C Program Partitioning with Fine-Grained Security Constraints and Post-Partition Verification

Maxwell Levatich*, Robert Brotzman†, Benjamin Flin†, Ta Chen†, Rajesh Krishnan†, and Stephen A. Edwards*
*Department of Computer Science, *Columbia University*, New York, NY
†*Peraton Labs*, Basking Ridge, NJ

***Abstract*—We address the problem of program partitioning: dividing a program into isolated compartments that communicate via remote procedure calls to follow a security policy. Existing solutions for C programs often use a simple model that offers only "sensitive or not" control and do not provide formal guarantees of partition correctness. We present a C program partitioner for security-conscious applications that addresses these shortcomings through annotation with fine-grained security constraints (chiefly, declassification of sensitive data to select parties); from these annotations, we automatically determine a partition and auto-generate code for marshaling, serialization, and remote procedure calls. We provide post-partition verification, which leverages translation validation to show that output program partitions are behaviorally equivalent to their input programs and satisfy the security policy specified by annotations. We present results that show our approach is practical when partitioning large realistic C applications with non-trivial security constraints.**

***Index Terms*—Program partitioning, Program equivalence, Constraint solving, Multi-level security, Compartmentalization**

I. Introduction

Software systems, especially those written in C, suffer from security vulnerabilities that adversaries can exploit to gain unauthorized access to sensitive data. Even unprivileged code can expose vulnerabilities when it shares memory [1].

Program partitioning secures software against these vulnerabilities by splitting the program into isolated hardware enclaves and configuring them to communicate via remote procedure calls (RPC). A partition must obey a *security policy*: rules prescribing data ownership and whether it can be released to other enclaves (e.g., an authentication service should not release stored passwords, but must release validity information about passwords). Effective partitioning ensures that if an unprivileged enclave is compromised, sensitive data in other enclaves is not necessarily also compromised. Such partitioning is important in both military (e.g., coalition information sharing) and civilian applications (e.g., IT/OT separation in critical infrastructure, patient medical records, and avionics).

The C language is a natural target for automated program partitioning [2]–[9] because it is commonly used for security-critical applications yet inadvertent vulnerabilities are easy to introduce. These tools allow a developer to express a security policy in annotations; static analysis determines a partition.

This material is based upon work supported by DARPA under the GAPS project. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of DARPA.

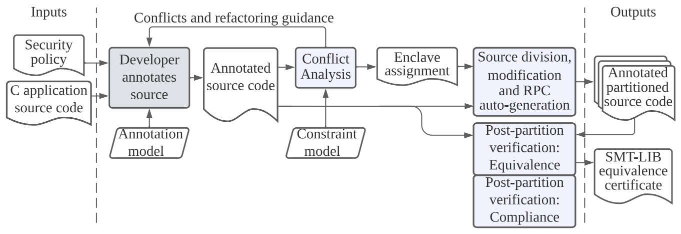
Inputs
Conflicts and refactoring guidance
Outputs
Security policy
C application source code
Developer annotates source
Annotation model
Annotated source code
Conflict Analysis
Constraint model
Enclave assignment
Source division, modification and RPC auto-generation
Post-partition verification: Equivalence
Post-partition verification: Compliance
Annotated partitioned source code
SMT-LIB equivalence certificate

Fig. 1. The operation of our C program partitioner

Unfortunately, existing C partitioners use a simplistic annotation model that does not allow sensitive data to be declassified to only certain enclaves. Furthermore, existing program partitioners do not make formal correctness guarantees.

We present *CAPO*, an annotation-driven C program partitioner for hardware isolation for security that provides *fine-grained data sharing* and *post-partition verification*. Fig. 1 illustrates how our system operates: from a C application and a Bell-LaPadula-inspired [10] security policy, the developer annotates source code with ownership and declassification rules. Developer-audited *annotated functions* may declassify sensitive data and pass it to a *shareability set* of other levels.

An automated *conflict analysis* step passes the annotated code to a constraint solver that attempts to distribute functions and global variables among enclaves according to the security policy. Success then directs a source transformation and RPC generation tool to generate an executable partition.

Finally, our *post-partition verification* tool, *ParTV*, evaluates whether the partitioned result correctly implements the original source and its security annotations, adapting *translation validation* (TV)—a studied verification technique for program transformers—to the domain of program partitioning. A TV pass proves that the generated partition is behaviorally equivalent to the original program and satisfies its security requirements, and outputs an SMT-LIB certificate of correctness.

We used our tool to partition certain Linux shadow-utils scripts, annotated according to a Bell-LaPadula model inexpressible with existing C partitioners; and Security Desk, a 25 kLoC application composed of an embedded web application framework, sqlite3, and OpenCV for facial recognition. We find the number of annotations required is low (thirteen for Security Desk), and the auto-generated RPC code is a small fraction of the application's size. Post-partition verification uncovered subtle partitioning errors, including leakage of a *#pragma pack(1)* into the entire application.

II. Related Work

Program partitioners are generally *language-extending* or *annotation-driven*, although few are verified. We adopt the latter to avoid a new language toolchain.

Language-extending partitioners employ a *security type system*, e.g., the Decentralized Label Model [11]. Type inference assigns levels to expressions and partitioning divides the application into enclaves by level. Jif [12] extends Java with a security type system for program partitioning. Swift [13] leverages Jif for web applications to partition client and server code. Viaduct [14] includes security types and a partitioner.

Such systems have been augmented with mutable dependent types [15], [16], applied to cryptography [14], and integrated with hardware security [17], and can enforce security models such as Bell-LaPadula, Biba [18], or Clark-Wilson [19].

An annotation-driven approach works better with an existing language like C because developers only need to add security requirements, not rewrite everything. Mir [20] follows annotations to add runtime checks to reduce privileges for third-party libraries. SOAAP [21] annotations guide program refactoring for memory isolation. JOANA [22] observes annotations on Java data sinks to check for sensitive data leaks, but does not partition and its security model is often too coarse.

Annotation-driven C program partitioners support numerous application domains. PtrSplit [7] generates marshaling code for data with pointers; PM [8] enables security and performance tradeoffs. SeCage [4] splits applications into memory-isolated compartments. Trellis [5] uses a hierarchy of privilege levels for multi-user applications. Glamdring [6] partitions C for execution on Intel SGX. ProgramCutter [3] and SeCage use execution traces to shrink the privileged partition.

None of these tools support fine-grained policies. PrivTrans [2], PtrSplit, PM, Glamdring, and Huang et al. use a simple privileged/unprivileged model. Trellis and SeCage allow multiple levels, but Trellis prohibits declassification and SeCage only allows it to one specific component. Such models are inadequate for common policies such as multi-party Bell-LaPadula [10] where certain data may be shared with certain parties. *CAPO* supports fine-grained access control: arbitrary, unordered data sensitivity levels and select declassification.

Formally verifying a partitioner is desired but challenging [14]. Employing a proof assistant would require Herculean proofs of the security model, partitioner algorithm, and the semantics of a language like C. Our post-partition verification tool, *ParTV*, instead employs *translation validation* (TV): each generated program is compared to the original [23]. We compare LLVM IR of the generated partition to that of the original for behavioral equivalence and security compliance. Alive [24] uses TV to verify peephole optimizations. Tools such as CompCert [25] combine TV with a proof assistant. General program equivalence is undecidable [26]; the limited scope of *CAPO*'s transformations keep our problem tractable.

While researchers continue to generalize and improve the power of TV [27], none of the aforementioned partitioning tools employ it or any kind of post-partition verification. To our knowledge, ours is the first to perform formal verification.

```
double get_C() {                        double get_B() {
  static double c = 0.0;                  static double b = 0.0;
  _read_C(&c);                            _read_B(&b);
  return c;                               return b;
}                                       }
int main() {
  static double a = 0.0;
  for(int i = 0; i < 10; i++) {
    _read_A(&a);
    printf ("sum: %f\n", a + get_B() + get_C());
} }
```

Fig. 2. A program fragment that sums values from three sensitive data sources

```
#pragma label A     Orange
#pragma label B     Purple
#pragma label C     Green  [ Purple ]
#pragma label SUM Purple [ Orange ]

#pragma func Green [ Purple ] codeTaints: C \
  retTaints: TAG_RESPONSE_GET_C
double get_C() {
  #pragma taint C
  static double c = 0.0;
  _read_C(&c);
  return c;
}
#pragma func Purple codeTaints: B C \
  retTaints: SUM
double sum_BC() {
  #pragma taint B
  static double b = 0.0;
  _read_B(&b);
  return b + get_C();
}
#pragma func Purple [ Orange ] codeTaints: SUM \
  retTaints: TAG_RESPONSE_SUM_BC
double get_sum_BC() { return sum_BC(); }

int main() {
  #pragma taint A
  double a = 1.0;
  for(int i = 0; i < 10; i++) {
    _read_A(&a);
    printf ("sum: %f\n", a + get_sum_BC());
}}
```

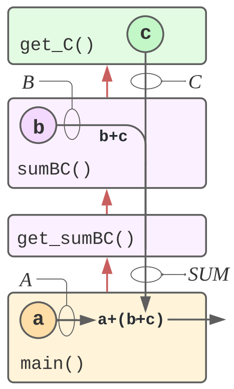

Fig. 3. A refactored, annotated version of Fig. 2

III. The Need For Fine-Grained Sharing Policies

Consider Fig. 2, an application fragment that models a computation that combines three pieces of sensitive data into a non-sensitive result. Its *main()* reads data from three sources, *_read_A()*, *_read_B()*, and *_read_C()*, and prints their sum. As written, an attacker able to gain control of a single *_read* function can call the others to access all sensitive data. A typical solution is to *partition* the application to isolate the data streams so that hijacking one does not yield the others.

Unfortunately, the security model in existing C partitioners is too coarse to handle this scenario. Most only allow data to be marked sensitive or not, so all three sources would have to be marked, along with *main()*, since it reads all three, and we are back to the original monolithic program.

It would be better to isolate the three streams at three security levels, say Orange for a, Purple for b, and Green for c, and prohibit them from communicating, but what to do about the sum operation in *main()*?

To sum three values, at least one party must be allowed to see two of the values. *CAPO*'s fine-grained access control policies allow us to express this without also exposing the third. Fig. 3 shows our solution, in which we moved the part of sum that accesses two values to a new function *sum_BC()*. To illustrate cross-domain calls, we also added *get_sum_BC()*, but this is optional.

The security policy expressed with pragmas in Fig. 3 is

(A) sensitive data *a* must remain at level Orange;
(B) sensitive data *b* must remain at level Purple;
(C) sensitive data *c* resides at level Green, and may be shared with Purple; and
(SUM) the sum of *b* and *c* may be shared with Orange.

Here, we have kept the three colored security levels with their three variables but introduced a fourth rule (SUM) that allows the partial sum to be passed to Orange.

First, consider *get_C()*. Rule (C) specifies data in Green may also be passed to Purple, written in Fig. 3 as *#pragma label C* (brackets indicate additional levels allowed access). The variable *c* is tainted C, indicating the rule its contents must follow. Like its data, the function *get_C()* is annotated as residing in Green and may be called from Purple.

The TAG_RESPONSE taint on the return data from *get_C()* refers to a synthesized rule that coerces the cross-enclave data from the level of the source to the destination without changing the data's shareability. Such labels do not affect the choice of partition; they make annotations consistent after partitioning.

Variable *b* is constrained to security level (Purple) with its own taint (B), but *sum_BC()* is tasked with declassifying data from both Green and Purple, so it is expressly allowed to handle data marked B or C and returns data tainted SUM, which resides in Purple but may be shared with Orange.

By design, we do not allow *sum_BC()* to be called outside Purple so we introduced *get_sum_BC()* to perform this role (different rules for *sum_BC()* could eliminate the need for *get_sum_BC()*). Like data tainted SUM, *get_sum_BC()* resides in Purple and may be called from Orange.

Finally, A restricts *a* to Orange. *main()* calls *get_sum_BC()*, which returns data that may be shared with Orange and Purple (SUM), so the taints in *main()* are consistent.

From this example, it may appear we require substantial annotation, but the burden only falls on sensitive data and cross-domain function calls, usually only a small portion of an application. For example, we found a 25 KLoC Security Desk application needed only thirteen annotations; see § VI.

IV. Annotation and Partitioning

Fig. 4 shows the grammar of annotations: *#pragma label* defines a label; *#pragma taint* marks a variable with a label; and *#pragma func* annotates a function with the constraints on how it may be called and what data it is allowed to send and receive. Annotations on functions and variables must immediately precede their definitions, as in Fig. 3.

```
annotation ::= #pragma label label level ( [ level+ ] )?
             | #pragma taint label
             | #pragma func level ( [ level+ ] )?
                 ( argTaints: label+ ( , label+ )* )?
                 ( codeTaints: label+ )?
                 ( retTaints: label+ )?
```

Fig. 4. The grammar of annotations: pragmas that define labels, taint variables, and specify function security constraints. *level* denotes the name of a security level (e.g., Orange); *label* is a taint name (e.g., SUM); *, $^?$, and $^+$ denote zero or more, zero or one, and one or more; Terminals include square brackets and commas, but not paretheses.

A label definition *#pragma label* defines a data taint: the security level of the data followed by an optional square-bracketed set of security levels to which the data can be declassified. E.g., in Fig. 3, data labeled *SUM* is pinned to Purple, but may also be declassified to Orange.

Preceding a variable declaration with *#pragma taint* prescribes the variable's taint. The constraint solver assigns consistent taints to the remaining unannotated variables.

Functions annotated with *#pragma func* are similarly pinned to a specific security level and may optionally be made callable from additional security levels. For example, in Fig. 3, *get_C()* is pinned to Green, but may also be called from Purple.

Annotated functions may also prescribe taints on input, output, and body data. The argTaints and codeTaints define sets of labels that the function accepts on its incoming arguments and in the function body; retTaints define those allowed on a function's return value. Annotated functions can thus declassify data: incoming argTaints and codeTaints are coerced to a retTaint, which may have different shareability. Taint coercing functions usually redact data. E.g., in Fig. 3, *sum_BC()* sums two tainted variables, hiding their values from the caller.

We designed our annotations to be sparse and highlight declassification, which is limited to annotated functions; unannotated functions cannot coerce taints or be called from other enclaves. Furthermore, since annotation is only mandatory on sensitive data and functions that coerce taints or are called cross-domain, the annotation burden only grows with the complexity of the security policy.

Our annotation grammar and semantics were inspired by the Bell-LaPadula security model [10] and CALIPSO [28]. By design, this model can express a far greater variety of policies than existing C partitioners, which usually only distinguish sensitive data from non-sensitive.

CAPO translates an annotated C program into the model in Fig. 5, applies the constraints from Fig. 6, and runs a constraint solver to assign enclaves to functions and variables. Our model only characterizes inter-function control- and data-flow; user-provided annotations give the security specification. We use *clang* to convert a C program into LLVM IR, build a program dependence graph (PDG) to identify inter-function control- and data-flow, then construct the Fig. 5 model.

In our model, "variables" (V) model data in a running program, but are not simply C variables. Ours are either global variables (G) or "locals" owned by a function that act primarily as anchor points for argument and return value data. We treat aggregate types as single variables.

Symbol	**Meaning**
Program Model	
$V,\ G \subseteq V$	Set of all variables; global variables
F	Set of all functions
$\mathrm{func} : V \to F \cup \{\mathrm{global}\}$	Function containing variable, if any
$\mathrm{calls} \subset F \times F$	Function call control flow edge
$\mathrm{argIn} \subset V \times V \times \mathbb{N}$	Function parameter passing
$\mathrm{argOut} \subset V \times V \times \mathbb{N}$	Pass-by-reference output
$\mathrm{returns} \subset V \times V$	Function return value edge
$\mathrm{globAcc} \subset F \times G$	Global variable access
Security Specification	
L	Set of security levels
E	Set of enclaves
$\mathrm{elevel} : E \to L$	Security level of each enclave
T	Set of labels
$\mathrm{tlevel} : T \to L$	Security level of each label
$\mathrm{shareable} : T \to 2^L$	Where data may be shared
$W \subseteq V$	User-annotated variables
$\mathrm{varTaint} : W \to T$	User-provided variable annotation
$A \subseteq F,\ U = F - A$	User-annotated and unannotated funcs.
$\mathrm{funLevel} : A \to L$	Level of annotated function
$\mathrm{callable} : A \to 2^L$	Can be called from these levels
$\mathrm{argTaints} : A \to (2^T)^*$	Labels allowed on each argument
$\mathrm{codeTaints} : A \to 2^T$	Labels allowed inside function
$\mathrm{retTaints} : A \to 2^T$	Labels allowed on return
Determined by the Constraint Solver	
$\mathrm{taint} : V \cup U \to T$	Label of variable or unannotated func.
$\mathrm{enclave} : G \cup F \to E$	Enclave of global variable or function

Fig. 5. An annotated C program is translated into this system model, then functions and global variables are assigned to enclaves following Fig. 6.

Helper $\mathrm{allTaints} : A \to 2^T$ collects a function's allowed labels:
$$\mathrm{allTaints}(a) = \mathrm{codeTaints}(a) \cup \mathrm{retTaints}(a) \cup \textstyle\bigcup_k \mathrm{argTaints}(a)_k$$

Labeled data is always shareable with its level:
$$\forall t \in T,\ \mathrm{tlevel}(t) \in \mathrm{shareable}(t)$$

Each function is callable at its level:
$$\forall a \in A,\ \mathrm{funLevel}(a) \in \mathrm{callable}(a)$$

Annotating a variable sets its label:
$$\forall v \in W,\ \mathrm{taint}(v) = \mathrm{varTaint}(v)$$

A global or unannotated function must reside in its labeled enclave:
$$\forall o \in G \cup U,\ \mathrm{tlevel}(\mathrm{taint}(o)) = \mathrm{elevel}(\mathrm{enclave}(o))$$

An annotated function must be in its labeled enclave:
$$\forall a \in A,\ \mathrm{funLevel}(a) = \mathrm{elevel}(\mathrm{enclave}(a))$$

Same label on unannotated func. vars.; annotated vars. constrained:
$$\forall v \in V,\ \mathrm{func}(v) \in U \Rightarrow \mathrm{taint}(v) = \mathrm{taint}(\mathrm{func}(v))$$
$$\forall v \in V,\ \mathrm{func}(v) \in A \Rightarrow \mathrm{taint}(v) \in \mathrm{allTaints}(\mathrm{func}(v))$$

Unannotated called intra-enclave; annotated prescribe levels:
$$\forall s \in F, d \in U,\ \mathrm{calls}(s,d) \Rightarrow \mathrm{enclave}(s) = \mathrm{enclave}(d)$$
$$\forall s \in F, d \in A,\ \mathrm{calls}(s,d) \Rightarrow \mathrm{tlevel}(\mathrm{taint}(s)) \in \mathrm{callable}(d)$$

Function arguments and return data must be shareable:
$$\forall s,d \in V, k \in \mathbb{N},\ \mathrm{argIn}(s,d,k) \vee \mathrm{argOut}(s,d,k) \vee$$
$$\mathrm{returns}(s,d) \Rightarrow \mathrm{tlevel}(\mathrm{taint}(d)) \in \mathrm{shareable}(\mathrm{taint}(s))$$

Unannotated access globals w/ same label; annotated have a policy:
$$\forall f \in U, g \in G,\ \mathrm{globAcc}(f,g) \Rightarrow \mathrm{taint}(g) = \mathrm{taint}(f)$$
$$\forall f \in A, g \in G,\ \mathrm{globAcc}(f,g) \Rightarrow \mathrm{taint}(g) \in \mathrm{allTaints}(f)$$

Unannotated may only communicate with those with the same label:
$$\forall s,d \in V, k \in \mathbb{N},\ \mathrm{argIn}(s,d,k) \wedge \mathrm{func}(d) \in U \Rightarrow \mathrm{taint}(s) = \mathrm{taint}(d)$$
$$\forall s,d \in V, k \in \mathbb{N},\ \mathrm{argOut}(s,d,k) \wedge \mathrm{func}(s) \in U \Rightarrow \mathrm{taint}(d) = \mathrm{taint}(s)$$
$$\forall s,d \in V,\ \mathrm{returns}(s,d) \wedge \mathrm{func}(s) \in U \Rightarrow \mathrm{taint}(d) = \mathrm{taint}(s)$$

Calling annotated within same enclave only with certain labels:
$$\forall s,d \in V, k \in \mathbb{N},\ \mathrm{enclave}(\mathrm{func}(s)) = \mathrm{enclave}(\mathrm{func}(d)) \Rightarrow$$
$$\mathrm{argIn}(s,d,k) \wedge \mathrm{func}(d) \in A \Rightarrow \mathrm{taint}(s) \in \mathrm{argTaints}(\mathrm{func}(d))_k$$
$$\mathrm{argOut}(s,d,k) \wedge \mathrm{func}(s) \in A \Rightarrow \mathrm{taint}(d) \in \mathrm{argTaints}(\mathrm{func}(s))_k$$
$$\mathrm{returns}(s,d) \wedge \mathrm{func}(s) \in A \Rightarrow \mathrm{taint}(d) \in \mathrm{retTaints}(\mathrm{func}(s))$$

Fig. 6. Constraints on the model

Overall, unannotated functions are placed in a single enclave and only operate on data with a single taint, annotated variables are placed in a suitable enclave and only accessed by functions with permission, and that data entering and leaving annotated functions comply with their policy.

Taint propagation is a non-trivial but decidable problem. We encode the constraint problem in the solver frontend Minizinc, which converts our constraints into an Integer Linear Programming (ILP) problem and dispatches it to the Gecode solver. We call this *conflict analysis* since taint propagation only fails as a result of a *conflicting* assignment of two different enclaves to a single function or variable.

Using a constraint solver allows us to optimize for secondary objectives: when multiple acceptable solutions exist, we instruct Minizinc to choose the partition with the fewest functions called cross-domain to reduce the amount of RPC wrapper code and improve performance.

Modern constraint solvers fly through ILP problems. Gecode solved our 550k-variable constraint problem for the 25 kLoC Security Desk application in under five minutes.

Using a constraint solver here also benefits a human auditor: the input to Minizinc is textual, declarative, and far easier to reason about than an imperative, algorithmic solution.

Conflict analysis produces a *topology* assigning functions and variables to hardware enclaves. *CAPO* then follows this to transform the code into an executable partition.

First, *CAPO* divides the source by enclaves, replaces cross-domain function calls with RPC wrapper calls, and adds headers for auto-generated code. For all but the main enclave, new main functions are created that initialize cross-domain communications before waiting for incoming RPC calls.

CAPO then collects data about cross-domain calls: the function signature, whether each argument is an input, output, or both, and dimensions of any arrays. These are inferred through heuristics; the developer must fill in any remaining information the heuristics could not infer.

CAPO then uses per-function data to generate (i) an RPC wrapper that serializes and sends a function's inputs then waits for and unmarshals a response back to the caller; (ii) a matching RPC handler that receives and passes the request to the actual function then serializes and sends output arguments and the return values (iii) a listener thread for each RPC handler; (iv) type definitions for each request and response structure; (v) serialization, deserialization, and print procedures for each request and response structure; (vi) Data Format Description Language specifications of the messages to allow deep content inspection; and (vii) configuration information for communication interfaces.

The partitioned and auto-generated code is compiled into one executable per enclave. The generated RPC code is small, typically around a hundred lines per cross-domain function.

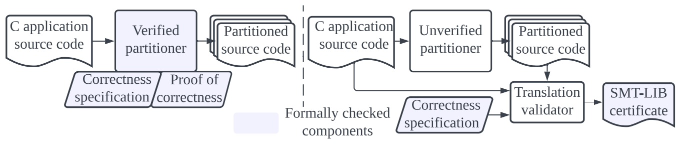

Fig. 7. Two verification techniques. (Left) Traditional proof-assisted verification (Right) ours: translation validation backed by a correctness certificate.

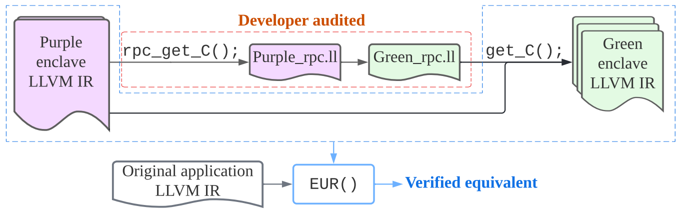

Fig. 8. Code EUR() verifies equivalent vs. what must be manually audited

V. Post-Partition Verification

Generating a secure partition that respects the developer's annotations and is semantically equivalent to the original is prone to subtle bugs. With security the goal of partitioning, correctness is paramount so we turn to formal verification.

Instead of the classical but time-consuming process of attempting to prove a program partitioner correct, we employ *post-partition verification* in which, for each run, we construct a proof that the particular output of the partitioner is behaviorally equivalent to the input (Fig. 7). Such *translation validation* (TV) [23] has checked compiler optimizations [29]; we appear to be the first to apply it to program partitioning.

Our translation validation pass *ParTV* checks the generated partition behaves like the original program, but instead of tackling arbitrary program equivalence, which is undecidable [26] and practically difficult [30], we adopt a stricter but easier-to-check policy that only accepts the modest changes of *CAPO*.

Definition 1 (Equivalence under renaming—EUR)**.** *Let O and P be two programs in the LLVM IR. Let M be a mapping between LLVM names, $M(n_1) = n_2$, and $P_{n\to M(n)}$ designate a version P in which every named IR element is renamed according to M. O and P are* equivalent under renaming *if there exists an M such that $main_{P_{n\to M(n)}} = main_O$, that is, when every named IR element in P is re-named according to M, the main() functions in both programs are deeply equal.*

EUR admits syntactic transformations such as function renaming but insists any instruction reachable from *main()* in one program is identical in the other. It largely ignores semantics to be checkable in linear time (Algorithm 1). EUR programs behave equivalently because they perform the same operations in the same order on corresponding data.

EUR does not hold for our partitioned programs because *CAPO* inserts communications code and produces multiple programs. We first effectively reverse partitioning (Fig. 8):

Definition 2 (Partition reconstruction)**.** *Let $P = \{p_1, p_2, \ldots, p_n\}$ be a partitioned program of n component programs communicating via RPC, all in LLVM IR. The* reconstruction *of P is $RP = \cup_{i=1}^{n} p_i^{rpc\_*\to *}$, the union of all global definitions in each program p_i, with every RPC invocation in p_i of a locally defined function in p_j replaced with an invocation of the local function in p_j.*

ParTV checks equivalence using $\text{EUR}(O, RP)$, which if true, implies O and its partition P can only differ in the behavior of the RPC initialization, send, and receive functions. *ParTV* does not analyze RPC wrapper code, but we plan to make it.

ParTV's EUR-checking algorithm, implemented in Haskell, reads the partitioned program's LLVM IR, translates it to native Haskell, and finds and replaces each RPC invocation to build *RP* from the set of components $p_1, \ldots, p_n$. The core of the algorithm, given in Algorithm 1, traverses the IR representations of O and RP in lockstep, beginning from the *main()* functions and recursively checking the equivalence of the two programs bottom-up. Certain nodes require special treatment e.g., basic blocks are matched under renaming.

Algorithm 1 Checking EUR on two LLVM IR programs

```
1:  function EUR($O, RP, M$)          ▷ Start at main() funcs., $M$ empty
2:      if $RP$.name then
3:          if $M[RP.\text{name}] \wedge M[RP.\text{name}] \neq O.\text{name}$ then
4:              return FALSE
5:          $M[RP.\text{name}] \leftarrow O.\text{name}$
6:      if $O.\text{constructor} \neq RP.\text{constructor}$ then
7:          return FALSE
8:      for $i \leftarrow 0, \text{LEN}(O.\text{fields})$ do
9:          EMITSMTEQUIV($O.\text{fields}[i], RP.\text{fields}[i]$)
10:         if $\neg\text{EUR}(O.\text{fields}[i], RP.\text{fields}[i], M)$ then
11:             return FALSE
12:     return TRUE
```

In addition to verifying behavioral equivalence, *ParTV* checks the partitioned program complies with the security policy. Since our solver-backed constraint model defines compliance, we run the generated partition *back through our conflict analyzer* to verify. This approach is sound because Algorithm 1 also check annotations, guaranteeing the original labels are preserved in the partition; conflict analysis therefore checks the same policy over equivalent code.

As with equivalence checking, we reassemble the partitioned programs by coupling RPC calls. The level-coercing TAG labels enable conflict analysis in the presence of RPC wrapper code.

Unlike computing a partition, levels of functions and variables are available during verification, so we populate the constraint model with them. This greatly simplifies constraint solving, which must only check whether the code has been partitioned consistently. A constraint solver allows most constraints to remain unmodified despite a different algorithm.

We generally trust our translation validator because it is small, simple, written in a safe language, and relies only on well-exercised libraries. A *ParTV* bug might bless an erroneous partition, but the tool and partitioner would require synergistic bugs. Nevertheless, the authors of Alive rewrote their translation validator in Lean and caught several bugs [31].

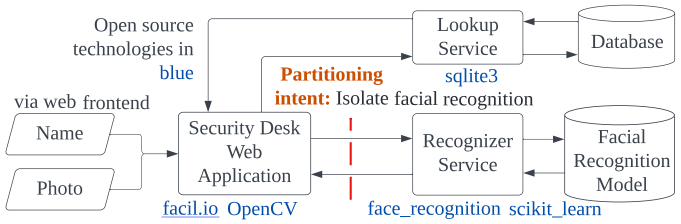
Open source technologies in blue
via web frontend
Name
Photo
Security Desk Web Application
facil.io OpenCV
Lookup Service
sqlite3
Database
Partitioning intent: Isolate facial recognition
Recognizer Service
Facial Recognition Model
face_recognition scikit_learn

Fig. 9. A model of the Security Desk application. A web frontend accepts a name and photo ID; facial recognition identifies a set of features, which are searched for in a sqlite3 database of user credentials.

Program	**SLoC**	**Globals**	**Target partition**
Toy example	37	8	Access only weighted avg
chsh	564	18	Protect password file
passwd	1168	47	Protect password file
useradd	2395	73	Protect password file
thttpd	11403	273	Separate *send()*, *send2()*
Security Desk	23654	1341	Isolate facial recognition

Fig. 10. A summary of the programs used for our evaluation.

Program	**Size**	**Annotations**		**RPC code**
	(SLoC)	**Function**	**Data**	**(SLoC)**
Toy example	37	1	3	1064
chsh	564	1	2	1054
passwd	1168	2	2	1111
useradd	2395	2	3	1176
thttpd	11403	3	2	1329
Security Desk	23654	7	6	1646

Fig. 11. The number of user-provided annotations to express a security policy

To further increase the trust in our translation validator (Algorithm 1), it emits a machine-checkable proof of each run that confirms its results meets an independently specified model of equivalence. *ParTV* emits an SMT-LIB file that contains models of the two programs it is comparing, assertions that corresponding parts of each program are equivalent (generated during the recursive check of the two programs), and a model of LLVM IR equivalence that effectively says objects are equivalent if their fields are equivalent (essentially, the EUR property). The first two components are specific to the programs being compared; the third is always the same. Furthermore, we generate the third using template metaprogramming because of its detailed yet mechanical nature.

Running the generated SMT-LIB file through the SMT solver/theorem prover z3 [32] provides further confirms that *ParTV* correctly concluded that its input programs were equivalent. But a bug in *ParTV* could produce a SMT-LIB theorem that is true yet does not faithfully reflect the input programs, something we have only manually checked. *ParTV* would benefit from being re-written in a proof assistant such a Coq.

VI. Evaluation

CAPO and *ParTV* are implemented as part of publicly available toolchain: https://github.com/gaps-closure. We evaluated them by annotating, refactoring, partitioning, and verifying a set of real-world C applications of varying size and complexity, with realistic security demands; these benchmarks are tabulated in Fig. 10 and further explained below.

In this section, we discuss the results of these experiments and the data we collected. All data was collected on an 8-core Intel i7 machine running Ubuntu 20.04. Our evaluation on all experiments was guided by the following questions: How many annotations are necessary to encode fine-grained security constraints, and how much additional code does the partitioner generate? Does the amount scale well with application size? (§ VI-A) How efficient is the partitioner? How does its performance scale to large programs? (§ VI-B) What insights can be gained, and what bugs detected, through the post-partition verification pass? (§ VI-C)

Fig. 10 lists the experiments we ran from small to large. We provide the total size in lines of code, the number of function and global variable definitions in the program, and a summary of the desired partition—that is, the specific data or functionality we wish to isolate in each enclave.

A toy example shows *CAPO*'s *baseline* code generation and annotation burden. Three smaller programs requiring authentication are from Linux's shadow-utils package to demonstrate the real-world applicability and lightweight annotation footprint. For stress testing and to demonstrate the variety of software and policies that *CAPO* and *ParTV* can handle, we secured the authentication functions of open-source HTTP server thttpd following PM [8]. We further partitioned Security Desk (Fig. 9), a custom embedded web application that uses OpenCV for facial recognition to match an image to a person's database credentials. We isolate the facial recognition from the rest of the application and control which data is shared.

Our experiments specify partitions that cannot be generated without fine-grained data sharing: what our tool is uniquely capable of generating (for the shadow-utils, we over-specify a security policy to exercise fine-grained access control, since the applications themselves are rather simple).

A. Annotation and Code Generation Footprint

Fig. 11 compares the size of the original programs against the number of annotations needed to express their security policies and the amount of auto-generated RPC wrapper code.

Security Desk and thttpd suggest the annotation burden remains small even for large applications, depending on the security policy as intended. The number of function and data annotations tend to be similar; this suggests there is usually some means of declassifying sensitive data to at least one party (e.g. every data annotation begets a function annotation).

The amount of auto-generated code similarly scales up with how many cross-domain calls need to be made, but it has a large constant value—no matter how simple the policy, upwards of 1000 lines of wrapper code are needed to initialize and mediate RPC communication. This makes code generation unwieldy for small programs, but much more reasonable for applications such as Security Desk and thttpd.

Experiment	Total time	Conflict Analysis	Code Gen.	Verification
Toy example	9.9 s	1.3	5.0	3.6
chsh	14	3.4	4.4	5.7
passwd	18	3.6	6.1	8.1
useradd	29	7.7	7.6	14
thttpd	1390	620	390	380
SecDesk	1560	630	430	500

Fig. 12. Total time taken by the partitioner in each phase.

B. Performance and Scalability

Fig. 12 lists the times taken by our partitioner and verifier at each stage of each experiment. We are pleased with the time for the solver-invoking conflict analyzer and verifier. Solvers are notoriously difficult to tune for performance.

For smaller benchmarks, code generation is a larger share of time spent; analysis and constraint solving dominate on larger ones. The rate of slowdown up to Security Desk's 25 minute runtime suggests that *CAPO* and *ParTV* are usable in practical settings, but will not scale indefinitely.

C. Verification Results

ParTV caught interesting bugs in *CAPO*. The most substantial was an un-closed *#pragma pack(1)* in auto-generated RPC header files. At best this incurs a performance hit, but it could have caused undefined behavior on some architectures, and yet would go undetected without either carefully examining auto-generated header files or the compiled LLVM IR.

Other problems we found include an incorrect conversion from zero-argument functions to variadic functions and inconsistent string outputs from applications that printed *__FILE__*.

Our experience with post-partition verification makes a strong argument for leveraging formal methods in program partitioning. Source code transformation and code generation is often mundane, and burdensome at scale, but it cannot be safely automated until it is backed by correctness guarantees.

VII. Conclusion

Automatically partitioning C programs to safeguard sensitive data remains valuable because C makes it easy to introduce vulnerabilities yet sensitive programs continue to be written. But C program partitioners still require work to support fine-grained access control, mandatory redaction of sensitive data, and data integrity constraints. Furthermore, to truly gain the confidence of developers, C partitioners need the strong correctness guarantees of formal verification.

Our annotation-driven C program partitioner expands the security expressiveness of existing work by allowing the developer to write annotations that express fine-grained access control among multiple parties in a computation. Ours is the first program partitioner to employ post-partition verification, which provides a per-partition guarantee of correctness and security compliance. Our experiments show we can quickly partition large, realistic applications with security requirements inexpressible by current partitioners and that our post-partition verification pass is scale and catch partitioner bugs.

References

[1] "Microsoft office CVE-2018-8412 privilege escalation vulnerability," Available from MITRE, CVE-ID CVE-2018-8412, Mar. 2018.

[2] D. Brumley *et al.*, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Sym.*, vol. 57, no. 72, 2004.

[3] Y. Wu, J. othersSun, Y. Liu, and J. S. Dong, "Automatically partition software into least privilege components using dynamic data dependency analysis," in *Proc. ASE*, 2013, pp. 323–333.

[4] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia, "Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation," in *Proc. Comp. Comm. Security*, 2015, pp. 1607–1619.

[5] A. Mambretti *et al.*, "Trellis: Privilege separation for multi-user applications made easy," in *Attacks, Intrs., and Defenses*, 2016, pp. 437–456.

[6] J. Lind *et al.*, "Glamdring: Automatic application partitioning for Intel SGX," in *USENIX Annual Technical Conf.*, 2017, pp. 285–298.

[7] S. Liu, G. Tan, and T. Jaeger, "PtrSplit: Supporting general pointers in automatic program partitioning," in *Proc. CCS*, 2017, pp. 2359–2371.

[8] S. Liu *et al.*, "Program-mandering: Quantitative privilege separation," in *Proc. Comp. Comm. Security*, 2019, pp. 1023–1040.

[9] Z. Huang, T. Jaeger, and G. Tan, "Fine-grained program partitioning for security," in *Proc. Euro. Workshop Systems Security*, 2021, pp. 21–26.

[10] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," MITRE Corp., Bedford, MA, Tech. Rep., 1973.

[11] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM TOSEM*, vol. 9, no. 4, pp. 410–442, 2000.

[12] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Secure program partitioning," *ACM TCS*, vol. 20, no. 3, pp. 283–328, Aug. 2002.

[13] S. Chong *et al.*, "Building secure web applications with automatic partitioning," *Comm. ACM*, vol. 52, no. 2, pp. 79–87, 2009.

[14] C. Acay *et al.*, "Viaduct: an extensible, optimizing compiler for secure distributed programs," in *Proc. PLDI*, 2021, pp. 740–755.

[15] A. Ferraiuolo *et al.*, "Secure information flow verification with mutable dependent types," in *Proc. DAC*, 2017, pp. 1–6.

[16] S. Ghosal *et al.*, "Static security certification of programs via dynamic labelling." in *ICETE (2)*, 2018, pp. 400–411.

[17] A. Oak *et al.*, "Language support for secure software development with enclaves," in *Proc. CSF*, 2021, pp. 1–16.

[18] K. J. Biba, "Integrity considerations for secure computer systems," MITRE Corp., Bedford, MA, Tech. Rep., 1977.

[19] D. D. Clark *et al.*, "A comparison of commercial and military computer security policies," in *Security & Privacy*, 1987, pp. 184–184.

[20] N. Vasilakis *et al.*, "MIR: Automated quantifiable privilege reduction against dynamic library compromise in JavaScript," *arXiv:2011.00253*, 2020.

[21] K. Gudka *et al.*, "Clean application compartmentalization with SOAAP," in *Proc. Comp. Comm. Security*, 2015, pp. 1016–1031.

[22] G. Snelting *et al.*, "Checking probabilistic noninterference using JOANA," *Info. Tech.*, vol. 56, no. 6, pp. 280–287, 2014.

[23] B. Goldberg *et al.*, "Into the loops: Practical issues in translation validation for opt. comp." *ENTCS*, vol. 132, no. 1, pp. 53–71, 2005.

[24] N. P. Lopes *et al.*, "Provably correct peephole optimizations with Alive," in *Proc. PLDI*, 2015, pp. 22–32.

[25] X. Leroy *et al.*, "CompCert—A formally verified optimizing compiler," in *ERTS*, 2016.

[26] R. Goldblatt and M. Jackson, "Well-structured program equivalence is highly undecidable," *ACM TCL*, vol. 13, no. 3, Aug. 2012.

[27] S. Kundu *et al.*, "Proving optimizations correct using parameterized program equivalence," *Sigplan Not.*, vol. 44, no. 6, pp. 327–337, 2009.

[28] R. Atkinson and M. StJohns, "Common Architecture Label IPv6 Security Option (CALIPSO)," RFC 5570, Jul. 2009.

[29] N. P. Lopes *et al.*, "Practical verification of peephole optimizations with Alive," *CACM*, vol. 61, no. 2, pp. 84–91, Feb. 2018.

[30] S. K. Lahiri, A. Murawski, O. Strichman, and M. Ulbrich, "Program equivalence," in *Dagstuhl Reports*, vol. 8, no. 4. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[31] J. Lee, C.-K. Hur, and N. P. Lopes, "Aliveinlean: A verified LLVM peephole optimization verifier," in *Proc. CAV*, Jul. 2019.

[32] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.