from the Minix operating system[12]. It has been tested on several examples and has succeeded in establishing contracts that were known true, suggesting that the technology is useful, and it has failed on contracts that were known to be false, reflecting the soundness of the approach. Our previous experiences suggest that the enhancements embodied in DFL will result in increased automation, improved maintainability and enhanced understanding of both the system under evaluation and its specification, ultimately resulting in both lower cost and more timely certifications of high assurance systems.

References

[1] D. Greve, R. Richards, and M. Wilding, “A summary of intrinsic partitioning verification”, Fifth International Workshop on the ACL2 Prover and Its Applications (ACL2-2004), 2004.

[2] “Green Hills Software Announces World’s First EAL6+ Operating System Security Certification”, Green Hills Software press release, November 17, 2008.

[3] Wilding, M.M, Greve, D.A, Richards, R.J., Hardin, D.S, “Formal Verification of Partition Management for the AAMP7G Microprocessor”, In Design and Verification of Microprocessor Systems for High-Assurance Applications, David Hardin Ed., Springer, ISBN 978-1-4419-1538-2, pp 175-192

[4] Richards, R, “Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel”, In Design and Verification of Microprocessor Systems for High-Assurance Applications, David Hardin Ed., Springer, ISBN 978-1-4419-1538-2, pp 301-322

[5] Leavens, Gary T, Baker, Albert L, “Enhancing the Pre- and Postcondition Technique for More Expressive Specifications”, 1999, Proceedings of the World Congress on Formal Methods in the Development of Computing Systems, Volume II, pp 1087-1106, Springer-Verlag.

[6] Barnes, J, “High Integrity Software: The SPARK Approach to Safety and Security”, 2003, Addison-Wesley Longman Publishing Co. Inc, ISBN 0-321-13616-0

[7] Data Flow Logic Language Overview, Rockwell Collins Internal Document

[8] http://www.cs.utexas.edu/~moore/acl2/

[9] http://www.eecs.berkeley.edu/~necula/cil/

[10] Greve, D, Wilding, M, Vanfleet, WM, “A Separation Kernel Formal Security Policy”, Fourth International Workshop on the ACL2 Prover and Its Applications (ACL2-2003), Boulder, CO, July 2003.

[11] Greve, D, “Information Security Modeling and Analysis”, In Design and Verification of Microprocessor Systems for High-Assurance Applications, David Hardin Ed., Springer, ISBN 978-1-4419-1538-2, pp 249-300

[12] Tanenbaum, Woodhull, “Operating Systems Design and Implementation,” 3rd ed., Pearson Prentice Hall 1987.

[13] http://www.minix3.org/