Uncovering Information Flow Policy Violations in C Programs (Extended Abstract)

Darion Cassel[1(✉)], Yan Huang[2], and Limin Jia[1]

[1] Carnegie Mellon University, Pittsburgh, PA 15213, USA
{darioncassel,liminjia}@cmu.edu
[2] Indiana University, Bloomington, IN 47405, USA
yh33@indiana.edu

Abstract. Programmers of cryptographic applications written in C need to avoid common mistakes such as sending private data over public channels or improperly ordering protocol steps. These secrecy, integrity, and sequencing policies can be cumbersome to check with existing general-purpose tools. We have developed a novel means of specifying and uncovering violations of these policies that allows for a much lighter-weight approach than previous tools. We embed the policy annotations in C's type system via a source-to-source translation and leverage existing C compilers to check for policy violations, achieving high performance and scalability. We show through case studies of recent cryptographic libraries and applications that our work is able to express detailed policies for large bodies of C code and can find subtle policy violations. We show formal connections between the policy annotations and an information flow type system and prove a noninterference guarantee of our design.

Keywords: Information flow · Type systems · Security

1 Introduction

Programs often have complex data invariants and API usage policies written in their documentation or comments. The ability to detect violations of these invariants and policies is key to the correctness and security of programs. This is particularly important for cryptographic protocols and libraries, which the security of large systems depend on. There has been much interest in checking implementations of cryptographic protocols [3,12–15,25,28,41]. These verification systems, while comprehensive in their scope, require expert knowledge of both the cryptographic protocols and the verification tool to be used effectively.

What remains missing is a lightweight, developer-friendly, compile-time tool to help programmers identify errors that violate high-level policies on C programs. Particularly important policies are secrecy (e.g., sensitive data is not

K. Sako et al. (Eds.): ESORICS 2019, LNCS 11736, pp. 26–46, 2019.
https://doi.org/10.1007/978-3-030-29962-0_2

given to untrusted functions), integrity (e.g., trusted data is not modified by untrusted functions), and API call sequencing (e.g., the ordering of cryptographic protocol steps is maintained), all of which are information flow policies.

In this paper, we present a framework called `FlowNotation` where C programmers can add lightweight annotations to their programs to express policy specifications. These policies are then automatically checked by C compiler, potentially revealing policy violations in the implementation. Our annotations are in the same family as *type qualifiers* (e.g. CQual [22,37,57]), where qualifiers such as *tainted* and *trusted* are used to identify violations of integrity properties of C programs; supplying tainted inputs to a function that requires a trusted argument will cause a type error. Our work extends previous results to support more complex and refined sequencing properties. Consider the following policy: a data object is initially tainted, then it is sanitized using an `encodeURI` API, then serialized using a `serialize` API, and finally written to disk using a `fileWrite` API. Such API sequencing patterns are quite common, but cannot be straightforwardly captured using previous type qualifier systems. `FlowNotation` extends type qualifiers to include a sequence of labels for specifying such policies. However, rather than implementing a new type system, `FlowNotation` translates an annotated C program to another C program, which is then checked by a C compiler for policy violations. The key insight is that qualified C types can be translated to C structures whose fields are the original C types. Consequently, we leverage performant C type checkers for checking large codebases.

To gain a formal understanding of the type of errors that we can uncover with this system, we model the annotated types as *information flow types*, which augment ordinary types with security labels. We define a core language *polC* and prove that its information flow type system enforces noninterference. The novelty of *polC*'s type system is that the security labels are sequences of secrecy and integrity labels, specifying the path under which data can be relabeled. Relabeling corresponds to *declassification* (marking secrets as public) and *endorsement* (marking data from untrusted source as trusted). The type system ensures that relabeling functions are called in the correct order. We also define μC, a core imperative language with nominal types but without information flow labels in order to model a fragment of C. We then formally define our translation algorithm based on *polC* and μC and prove the algorithm correct. The formalism not only makes explicit assumptions made by our algorithm, but also provides a formal account of the properties being checked by the annotations.

To demonstrate the effectiveness of `FlowNotation`, we implement a prototype for a subset of C and evaluate the prototype on several cryptographic libraries. Our evaluation shows that we are able to check useful information flow policies in a modular way and uncover subtle program flaws. Our full paper can be found at [21] and the source code of `FlowNotation` can be downloaded from the following URL: https://github.com/flownotation.

2 Overview and Motivating Examples

System Overview. The left side of Fig. 1 illustrates how `FlowNotation` works. First, a programmer annotates policies. `FlowNotation` takes the annotated program and produces a translated C program, which is then type-checked using an off-the-shelf C compiler. A type error implies a policy violation. Once free of violations, the unannotated, pre-translation program is used.

Checking Secrecy Policies. Suppose developers are working on a C project that uses customers' financial data. This project integrates a secure two-party computation component that allows Alice and Bob to find out which of the two is wealthier without revealing their wealth to the other or relying on a trusted third party. Let us assume that the program obtains Alice's balance using the function `get_alice_balance`, then calls function `wealthierA` to see whether Alice is wealthier than Bob. `wealthierA`'s implementation uses a library that provides APIs for secure computation primitives.

```
int bankHandler() { int balA; balA = get_alice_balance(); ...
    wealthierA(balA); }
```

The variable `balA` contains Alice's balance, so it should be handled with care; in particular, the secrecy of `balA` is maintained. One way is to use information flow types (e.g. [51]) to assign `balA` the type (`int AlicePriv`), indicating that it is an integer containing an `AlicePriv` type of secret. In contrast, the type (`int Public`) can be given to variables that do not contain secrets. The type system then makes sure that read and write operations involving `balA` are consistent with its secrecy label. For instance, if a function `postBalance`(`int Public`), which is meant to post the balance publicly, is called with `balA` as the argument, the type system will reject this program for violating the secrecy policy.

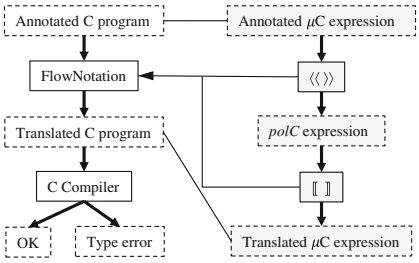

Fig. 1. Overview of `FlowNotation`.

`FlowNotation`'s annotations are *information flow labels*, each of which has a *secrecy* component and an *integrity* component. Programmers can provide these annotations above the declaration of `balA` to specify the secrecy policy as follows.

```
#requires AlicePriv:secrecy
int balA;
```

Here, `#requires` is a directive to help parse this annotation (in practice, `#pragma` prefaces it). `AlicePriv` is a secrecy label. `secrecy` indicates only the secrecy component is of concern and `balA`'s integrity component is automatically assigned `bot`, the lowest integrity. This annotation can be used to check P_1 below:

P_1: `balA` should never be given as input to an untrusted function.

The corresponding information flow type of `balA` is `int(AlicePriv, bot)`. Trusted functions are those trusted by the programmer not to leak `balA`. Let us assume our programmer trusts the following APIs. The API `encodeA` converts

an integer argument into a bit representation similar to what is used in Obliv-C [54] for use with a garbled circuit. The API `yao_execA` takes a pointer to a function `f` and an argument for `f`, and runs `f` as a circuit with Yao's protocol [52]. API `reveal` gives the result to both parties. The following code is for Alice (Bob's program is symmetric, which we omit):

```
int compare(int a, int b) { return a > b; }
int wealthierA(int balA) {
    int res = yao_execA(&compare, encodeA(balA));
    reveal(&res, ALICE); }
```

This program first encodes Alice's balance, and then calls `yao_execA` with the comparison function and Alice's encoded balance as arguments, and finally calls `reveal`. The code as it stands will not type-check after being translated, unless the programmer annotates the secure computation APIs.

```
#param AlicePriv:secrecy
int encodeA(int balA);
#param(2) AlicePriv:secrecy
int yao_execA(void* compare, int balA);
```

These two annotations state that the functions must accept parameters with the label `AlicePriv`. In the second annotation, `#param(2)` specifies that the annotation should only apply to the second parameter. A violation of P_1 will be detected, when `balA` is given to a function that does not have this kind of annotation; e.g. that is not allowed to accept `AlicePriv`-labeled data.

Checking Integrity and Sequencing Policies. `FlowNotation` can also be used to check for violations of the following, more refined, policy:

P_2 : `balA` should be used by the encoding function and then by the Yao protocol execution.

The annotation for `balA` is:

```
#requires AlicePriv:secrecy then EncodedBal:integrity
int balA;
```

The keyword `then` allows for the sequencing of labels. Corresponding changes are made to the other annotations:

```
#param AlicePriv:secrecy
#return EncodedBal:integrity
int encodeA(int balA);
#param(2) EncodedBal:integrity
int yao_execA(void* compare, int balA);
```

The `encodeA` function, as before, requires the argument to have the `AlicePriv` secrecy label. In addition, the return value from `encodeA` will have the integrity label `EncodedBal`, stating that it is endorsed by the `encodeA` function to be properly encoded. The `yao_execA` function requires the argument to have the same integrity label. If only `encodeA` is annotated with `EncodedBal` at its return value, the type system will check that an appropriate API call sequence (`encodeA` followed by `yao_execA`) is applied to the value stored in `balA`.

3 A Core Calculus for Staged Release

We formally define *polC*, which models annotated C programs that `FlowNotation` takes as input. We show that *polC*'s type system can enforce not only secrecy and integrity policies, but also staged information release and data endorsement policies. We prove that our type system enforces noninterference, from which the property of staged information release is a corollary.

3.1 Syntax and Operational Semantics

The syntax of *polC* is summarized in Fig. 2. We write ℓ to denote security labels, which consist of a secrecy tag s and an integrity tag ι. We assume there is a security lattice $(S, \sqsubseteq_S)$ for secrecy tags and a security lattice $(I, \sqsubseteq_I)$ for integrity tags. The security lattice $\mathcal{L} = (L, \sqsubseteq)$ is the product of the above two lattices. The top element of the lattice is $(\top_S, \bot_I)$ (abbreviated $\top$), denoting data that do not contain any secret and come from the most trusted source; and the bottom element is $(\bot_S, \top_I)$ (abbreviated $\bot$), denoting data that contain the most secretive information and come from the least trusted source.

A policy, denoted ρ is a sequence of labels specifying the precise sequence of relabeling (declassification and endorsement) of the data. The example from Sect. 2 uses the following policy: $(\mathit{AlicePrivate}, \bot_I)::(\bot_S, \mathit{EncodedBal})::\bot$. A policy always ends with either the top element, indicating no further relabeling is allowed, or the bottom element, indicating arbitrary relabeling is allowed. For our application domain, the labels are distinct points in the lattice that are not connected by any partial order relations except $\top$ and $\bot$.

$$
\begin{array}{lll}
\textit{Labels} & \ell & ::= (s, \iota) \\
\textit{Policies} & \rho & ::= \bot \mid \top \mid \ell :: \rho \\
\textit{1st order types} & b & ::= \mathsf{int} \mid \mathsf{ptr}(s) \mid T \\
\textit{Simple sec. types} & t & ::= b\ \rho \mid \mathsf{unit} \\
\textit{Security types} & s & ::= t \mid [pc](t \to t)^{\rho} \\
\textit{Values} & v & ::= x \mid n \mid () \mid f \mid T\{v_1, \cdots, v_k\} \mid \mathit{loc} \\
\textit{Expressions} & e & ::= v \mid e_1 \,\mathsf{bop}\, e_2 \mid v\, e \mid \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \\
 & & \quad \mid v.i \mid \mathsf{if}\, v_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \mid v := e \\
 & & \quad \mid \mathsf{new}\, e \mid *v \mid \mathsf{reLab}(\ell' ::\bot \leftarrow \ell ::\top)\, v
\end{array}
$$

Fig. 2. Syntax of *polC*

A simple (first-order) security type, denoted t, is obtained by adding policies to ordinary types. Our core language supports integers, unit, pointers, and record types ($\mathsf{struct}\ T\ \{t_1, \cdots, t_k\}$ to model C structs). Here T is the defined name for a record type. To simplify our formalism, we assume that defined type T is always a record type named T. Our information flow types use the policy ρ, rather than a single label ℓ. The meaning of an expression of type $\mathsf{int}\ \rho$ is that this expression is evaluated to an integer and it induces a sequence of declassification (endorsement) operations according to the sequence of labels specified by ρ. For instance, $e : \mathsf{int}\ H::L::\bot$ means that e initially is of $\mathsf{int}\ H$, then it can be given to a declassification function to be downgraded to $\mathsf{int}\ L$, the resulting expression can be further downgraded to bottom. $e : \mathsf{int}\ H::L::\top$ is similar except that the

last expression cannot be declassified further; i.e.. it stays at L security level. The annotated type for `balA` in Sect. 2 can be similarly interpreted.

The unit type is inhabited by one element (), so it does not need a label. A function type is of the form $[pc](t_1 \to t_2)^{\rho}$, where t_1 is the argument's type, t_2 is the return type, ρ is the security label of the function indicating who can receive this function, and pc, called the program counter, is the security label representing where this function can be called. A function f of type $[L::\bot](t_1 \to t_2)^{H::\bot}$ cannot be called in an if branch that branches on a value with label $H::\bot$ and the function itself cannot be given to an attacker whose label is $L::\bot$.

Our expressions are reminiscent of A-normal forms (ANF): all elimination forms use only values (e.g., $v.i$, instead of $e.i$). This not only simplifies our proofs, but also the translation rules (presented in Sect. 4). The fragment of C that is checked in our case studies is quite similar to this form. Values can be variables, integers, unit, functions, records, and store locations. Since we are modeling an imperative language, all functions are predefined, and stored in the context Ψ. Expressions include function calls, if statements, let bindings, and store operations. One special expression is the relabeling (declassification) operation, written $\mathsf{reLab}(\ell'::\bot \leftarrow \ell::\top)\ v$. This operation changes the label of v from $\ell::\top$ to $\ell'::\bot$. Such an expression should only appear in trusted declassification functions. For our applications, we further restrict the relabeling to be between two labels; from one ending with the top element to one ending with bottom element. We will explain this later when we explain the typing rules.

polC's small step semantic rules are denoted $\Psi \vdash \sigma \ / \ e \longrightarrow \sigma' \ / \ e'$, where Ψ stores all the functions, σ maps locations to values and e is the expression to be evaluated. The rules are standard and can be found in our full paper.

3.2 Typing Rules

The type system makes use of several typing contexts. We write D to denote the context for all the type definitions. We only consider type definitions of record (struct) types, written $T \mapsto \mathsf{struct}\ T\ \{t_1, \cdots, t_k\}$. The typing context for functions is denoted F. We distinguish two types of functions: ordinary functions, and declassification/endorsement functions whose bodies are allowed to contain relabeling operations, written $f{:}(\mathsf{d\&e})[pc]t_1 \to t_2$. F does not dictate the label of a function f. Instead, the context in which f is used decides f's label.

$$
\begin{array}{ll}
\textit{Type def. ctx} & D ::= \cdot \mid D, T \mapsto \mathsf{struct}\ T\ \{t_1, \cdots, t_k\} \\
\textit{Func typing ctx} & F ::= \cdot \mid F, f{:}[pc]t_1 \to t_2 \mid F, f{:}(\mathsf{d\&e})[pc]t_1 \to t_2 \\
\textit{Store Typing} & \Sigma ::= \cdot \mid \Sigma, loc : s
\end{array}
$$

We write Σ to denote the typing context for pointers. It maps a pointer to the type of its content. Γ is the typing context for variables, and pc is the security label representing the program counter. Typing judgment: $D; F; \Sigma; \Gamma \vdash v : t$ types values, and $D; F; \Sigma; \Gamma; pc \vdash e : t$ types expressions. The typing rules for *polC* are summarized in the full paper.

Most of these typing rules are standard. These rules carefully arrange the constraints on policies and the program counter so that the noninterference theorem can be proven. We explain the rule P-T-E-DE, which types the application of a declassification/endorsement function and is unique to our system.

$$\frac{D; F; \Sigma; \Gamma \vdash v_f : (\mathsf{d\&e})[pc'](b\ \ell_1{::}\top \to b\ \ell_2{::}\bot)^{\rho_f} \quad D; F; \Sigma; \Gamma; pc \vdash e_a : b\ \rho \qquad \rho = \ell_1{::}\ell_2{::}\rho' \qquad \rho_f \sqcup pc \sqsubseteq pc'}{D; F; \Sigma; \Gamma; pc \vdash v_f\ e_a : b\ \ell_2{::}\rho'}\ \text{P-T-E-DE}$$

We first define when a policy ρ_1 is less strict than another, ρ_2, written $\rho_1 \sqsubseteq \rho_2$, as the point-wise lifting of the label operation $\ell_1 \sqsubseteq \ell_2$. When one policy reaches its end, we use $\bot \sqsubseteq \rho$ or $\rho \sqsubseteq \top$. $\bot$ represents a policy that can be arbitrarily reclassified and thus is a subtype of any policy ρ. $\top$ is the strictest policy that forbids any reclassification; so any policy is less strict than $\top$.

The first premise checks that v_f relabels data from ℓ_1 to ℓ_2. The second premise checks that e_a's type matches that of the argument of v_f; further, e_a's policy ρ has ℓ_1 and ℓ_2 as the first two labels, indicating that e_a is currently at security level ℓ_1 and the result of processing e_a has label ℓ_2. Finally, the return type of the function application has the tail of the policy ρ. The policy of e_a does not change; instead, the policy of the result of the relabeling function inherits the tail of e_a's policy. Therefore, our type system is not enforcing type states of variables as found in the Typestate system [48]. These declassification and endorsement functions only rewrite one label, not a sequence of labels. This allows us to have finer-grained control over the stages of relabeling.

$$\frac{D; F; \Sigma; \Gamma \vdash v : b\ \rho \qquad pc \sqsubseteq \rho'}{D; F; \Sigma; \Gamma; pc \vdash \mathsf{reLab}(\rho' \Leftarrow \rho)\ v : b\ \rho'}\ \text{P-T-E-Relabel}$$

$$\frac{D; F; \Sigma; \Gamma; pc \vdash e : s' \qquad s' \leq s}{D; F; \Sigma; \Gamma; pc \vdash e : s}\ \text{P-T-E-Sub}$$

The typing rule for relabeling ensures that the pc label is lower than or equal to the resulting label. We have the standard subtyping rule, which uses the same notion of label subtyping introduced above.

3.3 Noninterference

We prove a noninterference theorem for *polC*'s type system by adapting the proof technique used in FlowML [45]. We extend our language to include pairs of expressions and pairs of values to simulate two executions that differ in "high" values. We only explain the key definitions for the theorem.

We first define equivalences of expressions in terms of an attacker's observation. We assume that the attacker knows the program and can observe expressions at the security level ℓ_A. To be consistent, when ℓ_A is not $\top$ or $\bot$, the

attacker's policy is written $\ell_A{::}\top$. Intuitively, an expression of type $b\ \rho$ should not be visible to the attacker if existing declassification functions cannot relabel data with label ρ down to $\ell_A{::}\top$. For instance, if $\rho = H{::}L{::}\bot$ and there is no declassification function from H to L, then an attacker at L cannot distinguish between two different integers v_1 and v_2 of type $\mathsf{int}\ \rho$. On the other hand, if there is a function $f :_{\mathsf{d\&e}} \mathsf{int}\ H{::}\top \to L{::}\bot$, then v_1 and v_2 are distinguishable by the attacker. We define when a policy ρ is in H w.r.t. the attacker's label, the function context, and the relabeling operations (when values of type $b\ \rho$ are not observable to the attacker) as follows. $\rho \in H$ if ρ cannot be *rewritten* to be a policy that is lower or equal to the attackers' policy. Here $F; R \vdash \rho \rightsquigarrow \rho'$ holds when $\rho = \ell_1{::}\cdots{::}\ell_i{::}\rho'$ and there is a sequence of relabeling operations in F and R, using which ρ can be rewritten to ρ'. For instance, when $\ell_A = \bot$

$$F_1 = \texttt{encodeA} : (\mathsf{d\&e})\mathsf{int}\ (\mathit{AlicePrivate}, \bot_I) :: \top \to \mathsf{int}\ (\bot_S, \mathit{EncodedBal}) :: \bot$$
$$F_2 = F_1, \texttt{yao\_execA} : (\mathsf{d\&e})\mathsf{int}\ (\bot_S, \mathit{EncodedBal}) :: \top \to \mathsf{int}\ \bot$$
$$\ell_A; \cdot; \cdot \vdash\ (\mathit{AlicePrivate}, \bot_I) \in H \qquad \ell_A; F_1; \cdot \vdash\ (\bot_S, \mathit{EncodedBal}) \in H$$
$$\ell_A; F_2; \cdot \nvdash\ (\bot_S, \mathit{EncodedBal}) \in H$$

Our noninterference theorem (defined below) states that given an expression e that is observable by the attacker, and two equivalent substitutions δ_1 and δ_2 for free variables in e (denoted $\delta_1 \approx_H \delta_2$), and both $e\delta_1$ and $e\delta_2$ terminate, then they must evaluate to the same value. In other words, the values of subexpressions that are not observable by the attacker do not influence the value of observable expressions.

Theorem 1 (Noninterference). *If $D; F; \Gamma; \bot \vdash e : s$, e does not contain any relabeling operations, given attacker's label ℓ, and substitution δ_1, δ_2 s.t. $F \vdash \delta_1 \approx_H \delta_2 : \Gamma$, and $\ell; F; \cdot \vdash \mathit{labOf}(s) \notin H$ and $\Psi \vdash \emptyset\ /\ e\delta_1 \longrightarrow^* \sigma_1\ /\ v_1$ and $\Psi \vdash \emptyset\ /\ e\delta_2 \longrightarrow^* \sigma_2\ /\ v_2$, then $v_1 = v_2$.*

It follows from Noninterference that given $D; F; x{:}\mathsf{int}\ \ell_1{::}\cdots{::}\ell_n{::}\bot \vdash e : \mathsf{int}\ \ell_n{::}\top$ where the attacker's label is ℓ_n, the attacker can only gain knowledge about the value for x if there is a sequence of declassification/endorsement functions f_is that remove label ℓ_i from the policy to reach $\ell_n{::}\top$. If $\ell_i \not\sqsubseteq \ell_{i+1}$, the f_is have to be applied in the correct order, as dictated by the typing rules.

4 Embedding in a Nominal Type System

The type system of *polC* can encode interesting security policies and help programmers identify subtle bugs during development. However, implementing a feature-rich language with *polC*'s type system requires non-trivial effort. Moreover, only programmers who are willing to rewrite their codebase in this new language can benefit from it. Rather than create a new language, `FlowNotation` leverages C's type system to enforce policies specified by *polC*'s types.

The mapping between the concrete workflow of `FlowNotation`, *polC* and μC, and the algorithms defined here is shown in Fig. 1. We first define a simple imperative language μC with nominal types and annotations, which models

the fragment of C that `FlowNotation` works within. We show how the annotated types and expressions can be mapped to types and expressions in *polC* in Sect. 4.1. Then in Sect. 4.2, we show how to translate *polC* programs back to μC. These two algorithms combined describe the core algorithm of `FlowNotation`. We prove our translation correct in Sect. 4.3.

4.1 μC and Annotated μC

Expressions and the typing context names of μC are the same as those in *polC*. The types in μC do not have information flow policies.

$$
\begin{array}{llll}
\textit{Expressions} & e ::= \cdots \mid \mathsf{let}\, x : \beta = e_1 \,\mathsf{in}\, e_2 & \textit{Typ. Annot.} & \beta ::= a \mid a_1 \to a_2 \\
\textit{Basic types} & \pi ::= T \mid \mathsf{int} \mid \mathsf{unit} \mid \mathsf{ptr}(\tau) & \textit{Types} & \tau ::= \pi \mid \pi_1 \to \pi_2 \\
\textit{Annotation} & a ::= \pi \mid T \;\mathsf{at}\; \rho \mid \mathsf{int} \;\mathsf{at}\; \rho \mid \mathsf{ptr}(\beta) \;\mathsf{at}\; \rho & & \\
\textit{Annot. typedef} & D_a ::= \cdot \mid D_a, T \mapsto \mathsf{struct}\; T\{a_1, \cdots, a_k\} & & \\
\textit{Annot. Func.} & F_a ::= \cdot \mid F_a, f : a_1 \to a_2 \mid F_a, f : (\mathsf{d\&e})a_1 \to a_2 & &
\end{array}
$$

Programmers will provide policy annotations, denoted β, which are very similar to labeled types s. We keep them separate, as programmers do not need to write out the fully labeled types. A programmer can annotate defined record types T at ρ, integers int at ρ, both the content and the pointer itself $\mathsf{ptr}(\beta)$ at ρ, or the record type $\mathsf{struct}\; T\{\beta_1, \cdots, \beta_k\}$. The last case is used to annotate type declarations in the context D. We extend expressions with annotated expressions; $\mathsf{let}\, x : a = e_1 \,\mathsf{in}\, e_2$. We assume that let bindings, type declarations, and function types are the only places where programmers provide annotations.

4.2 Translating Annotated Programs to μC

Instead of defining an algorithm to translate an annotated μC program e_a to another μC program, we first define an algorithm that maps e_a into a program e_l in *polC*; then an algorithm that translates e_l to a μC program.

Mapping from Annotated μC to *polC*. This mapping helps make explicit all the assumptions and necessary declassification and endorsement operations needed to interpret those annotations as proper *polC* types and programs.

We write $\langle\!\langle \beta \rangle\!\rangle$ to denote the mapping of unannotated and annotated μC types to *polC* types. Unannotated types are given a special label U (unlabeled, defined as $(\bot_S, \bot_I)$); annotated types are translated as labeled types. All function types are given the pc label $\bot$, so the function body can be typed with few restrictions. The mapping is straightforwardly defined over the structure of the annotated types and we show a few rules below.

$$
\frac{\pi \in \{\mathsf{int}, T\}}{\langle\!\langle \pi \rangle\!\rangle = \pi\ U} \qquad \frac{\pi \in \{\mathsf{int}, T\}}{\langle\!\langle \pi \;\mathsf{at}\; \rho \rangle\!\rangle = \pi\ \rho} \qquad \frac{\forall i \in [1,2], \langle\!\langle a_i \rangle\!\rangle = t_i}{\langle\!\langle a_1 \to a_2 \rangle\!\rangle = [\bot](t_1 \to t_2)}
$$

$$\frac{D_a; F_a; \Gamma_a \vdash \langle\!\langle v \rangle\!\rangle \Rightarrow lv \qquad tpOf(lv) = T\ \rho \qquad D_a(T) = (\mathsf{struct}\ T\{\beta_1, \cdots, \beta_n\}) \qquad \forall i \in [1,n], \rho = labOf(\langle\!\langle \beta_i \rangle\!\rangle)}{D_a; F_a; \Gamma_a; t \vdash \langle\!\langle v.i \rangle\!\rangle \Rightarrow lv.i}\ \text{L-FIELD-U}$$

$$\frac{D_a; F_a; \Gamma_a \vdash \langle\!\langle v \rangle\!\rangle \Rightarrow lv \qquad tpOf(lv) = T\ \rho \qquad D_a(T) = (\mathsf{struct}\ T\{\beta_1, \cdots, \beta_n\}) \qquad \exists i \in [1,n], \rho \neq labOf(\langle\!\langle \beta_i \rangle\!\rangle)}{D_a; F_a; \Gamma_a; t \vdash \langle\!\langle v.i \rangle\!\rangle \Rightarrow \mathsf{let}\ y : T\ \bot = \mathsf{reLab}(\bot \Leftarrow \rho)\ lv\ \mathsf{in}\ (y@T\ \bot).i}\ \text{L-FIELD}$$

$$\frac{\begin{array}{c} D_a; F_a; \Gamma_a \vdash \langle\!\langle v_1 \rangle\!\rangle \Rightarrow lv_1 \qquad tpOf(lv_1) = \mathsf{int}\ \rho \\ D_a; F_a; \Gamma_a; t \vdash \langle\!\langle e_2 \rangle\!\rangle \Rightarrow le_2 \qquad D_a; F_a; \Gamma_a; t \vdash \langle\!\langle e_3 \rangle\!\rangle \Rightarrow le_3 \end{array}}{\begin{array}{l} D_a; F_a; \Gamma_a; t \vdash \langle\!\langle \mathsf{if}\ v_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle\!\rangle \\ \quad \Rightarrow \mathsf{let}\ x : \mathsf{int}\ \bot = (\mathsf{reLab}(\bot \Leftarrow \rho)\ lv_1)\ \ \mathsf{in}\ \mathsf{if}\ x@\mathsf{int}\ \bot\ \mathsf{then}\ le_2\ \mathsf{else}\ le_3 \end{array}}\ \text{L-IF}$$

Fig. 3. Mapping of expressions (selected rules)

Expressions have two sets of mapping rules: $D_a; F_a; \Gamma_a; s \vdash \langle\!\langle e \rangle\!\rangle \Rightarrow le$ and $D_a; F_a; \Gamma_a \vdash \langle\!\langle v \rangle\!\rangle \Rightarrow lv$. The mapping rules use the annotated typing contexts: D_a, F_a, and Γ_a. The reading of the first judgement is that an annotated expression e is mapped to a labeled expression le given annotated typing contexts D_a, F_a, Γ_a, and e's *polC* type s. The second judgment is similar, except that it only applies to values and the type of v is not given. Here le and lv are expressions with additional type annotations of form $@s$ to ease the translation process from *polC* to μC. For instance, $n@\mathsf{int}\ U$ means that n is an integer and it is supposed to have the type $\mathsf{int}\ U$. This way, we can give the same integer different types, depending on the context under which they are used: $n@\mathsf{int}\ U$ and $n@\mathsf{int}\ \rho$ are translated into different terms.

A value is mapped to itself with its type annotated. For example, integers are given $\mathsf{int}\ U$ type, since they are unlabeled. Selected expression mapping rules are listed in Fig. 3. The tricky part is mapping expressions whose typing rules in *polC* require label comparison and join operations. Obviously, the μC type system cannot enforce such complex rules. Instead, we add explicit relabeling to certain parts of the expression to ensure that the types of the translated μC program enforce the same property as types in the corresponding *polC* program.

There are two rules for record field access: one without explicit relabeling (L-FIELD) and one with (L-FIELD-U). Rule L-FIELD applies when all the elements in the record have the same label as the record itself. Rule L-FIELD-U explicitly relabels the record first, so the record type changes from $T\ \rho$ to $T\ \bot$, resulting in the field access having the same label as the element. This is because when the labels of the elements are not the same as the record, the typing rule P-T-E-FIELD will join the type of the field with the label of the record. However, this involves label operations, which μC's type system cannot handle. L-DEREF and L-ASSIGN are similar. The mapping of if statements (L-IF) relabels the conditional v_1 to have $\mathsf{int}\ \bot$ type, so the branches are typed under the same program counter as the if expression. We write $\mathsf{reLab}(\bot \Leftarrow \rho)$ as a short hand

for a sequence of relabeling operations $\mathsf{reLab}(\ell{::}\bot \Leftarrow \ell_n{::}\top)\cdots\mathsf{reLab}(\ell_i{::}\bot \Leftarrow \ell_{i-1}{::}\top)\cdots\mathsf{reLab}(\ell_2{::}\bot \Leftarrow \ell_1{::}\top)$ where $\rho = \ell_1{::}\cdots{::}\ell_n{::}\ell$ and ℓ is either $\top$ or $\bot$. The implications of relabeling operations are discussed at the end of this section.

Translation from *polC* to μC. The translation of types is shown below. It returns a μC type and a set of new type definitions. We use a function $genName(t, \rho)$ to deterministically generate a string based on t and ρ as the identifier for a record type. It can simply be the concatenation of the string representation of t and ρ, which is indeed what we implemented for C (Sect. 5).

$$\frac{\rho \in \{U, \bot\}}{[\![\mathsf{int}\ \rho]\!]_D = (\mathsf{int}, \cdot)} \qquad \frac{\rho \notin \{U, \bot\} \qquad T = genName(\mathsf{int}, \rho)}{[\![\mathsf{int}\ \rho]\!]_D = (T, T \mapsto \mathsf{struct}\ T\ \{\mathsf{int}\})}$$

$$\frac{\rho \notin \{U, \bot\} \qquad T' = genName(T, \rho) \qquad T \mapsto \mathsf{struct}\ T\ \{\tau_1, \cdots, \tau_n\} \in D}{[\![T\ \rho]\!]_D = (T', T' \mapsto \mathsf{struct}\ T'\ \{\tau_1, \cdots, \tau_n\})}$$

We distinguish between a type with a label that is U or $\bot$ and a meaningful label. The translation of the type $b\ U$ is simply b. This is because $b\ U$ is mapped from an unannotated type b to begin with, so the translation returns its original type. Similarly $b\ \bot$ is generated by our relabeling operations during the mapping process, and should be translated to its original type b. A type annotated with a meaningful policy ρ is translated into a record type to take advantage of nominal typing. The translation also returns the new type definition. This would also prevent label subtyping based on the security lattice. However, this is acceptable given our application domain because the labels provided by programmers are distinct points in the lattice that are not connected by any partial order relations except the $\top$ and $\bot$ elements. Record types are translated to record types and the fields of the labeled record type $T\ \rho$ have the same type as those for T, stored in the translated context D. This works because we assume that all labeled instances of the record type T (i.e., all $T\ \rho$) share the same definition.

Expression translation recursively translates the sub-expressions. The μC type system does not have complex label checking, so rule T-App-De has to insert label conversions. The argument label is cast from $\ell_1{::}\ell_2{::}\rho'$ to $\ell_1{::}\top$, as required by f, and the result of the function is cast from $\ell_2{::}\bot$ to $\ell_2{::}\rho'$.

$$\frac{\begin{array}{c} tpOf(lv_f) = (\mathsf{d\&e})[pc](t_1 \to t_2)^{\rho_f} \\ [\![lv_f]\!]_D = (v_f, D_f) \qquad tpOf(lv_a) = b\ \rho \\ \rho = \ell_1{::}\ell_2{::}\rho' \qquad [\![\mathsf{reLab}(\ell_1{::}\top \Leftarrow \rho) lv_a]\!]_D = (e', D_1) \\ [\![\mathsf{reLab}(\ell_2{::}\rho' \Leftarrow \ell_2{::}\bot)(z@b\ \ell_2{::}\bot)]\!]_D = (e'', D_2) \\ [\![t_1]\!]_D = (\tau_1, D_3) \qquad [\![t_2]\!]_D = (\tau_2, D_4) \end{array}}{\begin{array}{l} [\![lv_f\ lv_a]\!]_D = (\mathsf{let}\ y : \tau_1 = e'\ \mathsf{in}\ \mathsf{let}\ z : \tau_2 = v_f\ y \\ \qquad \mathsf{in}\ e'', D_f \cup D_1 \cup D_2 \cup D_3 \cup D_4) \end{array}}\ \text{T-App-De}$$

These operations are different from the ones inserted during the mapping process because they only exist to help μC simulate the E-App-De typing rule in *polC*, but do not really have declassification or endorsement effects. The relabeling operations are translated to record operations, as shown in Fig. 4.

$$\frac{\begin{array}{c} tpOf(lv_f) = (\mathsf{d\&e})[pc](t_1 \to t_2)^{\rho_f} \\ [\![lv_f]\!]_D = (v_f, D_f) \qquad tpOf(lv_a) = b\ \rho \\ \rho = \ell_1 :: \ell_2 :: \rho' \qquad [\![\mathsf{reLab}(\ell_1 :: \top \Leftarrow \rho) lv_a]\!]_D = (e', D_1) \\ [\![\mathsf{reLab}(\ell_2 :: \rho' \Leftarrow \ell_2 :: \bot)(z@b\ \ell_2 :: \bot)]\!]_D = (e'', D_2) \\ [\![t_1]\!]_D = (\tau_1, D_3) \qquad [\![t_2]\!]_D = (\tau_2, D_4) \end{array}}{\begin{array}{l} [\![lv_f\ lv_a]\!]_D = (\mathsf{let}\ y : \tau_1 = e'\ \mathsf{in}\ \mathsf{let}\ z : \tau_2 = v_f\ y \\ \qquad \mathsf{in}\ e'', D_f \cup D_1 \cup D_2 \cup D_3 \cup D_4) \end{array}}\ \text{T-App-DE}$$

$$\frac{[\![lv]\!]_D = (v, D_1) \quad tpOf(lv) = b\ \rho\ (b \text{ is not a struct type}) \quad \rho' \notin \{\bot, U\} \qquad \rho \notin \{\bot, U\} \qquad [\![b\ \rho']\!]_D = (T, D_2)}{[\![\mathsf{reLab}(\rho' \Leftarrow \rho) lv]\!]_D = (\mathsf{let}\ x = v.1\ \mathsf{in}\ (T)\{x\}, D_1 \cup D_2)}\ \text{T-ReLab-N1}$$

$$\frac{[\![lv]\!]_D = (v, D_1) \quad tpOf(lv) = b\ \rho\ (b \text{ is not a struct type}) \quad \rho' \notin \{\bot, U\} \qquad \rho \in \{\bot, U\} \qquad [\![b\ \rho']\!]_D = (T, D_2)}{[\![\mathsf{reLab}(\rho' \Leftarrow \rho) lv]\!]_D = ((T)\{v\}, D_1 \cup D_2)}\ \text{T-ReLab-N2}$$

$$\frac{[\![lv]\!]_D = (v, D_1) \qquad tpOf(lv) = b\ \rho \quad b \text{ is not a struct type} \qquad \rho \notin \{\bot, U\} \qquad \rho' \in \{\bot, U\}}{[\![\mathsf{reLab}(\rho' \Leftarrow \rho) lv]\!]_D = (v.1 D_1)}\ \text{T-ReLab-N3}$$

$$\frac{[\![lv]\!]_D = (v, D_1) \qquad labOf(lv) = b\ \rho \qquad \rho, \rho' \in \{U, \bot\}}{[\![\mathsf{reLab}(\rho' \Leftarrow \rho) lv]\!]_D = (v, D_1)}\ \text{T-ReLab-same}$$

$$\frac{\begin{array}{c} \rho \notin \{\bot, U\} \text{ or } \rho' \notin \{\bot, U\} \\ tpOf(lv) = T\ \rho \qquad [\![T\ \rho']\!]_D = (T', D_1) \qquad [\![lv]\!]_D = (v, D_2) \end{array}}{\begin{array}{l} [\![\mathsf{reLab}(\rho' \Leftarrow \rho) lv]\!]_D = \mathsf{let}\ x_1 = v.1\ \mathsf{in}\ \cdots \mathsf{let}\ x_n = v.n \\ \qquad \mathsf{in}\ (T')\{x_1, \cdots, x_n\}, D_1 \cup D_2) \end{array}}\ \text{T-ReLab-Struct}$$

Fig. 4. Selected relabeling rules

Rule T-Relab-N1 relabels a value with a labeled type. The translated expression is a reassembled record using the fields of the original record. Rule T-Relab-N2 relabels an expression with a U and $\bot$ label to a meaningful label. In this case, the translated expression is a record. Rule T-Relab-N3 translates an expression relabeled from a meaningful label to a U or $\bot$ label to a projection of the record. The next rule, T-Relab-Same, does not change the value itself, because we are just relabeling between U and $\bot$ labels. The final relabeling rule, T-Relab-Struct, deals with records. In this case, we simply return the reassembled record because record types that only differ in labels have the same types for the fields.

4.3 Correctness

We prove a correctness theorem, which states that if our translated nominal type system declares an expression e well-typed, then the labeled expression e_l, where e is translated from, is well-typed under *polC*'s type system. Formally:

Theorem 2 (Translation Soundness (Typing)). *If* $D_a; F_a; \Gamma_a; s \vdash \langle\!\langle e \rangle\!\rangle = le$, $\langle\!\langle D_a \rangle\!\rangle = D_l$, $\langle\!\langle F_a \rangle\!\rangle = F_l$, $\langle\!\langle \Gamma_a \rangle\!\rangle = \Gamma_l$, $[\![D_l]\!] = D$, $[\![\Gamma_l]\!]_D = (\Gamma, D_1)$, $[\![F_l]\!]_D = (F, D_2)$, $[\![le]\!]_D = (e', D_3)$, *and* $D \cup D_1 \cup D_2 \cup D_3; F; \cdot; \Gamma \vdash e' : \tau$ *implies* $D_l; F_l; \cdot; \Gamma_l \vdash tmOf(le) : s$ *and* $[\![s]\!] = (\tau, \_)$

Here, $tmOf(le)$ denotes an expression that is the same as le, with labels (e.g., @int U) removed. The proof is by induction over the derivation of $D_a; F_a; \Gamma_a; s \vdash \langle\!\langle e \rangle\!\rangle \Rightarrow le$. It is not hard to see that the translated program has the same behavior as the original program, as they only differ in that the translated program has many indirect record constructions and field accesses.

Relabeling Precision. It is clear from the mapping algorithm that a number of powerful relabeling operations are added. In all cases (except the if statement) we could do better by not relabeling all the way to bottom, but to the label of the sub-expressions. However, that would require a heavy-weight translation algorithm that essentially does full type-checking.

Implicit Flows. The security guarantees of programs that require relabeling operations to be inserted are weakened in the sense that in addition to the special declassification and endorsement functions, these relabeling operations allow additional observation by the attacker. This means that the resulting program can implicitly leak information via branches, de-referencing, and record field access. However, for our application domain we aim to check simple data usage and function call patterns which, as seen in our case studies, manifest errors with explicit flows. These policy violations are still detected if we don't have recursive types (See the full paper for explanations and our case studies are not affected).

5 Implementation

We explain how the annotations and translation algorithms of `FlowNotation` are implemented for C.

Translation of Annotations for Simple Types. Utilizing C's nominal typing via the `typedef` mechanism is key to realizing *polC* type system within the bounds of C's type system. The declaration of the *polC* type $t\ \rho$ in C will be `typedef struct {`t `d;} `ρ`@`t`;`. Here ρ@t is a string representing the type $t\ \rho$ and it is simply a concatenation of the string representation of the policy ρ and the type t. Consider the annotated code snippet. `#requires l1:secrecy then l2:secrecy int x;` In *polC*, the type of x is `int` $(l1, \bot_I)$::$(l2, \bot_I)$::$\bot$. The generated C typedef is `typedef struct {int d;} l1S_l2S_int;` This definition contains the original type, which allows access to the original data stored in x in the transformed program.

Structures and Unions. We allow programmers to annotate structures in two ways: an instance of a structure can be annotated with a particular policy, or individual fields of an instance of a structure can be given annotations. The names of structures hold a particular significance within C since they are nominal types, and thus, they need to be properly handled. Unions are treated in a parallel manner, so we omit the details.

A policy on an instance of a structure is annotated and translated following the same formula as annotations on simple C types. Suppose we have the following annotation and code: `#requires l1:secrecy then l2:secrecy struct foo x;`. `FlowNotation` will produce the following generated type definition: `typedef struct {struct foo d;} l1S_l2S_foo;` This is different from the algorithm in Sect. 4, where structures are not nested and annotations are applied to structure definitions rather than instances. This is done in the implementation because the definition of `foo` might be external and therefore may not be known to the translation algorithm, so we simply nest the entire structure.

Finally, we explain how member accesses are handled. Suppose a struct `foo` contains members `f1` and `f2`, and an annotation of policy `p` has been placed on member `f1`, but no annotation has been placed on member `f2`. The generated type definition for the structure is as follows: `typedef struct { p_int f1; foo d; } p_foo;`. Assume `x` has type `p_foo`. Access to `f1` is still `x.f1`, since there is a copy of it in `x`. Access to `f2` is rewritten to `x.d.f2`. The field initialization is rewritten similarly. `foo x={.f1=1,.f2=2};` is transformed to this: `foo x={.f1=1,.d={.f2=2}};`.

Pointers. We provide limited support for pointers. Here is an example of how annotations on pointers are handled: `#requires AlicePriv:secrecy int* x;` The translated code is below; a type definition of struct `AlivePrivS_int` is generated: `AlicePrivS_int* x;`. The following function can receive `x` as an argument because the annotation for its parameter matches that of `x`: `#param AlicePriv:secrecy int f(int* x){...}`.

The annotation for pointers only annotates the content of the pointer. Even though *polC* allows policies on the pointer themselves, we did not implement that feature. We also do not support pointer arithmetic, which is difficult to handle for many static analysis tools, especially lightweight ones like ours. However, our system will flag aliasing of pointers across mismatched annotated types. Our system will also flag pointer arithmetic operations on annotated types as errors. Programmers can encapsulate those operations in trusted functions and annotate them to avoid such errors.

Typecasts. The C type system permits typecasts, allowing one to redefine the type of a variable in unsound ways. Casting of non-pointer annotated types will be flagged as an error by `FlowNotation`. This is because our types are realized as C structures; type checkers do not allow arbitrary casting of structures. However, our tool cannot catch typecasts made on annotated pointers; a policy on a pointer will be lost if a typecast is performed.

Limitations. As previously mentioned, we do not handle pointer arithmetic. We only provide limited support for function pointers. We do not support C's builtin operators, such as the unary ++. We do not support typecasts on pointers, nor can we flag violations due to implicit void pointer conversion. We provide partial support for variadic functions. Finally, we do not support using `#return` with a function that has a `void` return type. These are careful design choices we made so our tool is lightweight and remains practical; we emphasize that our tool is not meant for verification. Further explanations can be found in our full paper, and limitations of our type system can be found in Sect. 4.3.

6 Case Studies

We evaluate the effectiveness of `FlowNotation` at discovering violations of secrecy, integrity, and sequencing API usage policies on several open-source cryptographic libraries. Our results are summarized in Fig. 5. We examine: Obliv-C, a compiler for dialect of C directed at secure computation [54,55]; SCDtoObliv, a set of floating point circuits synthesized into C code [56]; the Absent-minded Crypto Kit, a library of Secure Computation protocols and primitives [33,34]; Secure Mux, a secure multiplexer application [59]; the Pool Framework, a secure computation memory management library [58,59]; Pantaloons RSA, the top GitHub result for an RSA implementation in C [43]; MiniAES, an AES multiparty computation implementation [30,31]; Bellare-Micali OT, an implementation of the Bellare-Micali oblivious transfer protocol [6]; Kerberos ASN.1 Encoder, the ASN.1 encoder module of Kerberos [1]; Gnuk OpenPGP-do, a portion of the OpenPGP module from gnuk [53]; Tiny SHA3, a reference implementation of SHA3 [46]. We determine application-specific policies and implement them with our annotations. Representative cases are explained next; additional cases are in the full paper.

Library	# Pol	Sec	Int	Seq	LoA	∼ LoC	Issues	RT(s)
Obliv-C Library	2	1	1	0	11	80	0	0.04
SCDtoObliv FP Circuits	4	4	0	0	10	43,000	1	5.55
ACK Oqueue	7	7	7	2	19	700	0	0.17
Secure Mux Application	4	3	4	0	11	150	0	0.06
Pool Framework	4	2	4	0	8	500	1	0.16
Pantaloons RSA	5	2	3	0	12	300	1	0.11
MiniAES	9	4	4	1	13	2000	0	0.08
Bellare-Micali OT	5	3	2	0	12	100	2	0.05
Kerberos ASN.1 Encoder	2	2	0	1	8	300	0	0.12
Gnuk OpenPGP-do	5	0	5	1	11	250	1	0.10
Tiny SHA3	3	3	0	1	6	200	0	0.10

Fig. 5. Evaluation Results. #Pol is the number of policies. Sec, int, seq are secrecy, integrity, and sequencing policies. LoA, LoC are lines of annotations, code. RT(s) is runtime in seconds.

SCDtoObliv Floating Point Circuits. First, we show that `FlowNotation` can be used to discover flaws in large, automatically generated segments of code that would be very difficult for a programmer to manually analyze.

SCDtoObliv [56] synthesizes floating point circuit in C via calls to boolean gate primitives implemented in C. While this approach produces performant floating point circuits for secure computation applications, the resulting circuit files are hard to interpret and debug. The smallest of these generated circuit files is around 4000 lines of C code while the largest is over 14,000 lines. We annotate particular wires based on the circuit function to check that particular invariants such as which bits should be used in the output and which bits should be flipped are maintained.

`FlowNotation` uncovered a flaw in the subtraction circuit. The Obliv-C subtraction circuit actually uses an addition circuit to compute $A + (-B)$. The function that does the sign bit flipping, `__obliv_c__flipBit`, is annotated so that it can only accept an input with the *needsFlipping* label as follows.

```
#param needsFlipping:secrecy
void __obliv_c__flipBit(OblivBit* src)
```

Our tool reports an error; rather than the sign bit of the second operand being given to `__obliv_c__flipBit` the sign bit of the *first* operand was given to `__obliv_c__flipBit`. Instead of computing $A + (-B)$ the circuit computes $(-A) + B$; the result of evaluating the circuit is negated with respect to the correct answer.

Gnuk OpenPGP-DO. The last case study shows that `FlowNotation` can uncover a known null-pointer dereferencing bug and another potential bug in the gnuk OpenPGP-DO file, which handles OpenPGP Smart Card Data Objects (DO). We present the latter in the full paper.

The function `w_kdf` handles the reading or writing of DOs that support encryption via a Key Derivation Function (KDF) in the OpenPGP-DO file.

```
static int rw_kdf (uint16_t tag, int with_tag,
   const uint8_t *data, int len, int is_write)
```

If the data is being read, it is copied to a buffer via the function `copy_do_1`: `static void copy_do_1(uint16_t tag, const uint8_t *do_data, int with_tag)`. One invariant is that the `do_data` pointer must point to a valid segment of data; it must not be null. We provide the following annotation:

```
#param(2) check_valid_ptr:integrity
static void copy_do_1(uint16_t tag, const uint8_t *do_data, int
    with_tag)
```

This annotation states that the second parameter will only be accepted if it has been endorsed by a function that returns data annotated with the *check_valid_ptr* label. We provide such a function and rewrite all nullity checks to use it.

```
#return check_valid_ptr:integrity
const uint8_t *check_do_ptr(const uint8_t *do_ptr)
```

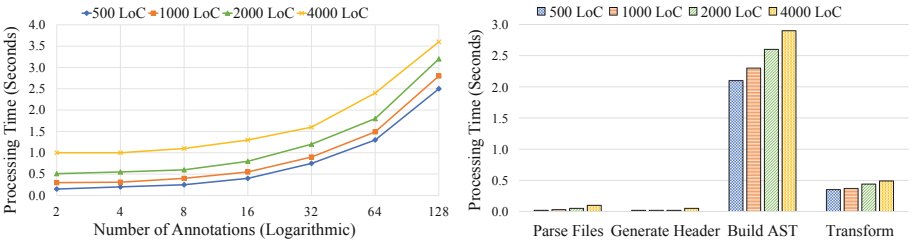

Fig. 6. Processing time vs Number of annotations and time per processing stage

Returning back to the `rw_kdf` function, when data is being read, the following call of `copy_do_1` occurs: `copy_do_1(tag, do_ptr[NR_DO_KDF], with_tag);` The issue is `copy_do_1` is annotated to require a null-pointer check for parameter two, but that check was not performed.

Performance Evaluation. We evaluate the performance of `FlowNotation` on synthetically generated C programs and annotations. To elicit worse-case behavior, the generated annotations are predominantly sequencing annotations constructed from a set of templates representative of common API patterns from our case studies.

We evaluate how the runtime of `FlowNotation` is affected by the program size and the number of annotations (Fig. 6). We evaluate the runtime of four C programs, with 500, 1000, 2000, and 4000 lines of code respectively. For each program, we increase the number of annotations, up to 128. `FlowNotation` is efficient: all experiments finish within 4 s. `FlowNotation` is intended to be run on individual modules (libraries) that rarely exceed a couple thousand lines of code unless they are automatically generated, like the SCDtoObliv circuit file (14,000 LoC). Even then, it finishes within 6 s.

To better understand how each component of `FlowNotation` contributes to the processing time, we profile execution time for each part. The results are summarized in Fig. 6, which shows a cross-section of Fig. 6 with only the samples with 128 annotations. The four stages of `FlowNotation` are: "Parse Files", where annotations are retreived; "Generate Header", where the header file with definitions for the transformed types is generated; "Build AST", where the C parsing library, pycparser [7] builds an AST; "Transform", where the implementation of the translation algorithm of `FlowNotation` runs. The majority of the overhead is due to the C parsing library we use.

7 Related Work

Tools for Analyzing C Programs. Many vulnerabilities stem from poorly written C programs. As a result, many C program analysis tools have been built. Several C model checkers (e.g. [4,10,11,24,38]) and program analysis tools [20, 27,29,42] are open source and readily downloadable. Our policies can be encoded

as state machines and checked by some of the tools mentioned above, which are general purpose and more powerful than ours but are not tuned for analyzing API usage patterns like ours.

Closest to our work is CQual [36]. Both theoretical foundations and practical applications of type qualifiers have been investigated [17,22,35,37,57]. Our annotations are type qualifiers and our work and prior work on type qualifiers share the goal of producing a lightweight tool to check simple secrecy and integrity properties. We additionally support sequencing of atomic qualifiers, which is a novel contribution. Both our's and prior work do not handle implicit flows. We prove noninterference of our core calculus, which other systems did not. Another difference is that CQual relies on a custom type checker, while our policies are checked using C's type system. Finally, CQual supports qualifier inference, which can reduce the annotation burden on programmers. We do not have general qualifier inference because we rely on existing C compiler's type checkers.

Information Flow Type Systems. Information flow type systems is a well-studied field. Several projects have extended existing languages to include information flow types (e.g., [44,45]). Sabelfeld et al. provided a comprehensive summary in their survey paper [47]. Most information flow type systems do not deal with declassification. At most, they will include a "declassify" primitive to allow information downgrade, similar to our relabel operations. However, we have not seen work where the sequence of labels is part of the information flow type like ours, except for JRIF [39]. As a result, we are able to prove a noninterference theorem that implies API sequencing. JRIF uses finite state automata to enforce sequencing policies, which can entail a large runtime overhead.

Other projects that target enforcement of sequencing policies similar to those we have presented rely on runtime monitoring, not types [5,9,18,19,23,50].

Linear Types and Typestate. Our sequencing policies are tangentially related to other type systems that aim to enforce API contracts. This line of work includes typestate and linear types [2,32,48]. The idea is that by using typestate/linear types one can model and check behaviors such as files being opened and closed in a balanced manner [2]. However, unlike in typestate the types on variables don't change in our system; when a part of a policy is fulfilled there is a new variable that "takes on" the rest of the policy.

Cryptographic Protocol Verification. Several projects have proposed languages to make verification of cryptographic programs more feasible: Jasmine, Cryptol, Vale, Dafny, F*, and Idris [3,15,16,40,41,49], to name a few. There are also general tools for verifying cryptographic protocols [8,12–14,25,26,28]. These languages and tools are general purpose and more powerful than ours. However, none of these tools directly support checking properties of C implementations of cryptographic libraries like we do. Bhargavan et al.'s work uses refinement types to achieve similar goals as ours [13]. The annotated types can be viewed as refinement types: $\{x : \tau \mid \rho\}$, where the policy is encoded as a predicate. Their system is more powerful, however it only supports F# code.

Acknowledgement. This work is supported in part by the National Science Foundation via grants CNS1704542 and CNS1464113, and by the National Institutes of Health via award 1U01EB023685-01.

References

1. Kerberos ASN.1 encoder. https://github.com/krb5/krb5/tree/master/src/lib/krb5/asn.1. Accessed 2018
2. Aldrich, J., Sunshine, J., Saini, D., Sparks, Z.: Typestate-oriented programming. In: Proceedings of OOPSLA (2009)
3. Almeida, J.B., et al.: Jasmin: high-assurance and high-speed cryptography. In: Proceedings of CCS (2017)
4. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: Proceedings of POPL (2002)
5. Barany, G., Signoles, J.: Hybrid information flow analysis for real-world C code. In: Gabmeyer, S., Johnsen, E.B. (eds.) TAP 2017. LNCS, vol. 10375, pp. 23–40. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61467-0_2
6. Bellare, M., Micali, S.: Non-interactive oblivious transfer and applications. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 547–557. Springer, New York (1990). https://doi.org/10.1007/0-387-34805-0_48
7. Bendersky, E.: pycparser. https://github.com/eliben/pycparser. Accessed 2017
8. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. ACM Trans. Program. Lang. Syst. (TOPLAS) **33**(2), 8 (2011)
9. Beringer, L.: End-to-end multilevel hybrid information flow control. In: Jhala, R., Igarashi, A. (eds.) APLAS 2012. LNCS, vol. 7705, pp. 50–65. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35182-2_5
10. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast. Int. J. Softw. Tools Technol. Transfer **9**, 505–525 (2007)
11. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
12. Bhargavan, K., Fournet, C., Corin, R., Zalinescu, E.: Cryptographically verified implementations for TLS. In: Proceedings of CCS (2008)
13. Bhargavan, K., Fournet, C., Gordon, A.D.: Modular verification of security protocol code by typing. In: Proceedings of POPL (2010)
14. Blanchet, B.: An efficient cryptographic protocol verifier based on prolog rules. In: Proceedings of CSFW (2001)
15. Bond, B., et al.: Vale: verifying high-performance cryptographic assembly code. In: Proceedings of USENIX (2017)
16. Brady, E.: Idris, a general-purpose dependently typed programming language: design and implementation. J. Functional Program. **23**, 552–593 (2013)
17. Broadwell, P., Harren, M., Sastry, N.: Scrash: a system for generating secure crash information. In: Proceedings of SSYM (2003)
18. Broberg, N., van Delft, B., Sands, D.: Paragon for practical programming with information-flow control. In: Shan, C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 217–232. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03542-0_16
19. Broberg, N., Sands, D.: Paralocks: role-based information flow control and beyond. In: Proceedings of POPL (2010)

20. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of OSDI (2008)
21. Cassel, D., Huang, Y., Jia, L.: FlowNotation technical report. https://arxiv.org/abs/1907.01727 (2019)
22. Chin, B., Markstrum, S., Millstein, T.: Semantic type qualifiers. In: Proceedings of PLDI (2005)
23. Chong, S., Myers, A.C.: End-to-end enforcement of erasure and declassification. In: Proceedings of CSF (2008)
24. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
25. Cortier, V., Warinschi, B.: Computationally sound, automated proofs for security protocols. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 157–171. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_12
26. Costanzo, D., Shao, Z., Gu, R.: End-to-end verification of information-flow security for C and assembly programs. In: Proceedings of PLDI (2016)
27. Cousot, P., et al.: The ASTREÉ analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_3
28. Cremers, C.J.F.: The scyther tool: verification, falsification, and analysis of security protocols. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 414–418. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_38
29. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33826-7_16
30. Damgård, I., Zakarias, R.: MiniAES repository. https://github.com/AarhusCrypto/MiniAES. Accessed 2017
31. Damgård, I., Zakarias, R.: Fast oblivious AES a dedicated application of the MiniMac protocol. In: Pointcheval, D., Nitaj, A., Rachidi, T. (eds.) AFRICACRYPT 2016. LNCS, vol. 9646, pp. 245–264. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-31517-1_13
32. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: Proceedings of PLDI (2001)
33. Doerner, J.: Absentminded crypto kit repository. https://bitbucket.org/jackdoerner/absentminded-crypto-kit/. Accessed 2017
34. Doerner, J., Shelat, A.: Scaling ORAM for secure computation. In: Proceedings of CCS (2017)
35. Evans, D.: Static detection of dynamic memory errors. In: Proceedings of PLDI (1996)
36. Foster, J.S., Fähndrich, M., Aiken, A.: A theory of type qualifiers. In: Proceedings of PLDI (1999)
37. Foster, J.S., Aiken, A.S.: Type qualifiers: lightweight specifications to improve software quality. Ph.D. thesis, University of California, Berkeley (2002)
38. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
39. Kozyri, E., Arden, O., Myers, A.C., Schneider, F.B.: JRIF: Reactive Information Flow Control for Java (2016). http://hdl.handle.net/1813/41194

40. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
41. Lewis, J.R., Martin, B.: Cryptol: high assurance, retargetable crypto development and validation. In: Proceedings of MILCOM (2003)
42. Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., Vigna, G.: DR. CHECKER: a soundy analysis for Linux kernel drivers. In: Proceedings of USENIX (2017)
43. McGee, M.: Pantaloons/RSA repository. https://github.com/pantaloons/RSA/. Accessed 2017
44. Myers, A.C.: JFlow: practical mostly-static information flow control. In: Proceedings of POPL (1999)
45. Pottier, F., Simonet, V.: Information flow inference for ML. In: Proceedings of POPL (2002)
46. Saarinen, M.J.O.: Tiny SHA3. https://github.com/mjosaarinen/tiny_sha3. Accessed 2017
47. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**, 5–19 (2003)
48. Strom, R.E., Yemini, S.: Typestate: a programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. **SE-12**, 157–171 (1986)
49. Swamy, N., Chen, J., Fournet, C., Strub, P., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: Proceedings of ICFP (2011)
50. Vachharajani, N., et al.: RIFLE: an architectural framework for user-centric information-flow security. In: Proceedings of MICRO (2004)
51. Volpano, D., Smith, G.: A type-based approach to program security. In: Bidoit, M., Dauchet, M. (eds.) CAAP 1997. LNCS, vol. 1214, pp. 607–621. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0030629
52. Yao, A.C.C.: How to generate and exchange secrets. In: Proceedings of SFCS (1986)
53. Yutaka, N.: Gnuk. https://www.fsij.org/category/gnuk.html. Accessed 2018
54. Zahur, S., David, E.: Obliv-C: a language for extensible data-oblivious computation (2015)
55. Zahur, S.: Obliv-C repository. https://github.com/samee/obliv-c/. Accessed 2017
56. Zahur, S., Cassel, D.: SCDtoObliv repository. https://github.com/samee/obliv-c/tree/obliv-c/SCDtoObliv. Accessed 2017
57. Zhang, X., Edwards, A., Jaeger, T.: Using CQUAL for static analysis of authorization hook placement. In: Proceedings of USENIX (2002)
58. Zhu, R., Huang, Y., Cassel, D.: Pool framework repository. https://github.com/jimu-pool/PoolFramework/. Accessed 2017
59. Zhu, R., Huang, Y., Cassel, D.: Pool: Scalable on-demand secure computation service against malicious adversaries. In: Proceedings of CCS (2017)