Property Templates for Checking Source Code Security

Elizabeth I. Leonard
Naval Research Laboratory
Washington, DC
elizabeth.leonard@nrl.navy.mil

Myla M. Archer
Naval Research Laboratory
Washington, DC
myla.archer@nrl.navy.mil

Constance L. Heitmeyer
Naval Research Laboratory
Washington, DC
constance.heitmeyer@nrl.navy.mil

ABSTRACT

This paper describes a method for using property definition templates to support automatic analysis of source code for application-specific security properties. The method is illustrated on an example data flow property of a C program.

CCS CONCEPTS

• **Security and privacy** → **Logic and verification**; **Software and application security**; Information flow control;

KEYWORDS

application-specific security properties, template-based specification, verification

ACM Reference Format:
Elizabeth I. Leonard, Myla M. Archer, and Constance L. Heitmeyer. 2017. Property Templates for Checking Source Code Security. In *Proceedings of MEMOCODE '17, Vienna, Austria, September 29-October 2, 2017,* 4 pages.
https://doi.org/10.1145/3127041.3127063

1 INTRODUCTION

This paper describes a method for using property definition templates to support automatic analysis of source code for application-specific security properties. The method is based on automatically generated code assertions. Section 2 introduces an example C program for analysis, and discusses the nature of assertions, how assertions are generated, and how templates are used. It illustrates the use of our method, as well as its ability to handle property interactions, by applying the method to a data flow property for an example program in which a sanitization property is also desired to hold. Section 3 discusses related work and planned future work.

2 VERIFICATION METHOD

We illustrate our verification method on `Program 1` in Fig. 1, a C program processing data at two security levels (in arrays `high` and `low`). The method must ensure that data classified at the high security level does not flow to memory areas classified at the low security level. High and low data are processed separately, but a shared memory area `share` is used in both calls to `compute` during individual processing of data from both `high` (line 23) and `low` (line 27).

This research was sponsored by the Office of Naval Research.

MEMOCODE '17, September 29-October 2, 2017, Vienna, Austria
2017. ACM ISBN 978-1-4503-5093-8/17/09…$15.00
https://doi.org/10.1145/3127041.3127063

```
1  void process(int *mem) { }
2  void init(int *mem) { }
3  void get_in(int *part) { }
4  void put_out(int *part) {}
5  void cleanup (int *mem) {}
6
7  void compute (int *part, int *work) {
8     int i;
9     init(work);
10    for (i=0; i<10; i=i+1) {
11      work[i]= work[i] * part[i]; };
12    process(work);
13    for (i=0; i<10; i=i+1) {
14      part[i]=work[i]; }; }
15
16 int main() {
17   int high[10];
18   int low[10];
19   int share[10];
20   cleanup(share);
21   while(1) {
22     get_in(high);
23     compute(high, share);
24     put_out(high);
25     cleanup(share);
26     get_in(low);
27     compute(low, share);
28     put_out(low);
29     cleanup(share); };
30   return(1); }
```

Figure 1: `Program 1`

For associating assertions with lines of code, SecProve invokes a preprocessor that translates (a disciplined subset of) C code into our high level intermediate language LEMA [2]. LEMA is a language of **while** programs with procedures, having 1) `VAR` (input/output) and non-`VAR` (input only) parameters and 2) contracts consisting of two assertions—a precondition and a postcondition. In LEMA, the lines of code generated from the source code provide a natural target line for every generated assertion. Global variables in the C program become local variables in procedure `main` and `VAR` parameters to all other procedures in the LEMA representation. Fig. 2 shows the LEMA representation of the non-stub routines of `Program 1`.

2.1 Assertion-Based Verification

A major motivation for our approach to proving application-specific security properties of code is our success with the use of code assertions in proving data separation in an embedded software device [7]. An assertion associated with a particular line of code in a program makes a statement about the execution state of the program when that line of code is reached. The use of code assertions in [7] was inspired by the use of code assertions in standard program verification [5, 6]. The work described in [7] illustrated the need for automating assertion generation, both to reduce the required verification effort and to ensure that assertions are generated correctly.

```
PROC compute (VAR part: [nat -> int], VAR work: [nat -> int])
    part_save: [nat -> int] = part;
    work_save: [nat -> int] = work;
    VAR i : int;
L0:     SKIP;
L1:     EXECUTE(init(VAR work));
L2:     i := 0;
L3:     SKIP;
L4:     WHILE (i < 10) DO
L5:       BEGIN
L6:         work := work WITH [(i) := work(i) * part(i)];
L7:         i := i + 1;
L8:       END
L9:     ENDWHILE;
L10:    EXECUTE(process(VAR work));
L11:    i := 0;
L12:    SKIP;
L13:    WHILE (i < 10) DO
L14:      BEGIN
L15:        part := part WITH [(i) := work(i)];
L16:        i := i + 1;
L17:      END
L18:    ENDWHILE;
L19:    RETURN;
L_omega:
ENDPROC

PROC main (VAR  _res : int)
    VAR high : [nat -> int];
    VAR low : [nat -> int];
    VAR share : [nat -> int];
L0:     SKIP;
L1:     EXECUTE(cleanup(VAR share));
L2:     SKIP;
L3:     WHILE (1 != 0) DO
L4:       BEGIN
L5:         EXECUTE(get_in(VAR high));
L6:         EXECUTE(compute(VAR high, VAR share));
L7:         EXECUTE(put_out(VAR high));
L8:         EXECUTE(cleanup(VAR share));
L9:         EXECUTE(get_inp(VAR low));
L10:        EXECUTE(compute(VAR low, VAR share));
L11:        EXECUTE(put_out(VAR low));
L12:        EXECUTE(cleanup(VAR share));
L13:      END
L14:    ENDWHILE;
L15:     _res := (1);
L16:    RETURN;
L_omega:
ENDPROC
```

Figure 2: LEMA representation of `Program 1`

Our security property proofs based on code assertions differ in one important way from the functional correctness proofs in standard program verification. Functional correctness proofs are based on *value assertions* stating facts about the values of program variables. Security property proofs typically use *history assertions* stating facts about events that have (or have not) occurred during program execution prior to the line of code with which the assertion is associated. E.g., for a sanitization property, a memory area has been (and remains) sanitized; for a nonbypassability property, a certain segment of code has been executed; for a data flow property, a certain data flow has not occurred. Proving either functional correctness or security properties using code assertions relies on two types of assertions: *Desired* (which need to be checked) and *Known* (which need not be checked). Known assertions may be assumed (as preconditions) or generated by forward propagation (e.g., after an assignment $x := 2$, $x = 2$ becomes a Known assertion). To establish a property, one must prove the Desired assertions.

The proof goal in establishing a functional correctness property is generally clear: prove the (Desired) output assertions given the (Known) input assertions. In contrast, the proof goal in establishing a given security property needs to be specified. To aid a developer in doing so, SecProve provides a set of property definition templates. Associated with each template are assertion generation rules that tell

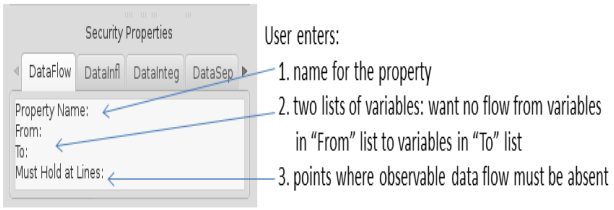

Figure 3: A data flow template

SecProve's assertion generator how to either *place* or *propagate* assertions (Desired and Known) relevant to proving the property. A filled-in template thus captures the property's proof goal. It also gives concrete names to entities such as history predicates and program variables and indicates line labels where assertions should be placed.

2.2 Specifying Properties using Templates

SecProve provides templates for a variety of security properties. Fig. 3 shows one of SecProve's data flow templates. To specify a data flow property, the user provides: 1) a property name, 2) a specification of the prohibited data flow by identifying a set of variables (`From` :) whose information may not flow to a second set of identified variables (`To` :), and 3) the points in the code at which the data flow property must hold (e.g., for the template of Fig. 3 the user will need to provide a list of lines in the code[1]). This is the only work required of the developer in the verification process—everything else is done automatically.

Fig. 4 shows a completed template specifying data flow property `no_high_to_low` for `Program 1`. The security property of interest is that data does not flow from variable `high` to variable `low`, both defined in `main`. The template specifies that the property will be checked at label `L11` of `main`, the call to procedure `put_out` with `low` as parameter.

Analysis of a data flow property may require information from additional templates—e.g., those specifying sanitization properties. The purpose of sanitization is to ensure that no trace of sensitive data remains in the sanitized memory area. Sanitization cleanses a memory area of any relationship to any value previously stored there; thus any data that flowed to that area is erased. Reference [3] describes how a program can be analyzed for a sanitization property.

What sanitization means is application-dependent; different programs may use different methods of sanitization (e.g., writing all zeroes in the memory area to be sanitized or writing random patterns of zeroes and ones to the area). Thus, the developer must identify the code that performs sanitization. The specification of a sanitization property includes: 1) a property name, 2) identification of the procedure performing sanitization, the subset of its parameters (which may include global variables) that it sanitizes, and the data types of those areas, 3) the name of a predicate to use in reasoning about sanitization, and 4) identification of the locations where sanitization must hold and the memory areas that must be sanitized at those locations.

[1]This requires access to the LEMA representation of the C program.

Figure 4: Data flow property for `Program 1`

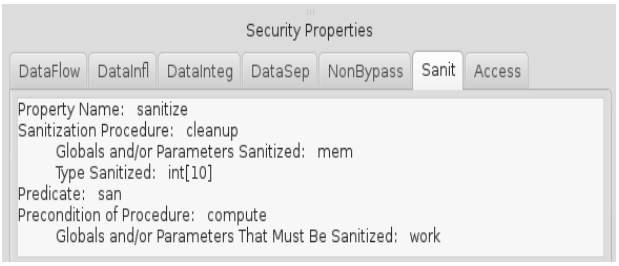

Figure 5: Sanitization property for `Program 1`

LABEL	PASS	to(high) =	to(low) =	to(share) =
L0-L5	1	∅	∅	∅
L6	1	{high}	∅	∅
L7-L8	1	{high, share}	∅	{share, high}
L9	1	{high, share}	∅	∅
L10	1	{high, share}	{low}	∅
L11-12	1	{high, share}	{low, share}	{share, low}
L13	1	{high, share}	{low, share}	∅
L3-L6	2	{high, share}	{low, share}	∅
L7-L8	2	{high, share}	{low, share}	{share, high}
L9-L10	2	{high, share}	{low, share}	∅
L11-L12	2	{high, share}	{low, share}	{share, low}
L13	2	{high, share}	{low, share}	∅
L3	3	{high, share}	{low, share}	∅
L14-15	1	{high, share}	{low, share}	∅
L_omega		{high, share}	{low, share}	∅

Figure 6: Known data flow assertions for `main`

LABEL	PASS	to(part) =	to(work) =
L0-L1	1	$\texttt{to(part)}_{save}$	$\texttt{to(work)}_{save}$
L2-L6	1	$\texttt{to(part)}_{save}$	{work} ∪ $\texttt{to(work)}_{save}$
L7-L8	1	$\texttt{to(part)}_{save}$	{work} ∪ $\texttt{to(work)}_{save}$ ∪{part} ∪ $\texttt{to(part)}_{save}$
L4-L8	2		
L4	3		
L9-L15	1		
L16-L17	1	{part} ∪$\texttt{to(part)}_{save}$ ∪{work} ∪$\texttt{to(work)}_{save}$	{work} ∪ $\texttt{to(work)}_{save}$ ∪{part} ∪ $\texttt{to(part)}_{save}$
L13-L17	2		
L13	3		
L18-L19	1		
L_omega			

Figure 7: Known data flow assertions for `compute`

Returning to `Program 1`, Fig. 5 shows a completed template specifying sanitization property `sanitize` for the program. The template identifies `cleanup` as the sanitization routine, `mem` as the parameter that is sanitized, and the data type sanitized as an integer array of size 10. The template specifies `san` as the predicate to be used when generating assertions and states that sanitization of `compute`'s parameter `work` must hold whenever `compute` is called.

2.3 Generating Assertions

Assertion generation for a data flow property of a program generates assertions for each LEMA procedure individually to infer its data flow contract. Because of the transitive nature of data flow, assertions must be generated for all parameters and local variables of a procedure simultaneously.

At each LEMA label, the set $\texttt{to}(x)$, representing the set of variables from which data has flowed to x, is computed for each variable x. In computing the data flow contract of a procedure `proc`, every parameter x of `proc` is conceptually supplemented with an auxiliary parameter $\texttt{to}(x)$ whose actual parameter becomes $\texttt{to}(a)$ whenever `proc` is called with actual parameter a. The notation $\texttt{to}(x)_{save}$ is used to represent the value of this auxiliary parameter when `proc` is invoked, and allows the data flow postcondition of `proc` to be specified in terms of the data flows existing when `proc` is called. The postcondition for a procedure `proc`, placed at label `L_omega` (the last line of the LEMA representation of the procedure), is computed by taking the assertions at the `RETURN` statement and 1) removing `to` sets for any local variables and non-`VAR` parameters of `proc`, and 2) for each `VAR` parameter of `proc`, removing any elements from its `to` set that are not `VAR` parameters or $\texttt{to}(x)_{save}$ variables.

Placement Rules. Assertion generation begins by placing a set of assertions derived from information in the completed template and some assumptions about data flow properties. Desired assertions are placed at locations at which the completed template specifies the property must hold. For `Program 1`, the (only non-trivial) Desired assertion $\texttt{high} \notin \texttt{to(low)}$ is placed at `main : L11`.

When variables are first declared, no data has flowed to them. For every variable x declared in a procedure `proc`, the Known assertion $\texttt{to}(x) = \emptyset$ is placed at line `proc : L0`. For `Program 1`, the following Known assertions are placed: $\texttt{to(high)} = \emptyset$, $\texttt{to(low)} = \emptyset$, and $\texttt{to(share)} = \emptyset$ at `main : L0` (see first row in Fig. 6) and $\texttt{to(i)} = \emptyset$ at `compute : L0`.[2]

For every formal parameter x of a procedure `proc`, the Known assertion $\texttt{to}(x) = \texttt{to}(x)_{save}$ is placed at line `proc : L0`, indicating that the initial data flows to a procedure parameter are the flows to the corresponding actual parameter at the call sites. For `Program 1`, Known assertions $\texttt{to(part)} = \texttt{to(part)}_{save}$ and $\texttt{to(work)} = \texttt{to(work)}_{save}$ are placed at `compute : L0` (see the first row of Fig. 7), and $\texttt{to(\_res)} = \texttt{to(\_res)}_{save}$ at `main : L0`.

For stub routines (i.e., routines with no body), we make the pessimistic assumption that all possible flows can occur between all variables to which the routine has access. The Known assertion `TRUE` is placed at `L0` for each stub routine. For each `VAR` parameter x of a stub routine, a Known assertion is placed at `stub : L_omega` stating that x's `to` set contains the union of every parameter y and $\texttt{to}(y)_{save}$. For a stub routine declared to be a sanitization routine, any parameter that is sanitized has the empty set as its `to` set in the contract postcondition. Fig. 8 shows the data flow contracts for the stub routines of `Program 1`.

[2] Assertions about the `to` sets for variable i in `compute` and `_res` in `main` are omitted from the tables as they don't impact analysis of the data flow property. To reason about loops, column PASS tracks visits to a line of code.

PROCEDURE	PRE	POST
process	TRUE	$\texttt{to(mem)} = \{\texttt{mem}\} \cup \texttt{to(mem)}_{save}$
init	TRUE	$\texttt{to(mem)} = \{\texttt{mem}\} \cup \texttt{to(mem)}_{save}$
get_in	TRUE	$\texttt{to(part)} = \{\texttt{part}\} \cup \texttt{to(part)}_{save}$
put_out	TRUE	$\texttt{to(part)} = \{\texttt{part}\} \cup \texttt{to(part)}_{save}$
cleanup	TRUE	$\texttt{to(mem)} = \emptyset$

Figure 8: Data flow contracts for stub routines

For every procedure call, a Desired assertion derived from the called procedure's precondition (all assertions at label L0 related to the procedure's parameters) is placed at the label associated with the call site. For Program 1, the Desired assertions placed at call sites are trivially true—TRUE for calls to stub routines, $(\texttt{to(high)} = \texttt{to(high)}_{save}) \wedge (\texttt{to(share)} = \texttt{to(share)}_{save})$ for the call to compute at main : L6, and a similar assertion with respect to low and share for the call to compute at main : L10.

Propagation Rules. After initial placement of assertions, forward propagation rules generate additional assertions. The to set for a variable x passes unchanged through a SKIP statement (e.g., to sets at L1 are the same as at L0 in Fig. 7) and through an assignment to a variable y that is not equal to x (e.g., Fig. 7 shows that the to set for part in compute is unchanged at L7 after the assignment to work is executed). For an assignment statement $x := f(y_1, .., y_m)$, the data flows to x are the variables appearing on the right hand side of the assignment statement and the flows to those variables at that point in the procedure. (See the to set for work at label L7—just after the assignment to work—in Fig. 7.) Note that if x appears on the right hand side of the assignment statement (i.e., if, for some $1 \leq j \leq m$, y_j is x) then x and all the data flows to x at the assignment statement are included in the data flow to x at the program location after the assignment statement. For x of array type, because we treat arrays as monolithic entities, retaining x in $\texttt{to}(x)$ ensures that flows to all individual array elements are carried forward in the assertion generation.

For branching statements (i.e., if-then-else and while loops), the to set for a variable x is the union of x's to sets forwarded from each of the branches. For the branch of a while loop that executes the loop, data flow computations through the loop will eventually reach a fixpoint that can be used as the strongest postcondition. This allows computations of data flows for even non-terminating while loops to be computed, but could result in an over-approximation of the data flows if the number of iterations needed to reach a fixpoint exceeds the number of times the loop code would execute if unwound. In Fig. 6, observe that the to sets at L3 are the same for the second and third passes, and thus, this is the point where we reach a fixpoint. The to set for each variable propagated to L14 is the union of the variable's to set at the first visit to L3 (loop not executed) and the final pass at L3 (loop executed).

If a procedure call does not include x as a VAR parameter, then the data flow assertion for x passes through the procedure call unchanged. This is true even if x is a non-VAR parameter for the procedure call since non-VAR parameters are not changed by procedure calls. If x is a VAR parameter in the procedure call, then the postcondition of the data flow contract for the procedure determines the data flows to x at the point immediately after the procedure call. (E.g., the to set for high at L6 in Fig. 6 is derived from the postcondition of get_in the procedure called at L5.)

Analyzing procedures compute and main produces the Known data flow assertions shown in tabular form in Figs. 7 and 6. The contract postcondition for each procedure is the assertion at L_omega. The Desired assertion $\texttt{high} \notin \texttt{to(low)}$ at main : L11 is evaluated against each to set for low at main : L11. The property holds with respect to the Known assertions placed at main : L11 on both passes through the loop. Note that the property only holds because the call to cleanup at L8 of main removes all flows to share, and thus, prevents the flow of data from high to low.

3 RELATED AND FUTURE WORK

Related Work. There has been considerable research on data flow at the low level (e.g., hardware and assembly code). In this short paper, we describe related work on the analysis of source code for data flow properties. The seminal work of Bergeretti et al. [4] provides rules for reasoning about data flow and information flow in while programs. Our work, in contrast, extends these rules to allow tracing of data flow through procedures. In [1], Amtoft et al. introduce a method based on code assertions for analyzing conditional data flow properties of procedure contracts. Their analysis method uses assertions of a different style from ours—*agreement assertions*—that capture agreement of two states on the values of specific variables. In contrast to the data flow analysis techniques of other authors, our technique based on property templates and code assertions can be used to verify many other application-specific security properties such as sanitization, nonbypassability, and data separation.

Future Work. We have already made progress in improving analysis precision by extending our generated assertions to include conditional assertions resulting from tests in the code. A problem to be solved is dealing with the computational complexity of the analysis that can result from multiple nested code branches. Another future effort will adapt our methods to handle more extensive use of pointers.

REFERENCES

[1] Torben Amtoft, John Hatcliff, et al. 2008. Specification and Checking of Software Contracts for Conditional Information Flow. In *Proc. 15th World Cong. on Formal Methods (FM 2008) (Lect. Notes in Comp. Sci.)*, Vol. 5014/2008. 229–245.

[2] M. Archer, E. Leonard, and C. Gasarch. 2013. *Verifying LEMA Specifications in TAME.* Technical Report NRL/MR/5540–12-9442. NRL, Washington, DC.

[3] M. Archer, E. Leonard, and C. Heitmeyer. 2013. Idea: Writing Secure C Programs with SecProve. In *Proc. 5th Int'l Symp. on Engg. Secure Softw. and Systems (ESSoS 2013) (LNCS)*, Vol. 7781. Springer, 171–180.

[4] Jean-Francois Bergeretti and Bernard A. Carré. 1985. Information-flow and data-flow analysis of while-programs. *ACM Trans. Pr. Lang. Syst.* 7, 1 (Jan. 1985), 37–61.

[5] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–583.

[6] R. W. Floyd. 1967. Assigning meanings to programs. In *Proc. of Symposia in Applied Math. (Math. Aspects of Computer Science)*, Vol. 19. Amer. Math. Soc., 19–32.

[7] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. 2008. Applying Formal Methods to a Certifiably Secure Software System. *IEEE Trans. Softw. Engin.* (Jan/Feb 2008).