Hybrid Information Flow Analysis for Real-World C Code

Gergö Barany, Julien Signoles

Hybrid Information Flow Analysis
for Real-World C Code[*]

Gergö Barany[1] and Julien Signoles[2]

[1] Inria Paris, France
`gergo.barany@inria.fr`
[2] CEA, LIST, Software Reliability and Security Laboratory
F-91911 Gif-sur-Yvette Cedex, France
`julien.signoles@cea.fr`

Abstract. Information flow analysis models the propagation of data through a software system and identifies unintended information leaks. There is a wide range of such analyses, tracking flows statically, dynamically, or in a hybrid way combining both static and dynamic approaches. We present a hybrid information flow analysis for a large subset of the C programming language. Extending previous work that handled a few difficult features of C, our analysis can now deal with arrays, pointers with pointer arithmetic, structures, dynamic memory allocation, complex control flow, and statically resolvable indirect function calls. The analysis is implemented as a plugin to the Frama-C framework.
We demonstrate the applicability and precision of our analyzer by applying it to an open-source cryptographic library. By combining abstract interpretation and monitoring techniques, we verify an information flow policy that proves the absence of control-flow based timing attacks against the implementations of many common cryptographic algorithms. Conversely, we demonstrate that our analysis is able to detect a known instance of this kind of vulnerability in another cryptographic primitive.

1 Introduction

Information flow analysis models the propagation of data through a software system. It identifies unintended information leaks to guarantee confidentiality of information. For instance, secret data are often forbidden from influencing public outputs [10]. This is an instance of the non-interference property [12] which states that certain kinds of computations have no effects on others. A more precise standard property is termination-insensitive non-interference (TINI), i. e., non-interference without taking into account covert channels due to termination.

Information flow analyses for ensuring TINI can be static [23,15] or dynamic [3]. The former examine the source code without executing it, while the latter check the desired properties at runtime. Dynamic monitors have neither knowledge of commands in non-executed control flow paths, nor knowledge of

---

commands ahead of their execution. Russo and Sabelfeld [21] prove that such dynamic monitors cannot be sound with respect to non-interference, while being at least as permissive as flow-sensitive type system *à la* Hunt and Sands [15]. Flow-sensitivity means that the same variable may carry data of different security levels (e.g. secret and public) over the course of the program execution. It is particularly important in practice in order to accept more programs without jeopardizing security. This leads to hybrid information flow in which the dynamic monitors are helped with statically-computed information [19,21]. This paper describes an analysis in this category.

TINI monitoring was refined by Bielova and Rezk by introducing the concept of termination-*aware* non-interference (TANI) [7]. TANI monitors are required to enforce TINI with the additional constraint that they do not introduce new termination channels. That is, no information about secret values may be derived from the fact that the monitored program terminates normally. The authors prove that hybrid monitors such as ours do enforce this property.

Flows tracked by monitors may be either explicit or implicit. Explicit flows are propagated through assignments when assigning secret data to a memory location which therefore becomes secret. Indirect flows are usually propagated through program control dependencies. For instance, when considering a sensitive variable `secret` and the code snippet `if (secret) x = 0; else y = 1;`, both variables `x` and `y` become sensitive because the fact whether `secret` is zero leaks to the values of both `x` and `y`. In order to detect such a flow at runtime when executing the `then`-branch (resp. `else`-branch), it is required to have the knowledge of the update of `y` in the non-executing `else`-branch (resp. `x` in the `then`-branch). This information is unfortunately not available at runtime: a static analysis using points-to information [22] is necessary to pre-compute it.

In this paper, we discuss hybrid flow-sensitive information flow analysis for **C programs**. In previous work, Assaf *et al.* demonstrated that indirect flows may be carried through pointers [2,1]. They proposed a hybrid analysis through a sound program transformation which encodes the information flows in an inline monitor to detect TINI (and TANI) violations. A prototype was implemented as a Frama-C plugin named Secure Flow. The soundness of the program transformation relies on a static analysis in order to compute over-approximations of written memory locations in some pieces of code. Later Secure Flow was extended to arrays [4]. One benefit of a transformation-based approach is that it lets end-users choose their verification techniques: they can verify all execution paths of the generated program by static analysis, or some individual paths by runtime verification, or even use a combination of both.

Our contributions are threefold: **extending hybrid information flow analysis** for C programs containing many complex constructs; **improving the Frama-C plugin Secure Flow** with this extension; and **evaluating it** on an open-source cryptographic library by **combining static and dynamic techniques**.

The structure of the paper is as follows. Section 2 gives an overview of Secure Flow, while Section 3 details the recently added features. Section 4 evaluates our tool on LibTomCrypt, and Section 5 describes related work.

2 An Overview of Secure Flow

This paper concerns the design and implementation of a hybrid information flow monitor in the style of Le Guernic et al. [19] for the C programming language. In this section we set the stage by describing the underlying Frama-C framework and the constraints and goals that influenced our design. To make the paper self-contained, we also discuss the handling of various language constructs of C in our previous work [2,4].

2.1 Frama-C

Our analysis is implemented as a plugin for Frama-C [17], an open source analysis and transformation framework for C programs. Frama-C parses the C source code to build an abstract syntax tree (AST) that represents the input code. After analyses and transformations, the AST can be pretty-printed to C source code that can be processed by other tools, or compiled with a C compiler and executed. Frama-C can be extended with plugins that implement new code analyzers and transformers.

To ease implementation of analyzers, Frama-C performs some normalizations of the AST: in particular, side effects can only occur due to assignments, function calls or assembly code, but not nested inside other expressions as in C. When needed, the frontend introduces assignments to temporary variables to hold the values of side effecting operations. In both our implementation and in the discussion below, we make use of the fact that side effects only occur at these well-defined places. Another normalization is relevant to our implementation: the control flow of the logical operators `&&` and `||` is made explicit by the frontend by generating appropriate `if` and `goto` statements. Finally, all C loop constructs are normalized into infinite loops of the form `while (1) { ... }` containing `if` statements controlling the loop's exits via `break`.

Frama-C supports a rich contract-based annotation language called ACSL that can express assertions, function pre- and post-conditions, loop variants and invariants, and other attributes of data types and variables [5]. ACSL annotations are formatted as special comments with a leading `@` character. Annotated programs are thus compatible with all other compilers and analyzers for C that ignore comments. Plugins may extend ACSL syntax with new kinds of annotations and new predicate symbols. Our plugin uses such custom ACSL annotations to specify initial information flow labels for variables and to express information flow policies as assertions to be verified.

Our hybrid information flow analysis needs precise information on the targets of pointers. For this we rely on Eva [8], a mature abstract interpretation based Frama-C plugin computing an over-approximation of the values of all variables in a program. Its results are accessible programmatically through its API.

2.2 Design Constraints

Our goal was to design an analysis that is as precise as possible while faithfully preserving the semantics of programs that do not violate the given information

flow policy. The latter constraint was important for choosing the representation of information flow labels in the instrumented program. A straightforward idea would be to package each monitored variable `x` of type `T` in a structure with its label, such as:

```
struct x_label { label_t label; T x };
```

However, this would change the sizes of such variables and of compound types containing such members. As a consequence, programs using C's `sizeof` or `offsetof` operator would compute different values with and without instrumentation. We therefore chose to completely separate the storage of the program's original variables and the label variables introduced by our instrumentation. Other design goals were related to precision: we track information flow through pointer-based data structures as precisely as possible. We also track information flow in structures in a field-sensitive way. For example, we want to keep separate labels for the members of a structure holding a pair of a public and a private cryptographic key.

2.3 Security Lattice and Status Annotations

At the time of writing, our analysis uses a simple two-element lattice using the values 0 (public) and 1 (private). We use the `char` type for storing the labels and the bitwise-or (|) operator as the join operator over the lattice. It would be easy to extend this scheme to support any lattice isomorphic to a lattice of bit vectors up to 64 bits. More general lattices would require a more complex combination operator.

Users can use the custom ACSL annotation /*@ private */ (specific to Secure Flow) to mark declarations of variables that have to be treated as sensitive; their label variables are initialized accordingly. All other variables are considered initially public. Custom ACSL annotations are also used to express the intended flow policy through *security annotations* at arbitrary program points e.g.,

```
/*@ assert security_status(result) == public; */
```

Similarly, functions (such as I/O functions) may be annotated with preconditions requiring their arguments to be public, e. g.

```
/*@ requires security_status(*x) == public; */
int send(char *x);
```

Such annotations are considered proof obligations to be discharged using Frama-C's static analyzers or provers, or to be turned into runtime assertions in the instrumented program. The predicate symbol `security_status` is an ACSL extension introduced by Secure Flow: its meaning is unknown to other Frama-C tools. To make the meaning explicit, our analysis translates such annotations into a reference to the corresponding label variable. The resulting predicate can then be analyzed by other Frama-C plugins as usual.

We do not predefine any flow policy: The policy is to be chosen by the user and expressed in the form of assertions. However, as an optional tool we provide a command line flag to ensure that the program's control flow always depends only on public information. The intention is to verify the absence of a certain class of timing-based attacks against cryptographic software. The same policy may be used to guard against denial-of-service attacks by users able to control the iteration counts of loops. This policy is implemented by inserting an assertion before every branching statement (`if` or `switch`) requiring the condition expression's label to be public.

2.4 Overview of Information Flow Monitoring

We briefly summarize the basics of the instrumentation done by Secure Flow. These operations follow the literature [19,2,1,4].

For every variable `x` of arithmetic type, we add a label variable $\underline{x}$ of type `char`. These are initialized to 1 (secret) for `/*@ private */` annotations, to 0 (public) otherwise. An expression's label is obtained by mapping variables to labels and replacing every operator by the combination operator `|`. Constants are public (label 0). Every assignment in the program is instrumented with a corresponding label assignment, e.g., for `x = a + b` we add `x = a | b`.

In a branching statement like `if(c) x = 0; else y = 1;` the final values of `x` and `y` depend on the path taken and thus on the condition. This is an *implicit* information flow. We model it by introducing a label variable $\underline{pc}$ for the *program counter context* in each branch or loop, initializing it from the conditional expression's label ($\underline{pc} = \underline{c}$) and using it in each label assignment controlled by the branch, e.g., $\underline{x} = 0 \mid \underline{pc}$. This handles dependencies in the branch that is actually executed, but the other branch must also be modeled. In the running example, we therefore add an update $\underline{y} \mathrel{|}= \underline{pc}$ in the true branch and the same for $\underline{x}$ in the false branch. Thus, no matter which branch is taken, all variables updated in the entire `if` statement have the implicit flow from the condition tracked correctly.

Pointers `p` that can be dereferenced n times are treated by introducing n corresponding label pointers $\underline{p\_d1}$ to $\underline{p\_dn}$ [2]. We maintain the invariant that whenever `p` points to some object `x`, its label pointer $\underline{p\_d1}$ points to $\underline{x}$, and analogously for multiple dereferences. As the number of label pointers depends on `p`'s type, pointer type casts are not supported in general. Reads/writes of `*p` are instrumented with corresponding label reads/writes of $*\underline{p\_d1}$. If `p` may refer to several targets, say `x` and `y`, the actual choice of a target at run time may depend on secret data (e.g., `if (secret) p = &x; else p = &y;`). An assignment to `*p` has an information flow from `p` to all of its targets because inspecting the targets after the assignment may allow inferences on `p` and thus the secret data. We use pointer analysis to resolve a safe overapproximation of the set of targets and insert appropriate label updates $\underline{x} \mathrel{|}= \underline{p}, \underline{y} \mathrel{|}= \underline{p}$.

Arrays introduce information flows from indices to any element the index may refer to [4]. The labels for distinct elements must therefore be shared; we use a single summary label $\underline{arr\_summary}$ to model all elements of an array `arr`.

Due to sharing, writes to the array must be modeled using monotonically non-decreasing *weak updates* of the summary label, e. g., <u>arr_summary</u> |= <u>i</u> | <u>v</u> for an assignment `arr[i] = v`. Pointers to array elements must be modeled by two label pointers: One to the array's summary label and one to the exact array element, which is needed to preserve the above pointer invariant.

3 New Features of Secure Flow

We now move on to the main contributions of this paper by presenting the way Secure Flow handles other features occurring in numerous C programs.

3.1 Structures and Unions

Our analysis treats structures in a field-sensitive way: for every `struct` type `s` defined by the program, we define a corresponding label `struct` type <u>s</u>. The members of this struct are the labels of the members of `s` computed in the usual way; that is, if `s` has a member `m` of a pointer type, <u>s</u> has the appropriate number of label pointer members <u>m_d</u>i and <u>m_d</u>i<u>_summary</u>. Arrays of structures have a corresponding summary structure.

We could, in principle, treat C's `union` types in exactly the same field-sensitive way as `struct`s. In practice, there are problems with the precision of our current implementation: mapping the results of the pointer analysis to our symbolic lvalue representation may identify too many overlapping fields and thus track too many information flows that are not present in the actual program.

3.2 Unstructured Control Flow: goto Statements

The use of the `goto` statement is widespread in systems software written in C: it is frequently used in functions that check a series of conditions and jump to cleanup code at the end of the function in case of an error. The `goto` statement may also appear in Frama-C's AST due to some normalizations: Frama-C introduces `goto` statements to model early `return`s from the middle of a function, for `continue` statements in `for` loops, and to model short-circuit evaluation of the logical `&&` and `||` operators. We must therefore be able to treat programs with `goto`s, at least in these restricted forms.

A problem with `goto` is the propagation of information flows to objects that might be modified if the branch containing the `goto` were not taken. This is similar to the label updates we insert in `if` statements for the objects modified in the other branch, but in the case of `goto` the effects are not local.

Consider the following example:

```
    x = 1;
    if (cond) { goto end; }
    x = 0;
end:
    return x;
```

At the `return` statement, the value of `x` is 0 iff the value of `cond` is 0. There is an information flow from the branch condition, via the `goto` statement, to `x`, whose assignment is skipped when the `goto` is taken. We instrument this example as follows:

```
    x = 1;
    x = 0 | pc;
    pc |= cond | pc;
    if (cond) { goto end; }
    x = 0;
    x = 0 | pc;
end:
    x |= pc;
    return x;
```

In general, we handle `goto` statements by ensuring that the program counter label captures the condition controlling the `goto` no matter which path is taken. We identify the branch controlling the `goto` and propagate its condition's label to all the program counter labels that may be traversed by the jump, including the label for the `goto`'s target. In the example, the condition controlling the `goto` is cond, and the only block possibly affected has the label pc (containing both the controlling branch and the jump target). The update pc |= cond | pc performs the propagation. Now, whether or not the jump is taken, subsequent assignments in any affected block will take place in a context including the condition's label. For the case when the `goto` *is* taken, we also insert label updates at the target statement. These update the labels of any variable whose assignments may have been skipped by the `goto`. For simplicity, we just use the set of all variables that may be modified anywhere in the target block (in the example, only x).

We handle not only `goto` statements from inner blocks to enclosing ones, but also from outer blocks into more deeply nested ones. We omit the details, but they follow the same principles as explained above. Our current implementation only allows forward `goto`s, i. e., all `goto` statements must appear textually before the corresponding target. This is just an artifact of the particular implementation strategy we chose, but there are no theoretic difficulties with treating backwards jumps the same way as forward jumps.

The analysis also handles `break` and `continue` statements in a similar way as `goto`s: It propagates the program counter label of the branching statement that controls the jump to the corresponding loop or `switch` statement.

3.3   Function Calls

Our analysis handles function calls in different ways, depending on whether the call is direct or indirect (i. e., through a function pointer) and whether the call's target has a definition in the same program or is external.

For direct calls to defined functions, we instrument both the caller and the callee. The function's parameter list is transformed by adding extra label parameters for the original parameters as well as a parameter for the program counter

label of the callee's calling context. We also add global variables for the labels of all defined functions' return values; these labels are assigned before the function returns. That is, a function to add two numbers is instrumented as follows:

```
char add_return;
float add(char local_pc, float x, char x, float y, char y) {
    float sum;
    char sum;
    sum = x + y;
    sum = x | y | local_pc;
    add_return = sum;
    return sum;
}
```

At the call site, the function call is transformed accordingly to pass in all the required labels. If the function's return value is assigned to some object, we insert corresponding assignments from the function's return label variables to the target object's labels. This is the only case we need to handle because function calls embedded in larger expressions are first transformed by Frama-C into assignments to temporary variables.

Library functions which do not have definitions in the target program must be treated separately. ACSL provides a syntax to express the side effects of such functions using (possibly several) annotations of the form

```
assigns x1, ..., xn \from y1, ..., ym
```

meaning that the function *may* only modify the lvalues `x1, ..., xn` by using *at most* the lvalues `y1, ..., ym` (but might not necessarily modify all the `xi` or use all the `yi`).

We require such annotations for all functions without visible definitions; Frama-C includes annotations for the C standard library functions. The analyzer emits a warning for external functions without annotations and continues with the (unsound) assumption that the called function has no visible side effects. If a function is defined but has an `assigns` annotation, the analyzer trusts the information from the annotation and does not use its own analysis of the function's body to model the function's externally visible effects. This improves the analyzer's efficiency while remaining safe since it is still possible to verify that the function body satisfies this annotation thanks to other tools (for instance Eva). Such annotations must be provided for recursive functions.

For a call to an external function with an annotation of

```
assigns x₁ \from y_{1,1}, ..., y_{n,1};
...
assigns x_k \from y_{k,1}, ..., y_{n,k};
```

we insert the corresponding label updates:

```
x_1 |= y_{1,1} | ...| y_{n,1} | pc;
...
```

$$\underline{x_k} \mathrel{|}= \underline{y_{k,1}} \mathrel{|} \ldots \mathrel{|} \underline{y_{n,k}} \mathrel{|} \underline{pc};$$

This approach works even for functions taking `void *` parameters, such as the standard `memcpy` function, whose `assigns` annotation expresses that it assigns bytes in its output buffer from its input buffer. At the call site we use the points-to analysis to resolve the pointer arguments to the underlying objects of concrete types and are able to generate well-typed label updates.

However, the approach does not work for functions whose `assigns` annotations include modifications to pointers. The semantics of `assigns` is that it models all assignments that *might* be performed by the called function. As we cannot be sure that these assignments will indeed take place, there is not enough information to insert the label pointer assignments needed to maintain our pointer invariants (Section 2.4). In such cases the analysis must reject the input program with an error message. For example, we allow `memcpy` on objects of the type `struct foo { int a; float b[10]; }` but not on objects of the type `struct bar { int a; float *b; }`. In the latter case, the analysis would conclude that it is not able to track all the pointer-based flows that may be performed by the function. However, the practical impact of this restriction is low: `memcpy` calls can often be rewritten as assignments if needed, and not many other standard C functions may have side effects on pointer-based structures.

For indirect function calls, we require that the points-to analysis is able to resolve the function pointer's target to a fixed set of candidate functions. The transformation generates a `switch` on the function pointer's value that dispatches to the appropriate direct function call.

Functions with variable argument lists are not handled directly. Instead we first invoke another Frama-C plugin named Variadic that transforms variable-argument functions into functions that take a fixed number of arguments. The resulting program can then be treated as usual by the information flow analysis.

Finally, functions introduce the issue of visibility of identifiers. Functions may refer to other functions' local variables via pointers, as in a call like `f(&a)`. Inside the function `f` we must be able to refer to `a`'s label variable $\underline{a}$. By default, we allocate every variable's label in the same scope as the original variable. However, for such locals that may be referenced from other functions (as determined by the points-to analysis), we make the corresponding label variables global instead.

3.4 Dynamic Memory Allocation

Our information flow analysis has special handling for the dynamic memory allocation semantics of the standard C functions `malloc`, `calloc`, and `realloc`, as well as the `free` function. These functions operate on pointers of type `void *`, which we do not allow in general. However, we do allow them in the context of dynamic allocation, as long as the type conversions (made explicit as casts by Frama-C) to or from more concrete types take place immediately at the place of the function call. Otherwise (e. g., if the program assigns the result of `malloc` to a pointer to `void` and only converts it at a later point), we reject the program.

We can thus obtain the concrete type of the allocated memory buffer. From the expression specifying the size of the allocation , and knowing the target type, we can compute the number of allocated elements. We insert calls to `calloc` to dynamically allocate the same number of labels of the appropriate types:

```
float *q = (float *) malloc(42 * sizeof(float));
char *q_d1 = (char *) calloc(42, sizeof(char));
char *q_d1_summary = ...;   // see text below
```

The information flow in dynamically allocated data structures is thus tracked via dynamically allocated labels, which allows us to track pointers with maximal precision. However, there is a problem related to label updates in branches for dynamically allocated memory areas. Consider the following program:

```
p = malloc(sizeof (int));
if (...) { *p = 1; } else { ... }
```

As before, in the `else` branch we must insert updates for the summary labels of all objects that may be referenced by `p`. This is easy if `p` may only point to variables (say, `x`): We can insert an update `x |= if_pc`. However, in the case where `p` points to dynamically allocated memory, we have no simple way of naming and enumerating the correct summary label to insert the necessary updates. We must therefore introduce an approximation. Our analysis introduces one statically allocated summary label (i. e., a global variable) per dynamic allocation site. At each such call site, label arrays are allocated dynamically, but the target's summary pointer is pointed to the call site's shared summary label. The example above is instrumented as follows:

```
p = malloc(sizeof (int));
p_d1 = calloc(1, sizeof (char));
static char dynalloc_site_1_summary = 0;
p_d1_summary = &dynalloc_site_1_summary;
if_pc = cond | global_pc;
if (cond) {
    *p = 1;
    *p_d1 = 0 | if_pc;
} else {
    dynalloc_site_1_summary |= if_pc;
    ...
}
```

This approach thus introduces aliasing between the summary labels of different buffers allocated at the same site. Raising the information flow label of one object allocated at a certain call site automatically raises the labels of any other object allocated at the same site. In the extreme, if a program contains only a single static allocation site (e. g., because it uses a wrapper function around `malloc`), *all* dynamically allocated objects share one summary. This is a source of imprecision in our current implementation.

The simplest way to resolve this issue is to turn an allocation site that may be used in different contexts into explicitly different allocation sites. This could be done by automatic inlining of functions performing dynamic allocation. In our experiments with cryptographic software, we manually duplicate an allocation function, introducing a special variant for the allocation of secret keys.

3.5 Summary of Restrictions

We briefly summarize the restrictions on input programs that can be analyzed by Secure Flow.

- Most kinds of type casts between pointer types are forbidden; casts to and from `void *` related to dynamic memory allocation are allowed, as are casts between `void *` and character pointer types.
- The program must contain `assigns` annotations for all external functions (provided by Frama-C for the C standard library) and recursive functions.
- Calls to external functions may not have side effects on pointers, but may have side effects on their targets.
- No backwards jumps with `goto` are allowed (this is only an artifact of the current implementation).
- The analysis is imprecise (i. e., overly conservative) if logically separate memory areas are allocated at the same call site of `malloc`. This can be avoided by inlining/specializing functions that perform dynamic allocation.

Overall, these conditions do not impose disproportional restrictions on well-written systems code, as long as the entire program is available for whole-program analysis, or annotated with `assigns` annotations for the missing parts.

4 Evaluation

We evaluate our hybrid information flow analysis by checking an information flow policy to protect against timing attacks on a cryptographic library. This verification also illustrates one of the main benefits of our hybrid approach: combining static and dynamic verification.

4.1 Background on the Chosen Flow Policy

Timing attacks and other side-channel attacks are an important class of vulnerability in the implementations of cryptosystems; a recent article by Genkin et al. gives a good overview of a wide range of techniques and targets [11]. The class of timing attack that interests us is caused by conditional branching on data derived from the cryptosystem's secret key. This type of vulnerability typically occurs in asymmetric (also known as public-key) cryptosystems such as RSA, where the core of the algorithm loops over the bits of the secret key and decides based on the value of each bit which mathematical operation to perform.

In general, the two branches of an `if` statement take different amounts of time, and an attacker who is able to measure a cryptographic operation's execution time may use this to deduce information about the secret key. Such attacks are known against implementations of several cryptosystems in common use, including both RSA and elliptic curve cryptography [11,6]. Even if timing differences cannot be measured directly, attackers on the same machine may be able to observe instruction cache misses that allow them to deduce the same kind of information [20].

We therefore chose a flow policy forbidding control flow based on secret information; this is a useful property to verify on real-world cryptographic code. We note in passing that there are also timing attacks based on the order of accesses to lookup tables and the corresponding data cache misses. These are independent of control flow and outside of the scope of our current analysis.

In the following sections we discuss our analysis of the cryptographic implementations in the LibTomCrypt library (`http://www.libtom.org/`).

4.2 Symmetric Cryptosystems

LibTomCrypt includes implementations of 14 different symmetric cryptosystems. This class of system typically works by breaking the input into fixed-sized blocks, then performing permutations and substitutions of the bytes in each block based on look-up tables indexed by the key and a loop counter. The number of operations per block is fixed by the algorithm, and the number of blocks to be processed depends only on the length of the message (which we do not consider secret information). Thus we did not expect to find timing attacks against this class of system. As these programs are safe with respect to our flow policy, they are a good test to ensure that our analysis does not introduce imprecisions.

We use a separate driver program for each of the cryptosystems. The driver calls LibTomCrypt's initialization routines for the given system, then encrypts 10 megabytes of random data, decrypts the encrypted data, and quits. We perform whole-program analysis, applying Frama-C to each driver program and the entire LibTomCrypt source code. However, the (fixed) key and data to be encrypted are not exposed to Frama-C to avoid the possibility that Eva specializes its results to a given key or input.

Our analyzer was able to instrument and analyze all of the programs with only one significant change to LibTomCrypt: the internal states of all the different systems are stored in a union, only one member of which is used at a time. As discussed in Section 3.1, our analysis is currently unable to analyze accesses to unions precisely. We therefore changed the type of this union to `struct`. After this change, we can successfully instrument each of the 14 different symmetric cryptosystems. On the instrumented program, Eva successfully statically verifies for each of the systems that the flow policy is satisfied, i. e., no branch condition in the program depends on the key.

We next turned to dynamic analysis of the instrumented programs to evaluate instrumentation overhead. We compiled both the original driver program and the program instrumented by our hybrid information flow analysis with GCC

Table 1. Execution time of symmetric cryptographic algorithms with and without instrumentation and with additional E-ACSL instrumentation.

Program	Original	Instrumented	Slowdown	+E-ACSL	+Slowdown
aes	0.11 s	0.33 s	3.0 ×	6.87 s	20.8 ×
anubis	0.13 s	0.36 s	2.8 ×	6.84 s	19.0 ×
blowfish	0.19 s	0.44 s	2.3 ×	6.99 s	15.9 ×
cast5	0.26 s	0.50 s	1.9 ×	8.46 s	16.9 ×
kasumi	0.42 s	0.76 s	1.8 ×	11.24 s	14.8 ×
khazad	0.15 s	0.33 s	2.2 ×	7.26 s	22.0 ×
kseed	0.30 s	0.53 s	1.8 ×	7.60 s	14.3 ×
noekeon	0.22 s	0.41 s	1.9 ×	6.69 s	16.3 ×
rc2	0.42 s	0.61 s	1.5 ×	7.42 s	12.2 ×
rc5	0.15 s	0.35 s	2.3 ×	7.76 s	22.2 ×
rc6	0.18 s	0.36 s	2.0 ×	7.41 s	20.6 ×
saferp	0.28 s	1.97 s	7.0 ×	27.66 s	14.0 ×
twofish	0.15 s	0.36 s	2.4 ×	6.97 s	19.4 ×
xtea	0.30 s	0.47 s	1.6 ×	7.04 s	15.0 ×

version 4.8.4 at optimization level `-O2` and ran them on an Ubuntu Linux system on an 8-core Intel Core i7 CPU clocked at 2.30 GHz. Table 1 shows the execution times of the programs in seconds. We ran each program five times and report the median of the runs. We report execution times of the original program and the instrumented version in seconds as well as the slowdown due to instrumentation.

Slowdowns for most of the programs are below or near a factor of 2. This is the order of magnitude we expected, inserting one or more assignment statements for every assignment in the program. We have not yet been able to determine the reason for the large slowdown factor of 7.0 for the one outlier, `saferp`.

The last two columns in Table 1 show the additional instrumentation overhead when using Frama-C's E-ACSL plugin [18]. Indeed Eva proves all security assertions, but must leave some memory safety properties unproved due to approximations made during the analysis. E-ACSL instruments the program to dynamically monitor accesses to memory blocks to ensure memory safety at runtime. This is needed because the correctness of the information flow analysis is only guaranteed if the program is memory safe. We report the absolute execution time in seconds for each benchmark instrumented with information flow tracking and memory safety monitoring. The last column shows a slowdown of 10 to 20 which is typical of this monitoring for E-ACSL version 0.8 [24].

The table does not show the memory use of the programs, which behaves fairly regularly: All of the original programs use about 32 MB as they are linked statically to the same library containing the lookup tables for all the different algorithms, and to the same buffer of random input data. These programs do not use dynamic memory allocation. For the instrumented programs memory use increases to about 52 MB, a factor of 1.6. This, too, is as expected: These algorithms perform byte-oriented processing, and we currently monitor each byte in the various byte arrays and lookup tables with a 1-byte label of type `char`. That is, we essentially double the memory used by monitored data. The relative

overhead would be smaller for programs that mainly use larger types than `char`. With E-ACSL, memory use grows to about 190 MB, an additional factor of 3.7.

This combination of Eva and E-ACSL demonstrates the ability of our hybrid approach to combine static and dynamic analyses. Here Eva proves the security assertions, while E-ACSL validates at runtime the remaining safety properties. Both together ensure that our programs are safe with respect to our flow policy.

4.3 Elliptic Curve Cryptography

As a representative of the class of public-key (asymmetric) cryptosystems, we chose to analyze LibTomCrypt's implementation of elliptic curve cryptography (ECC). The library offers two implementations of the underlying algorithm: one of the implementations is known to be faster but vulnerable to timing attacks, while the other is claimed to be resistant to timing attacks. The vulnerable operation is the multiplication of a point on an elliptic curve with a scalar in the function `ltc_ecc_mulmod`. The scalar `k` is part of the system's private key. In a loop, the function inspects `k`'s bits one by one and, depending on its value, performs some arithmetic or continues to the next loop iteration.

Elliptic curve operations must be done on multiple-precision integers. LibTom-Crypt is able to use one of several multiple-precision arithmetic libraries; we chose its sister project LibTomMath. Again, we perform whole-program analysis using Frama-C on a driver program, the LibTomCrypt library, and LibTomMath. Some rewrites were needed to make it possible to analyze this system with our information flow analyzer. The biggest change concerned LibTomCrypt's interface to its math library, which needlessly uses `void *` pointers throughout to refer to multiple-precision integers. As discussed in Section 2.4, our analysis cannot deal with such programs. We manually changed the types in this internal interface to the concrete type of LibTomMath's integers.

For the information flow analysis we also had to be able to label the private ECC key as secret. In LibTomCrypt all integers are allocated by the `mp_init` function, which calls `malloc`. Due to the context sensitivity problem described in Section 3.4, all integers share the same dynamic allocation summary label, thus marking the secret key as secret would trivially make every number secret. To avoid this we duplicated `mp_init` as `mp_init_secret` and used this variant to allocate the numbers in the secret key. As the `malloc` call site is no longer shared, this version no longer suffers from the false sharing issue.

We also had to change a few places where backwards jumps with `goto` statements were used to implement loops. In all cases we were able to rewrite the code to avoid using a `goto`. We added ACSL annotations of the form

```
assigns result->dp[..], result->sign \from
    a->dp[..], a->sign, b->dp[..], b->sign[..]
```

to some of the basic mathematical functions in LibTomMath (addition, subtraction, multiplication) to speed up Eva. These annotations express that the sign and digits (`dp`) of the result of such an operation depend only on the sign and

digits of the operands. Finally, we simplified code for parsing serialized ECC keys. The code was in principle analyzable, but its complexity caused unnecessary slowdowns in Eva, hindering our experiments.

After these changes, we were able to apply our program transformation. Subsequent static analysis using Eva has shown that the flow policy is indeed violated: The secret label is propagated from the secret key to a conditional branch as discussed above. We can thus confirm the known timing vulnerability.

However, our analysis also found essentially the same bug in the variant of the `ltc_ecc_mulmod` function that is claimed to be resistant to timing attacks by performing the same amount of work on different branches that depend on secret information. We give a simplified description of this implementation. Depending on a variable `i` (the next bit of the key, 0 or 1) and another 0-1 variable `mode` derived from `i`, the code performs a conditional branch and executes either a call `ecc_ptdbl(M[i], M[i])`, doubling some value `M[i]` and storing it back into `M[i]`, or `ecc_ptdbl(M[1], M[2])`, doubling `M[1]` and ignoring the result by storing it into a dummy location `M[2]`. This is in violation of our flow policy, as the conditional branch on secret data remains in the program. We believe that this implementation may give a false sense of security in the light of cache-based attacks, and that it should be replaced by a version that does not suffer from this problem [6, Section 3].

A possible solution is to replace the branch on `i` and `mode` by a lookup table to determine the arguments for the function call. The following variant of the computation sketched above is correctly accepted by our static analysis without raising an alarm about secret-dependent control flow:

```
int i1_tbl[2][2] = {{1,1},{i,i}}, i2_tbl[2][2] = {{2,2},{i,i}};
ecc_ptdbl(M[i1_tbl[mode][i]], M[i2_tbl[mode][i]]);
```

For dynamic analysis, we use a test program that performs 3000 repetitions of the basic ECC operation `ecc_shared_secret`. The original program executes in 0.73 s (median of five runs), using 1400 kB of memory. We run the dynamic analysis with instrumentation to propagate labels, but without aborting the program on policy violations (which would happen instantaneously). With instrumentation, execution time increases to 4.71 s ($6.5 \times$) and memory use to 1553 kB ($1.1 \times$). The current version of E-ACSL does not scale well to programs of several hundreds of thousands of lines of code, so we did not run it on this benchmark.

5 Related Work

As mentioned before, this paper is based on work by Assaf et al. [2,1] and Barany [4] which focused on sound hybrid information flow analysis of a subset of C including pointers, arrays and pointer arithmetics. We extend this work to a larger subset of C including structures, unstructured control flow, indirect function calls and dynamic memory allocation.

There is an abundant literature on information flow analysis. However, as far as we know, there is no hybrid information flow analysis that handles a subset

of the C programming language as large as ours. To our knowledge, the only approaches that handle real-world programs target JavaScript [16,13].

First, Kerschbaumer et al [16] handle JavaScript's arrays, structures, unstructured control flow and function calls. However, the details are omitted except for unstructured control flow. There is no mention of indirect function calls. Second, Hedin et al develop JSFlow [14] with its extended hybrid version [13]. They track arrays, structures, function calls and dynamic allocations precisely. They also support JavaScript's closures which are usually encoded via function pointers in C. However, there is no mention of unstructured control flow. Also they target a less permissive notion of non-interference than ours: they do not allow to assign secret values to locations that previously held public values (the converse, overwriting a secret value by a public value, is allowed). In contrast, our approach only uses such constraints at user-defined program points through security annotations. However, we could encode their non-interference property by adding an assertion to every assignment statement.

Regarding the C programming language, previous work which aims at targeting a large variety of C programs is based on taint analysis, either statically [9] or dynamically [25]. However taint analysis is not appropriate for verifying non-interference properties similar to ours because it does not detect all kinds of indirect flow. The aforementioned approaches are no exception.

6 Conclusions

We have presented Secure Flow, a hybrid information flow analysis for real-world C programs. Secure Flow instruments C code with monitoring code after an auxiliary static analysis. The instrumented code tracks information flow labels for all values of interest, as determined by a flexible annotation system for expressing information flow policies. After instrumentation, the code may be analyzed statically or may be executed for dynamic monitoring. Our experiments show that the overhead of dynamic monitoring is reasonable.

Secure Flow is implemented as a plugin for the Frama-C platform. It is about 3500 lines of OCaml code, blank lines excluded. It supports a large subset of C, including important real-world features such as pointers with pointer arithmetic, dynamic allocation, `goto` statements, and function pointers. Future work includes removing current restrictions, including at least backwards jumps and several memory allocations from the same call site.

We have demonstrated Secure Flow, in combination with the static analysis tool Eva and the monitoring tool E-ACSL, on a real-world cryptographic library. Our experiments show that they can verify the absence of an important class of timing attacks in many cryptosystem implementations. It has also found a known timing vulnerability in another cryptosystem, but also a similar issue in the alternative implementation supposedly correcting the vulnerability.

References

1. Assaf, M.: From Qualitative to Quantitative Program Analysis: Permissive Enforcement of Secure Information Flow. Ph.D. thesis, Université de Rennes 1 (2015), https://hal.inria.fr/tel-01184857

2. Assaf, M., Signoles, J., Tronel, F., Totel, É.: Program transformation for non-interference verification on programs with pointers. In: Janczewski, L.J., Wolfe, H.B., Shenoi, S. (eds.) Security and Privacy Protection in Information Processing Systems, IFIP Advances in Information and Communication Technology, vol. 405, pp. 231–244. Springer (2013), http://dx.doi.org/10.1007/978-3-642-39218-4_18

3. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security. pp. 113–124. PLAS '09, ACM (2009), http://doi.acm.org/10.1145/1554339.1554353

4. Barany, G.: Hybrid information flow analysis for programs with arrays. In: Hamilton, G., Lisitsa, A., Nemytykh, A.P. (eds.) Proceedings of the Fourth International Workshop on Verification and Program Transformation, Eindhoven, The Netherlands, 2nd April 2016. Electronic Proceedings in Theoretical Computer Science, vol. 216, pp. 5–23. Open Publishing Association (2016)

5. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/acsl.html

6. Bernstein, D.J., Lange, T.: Failures in NIST's ECC standards (2016), https://cr.yp.to/newelliptic/nistecc-20160106.pdf

7. Bielova, N., Rezk, T.: A taxonomy of information flow monitors. In: Piessens, F., Viganò, L. (eds.) Principles of Security and Trust: 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings. pp. 46–67. Springer Berlin Heidelberg (2016)

8. Blazy, S., Bühler, D., Yakobowski, B.: Structuring Abstract Interpreters through State and Value Abstractions. In: International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'17) (Jan 2017)

9. Ceara, D., Mounier, L., Potet, M.L.: Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences. In: the 3rd International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2010). pp. 371–380 (2010)

10. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM 20(7), 504–513 (Jul 1977), http://doi.acm.org/10.1145/359636.359712

11. Genkin, D., Packmanov, L., Pipman, I., Shamir, A., Tromer, E.: Physical key extraction attacks on PCs. Communications of the ACM 59(6), 70–79 (2016), http://cacm.acm.org/magazines/2016/6/202646-physical-key-extraction-attacks-on-pcs/fulltext

12. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Security and Privacy, 1982 IEEE Symposium on (April 1982)

13. Hedin, D., Bello, L., Sabelfeld, A.: Value-sensitive hybrid information flow control for a javascript-like language. In: Proceedings of the 2015 IEEE 28th Computer Security Foundations Symposium. pp. 351–365. CSF '15, IEEE Computer Society (2015)

14. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: Jsflow: Tracking information flow in javascript and its apis. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. pp. 1663–1671. SAC '14, ACM (2014)

15. Hunt, S., Sands, D.: On flow-sensitive security types. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 79–90. POPL '06, ACM (2006), http://doi.acm.org/10.1145/1111037.1111045

16. Kerschbaumer, C., Hennigan, E., Larsen, P., Brunthaler, S., Franz, M.: Information flow tracking meets just-in-time compilation. ACM Trans. Archit. Code Optim. 10(4), 38:1–38:25 (Dec 2013)

17. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing 27(3), 573–609 (2015), http://dx.doi.org/10.1007/s00165-014-0326-7

18. Kosmatov, N., Signoles, J.: A lesson on runtime assertion checking with Frama-C. In: International Conference on Runtime Verification (RV 2013). LNCS, vol. 8174, pp. 386–399. Springer (Sep 2013)

19. Le Guernic, G., Banerjee, A., Jensen, T., Schmidt, D.A.: Automata-based confidentiality monitoring. In: Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues. ASIAN'06, Springer-Verlag (2007), http://dl.acm.org/citation.cfm?id=1782734.1782741

20. Percival, C.: Cache missing for fun and profit (2005), http://www.daemonology.net/papers/cachemissing.pdf

21. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: Computer Security Foundations Symposium (CSF), 2010 23rd IEEE. pp. 186–199 (July 2010)

22. Smaragdakis, Y., Balatsouras, G.: Pointer analysis. Foundations and Trends in Programming Languages 2(1), 1–69 (2015), https://yanniss.github.io/points-to-tutorial15.pdf

23. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. 4(2-3), 167–187 (Jan 1996), http://dl.acm.org/citation.cfm?id=353629.353648

24. Vorobyov, K., Signoles, J., Kosmatov, N.: Shadow State Encoding for Efficient Monitoring of Block-level Properties Submitted for publication

25. Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In: Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15. USENIX-SS'06, USENIX Association (2006)