Secure Delivery of Program Properties through Optimizing Compilation

Son Tuan Vu
Sorbonne Université
CNRS, LIP6
France

Karine Heydemann
Sorbonne Université
CNRS, LIP6
France

Arnaud de Grandmaison
Arm
France

Albert Cohen
Google
France

Abstract

Annotations and assertions capturing static program properties are ubiquitous, from robust software engineering to safety-critical or secure code. These may be functional or non-functional properties of control and data flow, memory usage, I/O and real time. We propose an approach to encode, translate, and preserve the semantics of both functional and non-functional properties along the optimizing compilation of C to machine code. The approach involves (1) capturing and translating source-level properties through lowering passes and intermediate representations, such that data and control flow optimizations will preserve their consistency with the transformed program, and (2) carrying properties and their translation as debug information down to machine code. Our experiments using LLVM validate the soundness, expressiveness and efficiency of the approach, considering a reference suite of functional properties as well as established security properties and applications hardened against side-channel attacks.

CCS Concepts. • **Software and its engineering → Compilers**.

Keywords. Annotation, Security, Compiler, Optimization, LLVM

ACM Reference Format:
Son Tuan Vu, Karine Heydemann, Arnaud de Grandmaison, and Albert Cohen. 2020. Secure Delivery of Program Properties through Optimizing Compilation. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377555.3377897

CC '20, February 22–23, 2020, San Diego, CA, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7120-9/20/02.
https://doi.org/10.1145/3377555.3377897

1 Introduction

Research in software engineering and computer security has led to new kinds of tools for analyzing code for bugs and security vulnerabilities. A widely used approach is to perform static analysis on source code in order to determine whether the program can reach a bad state. One common class of source-level analysis consists in annotating the source code with logic properties then checking the program against its specification [20, 26, 39]. Logic property annotations provide a powerful way of expressing the program specification, i.e., assumptions about the soundness of the program and its execution; we refer to these as *functional properties.*

However, source-code analyses is not sufficient for several reasons. The main one is the lack of *full abstraction* in languages and compilers. It manifests practically as the WYSINWYX phenomenon [10] in software engineering, program verification and testing: the observable mismatch between the behavior intended by the programmer (the source code) and what is actually executed by the processor (the executable) due to compiler optimizations. In addition, analyzing machine code is sometimes mandatory due to the essential role played by machine-level details such as memory layout, register mapping, code placement, etc.

As a result, binary analysis and verification tools have been proposed to assess the properties of machine code. These properties may be inserted manually at binary level [15, 32], which is tedious and error-prone. Moreover, relying on code instrumentation evaluating functional properties at run-time [29] is only acceptable for evaluation and test. It would be most beneficial to have an automatic way to provide binary analysis tools with direct access to the source-level properties.

One natural direction is to annotate the source program with functional properties, but more issues arise: preserving these through the compilation flow, their consistency with the code undergoing transformations and optimizations, and their embedding into the compiled binary without interfering with the executable code itself. Unfortunately compilers only care about functional correctness and I/O (viewed as an opaque functional effect). *Compilers have no notion of the link between the extra properties and the code they refer to;*

they have no means to constrain transformations to preserve this link or to update the properties to adjust to any code transformation. Optimizing compilers tend to remove everything that is not behaviorally observable. Source code annotations implementing properties are obviously not supposed to modify the program logic, hence compilers do not know how to maintain such information through the compilation flow [36]. Besides, variables referenced in annotations may also be affected by compiler optimizations: e.g., an unused variable may be optimized out, invalidating the semantics of the functional property. It follows that we first have to define a notion of *functional property preservation*, capturing the logical property itself, its link with the program semantics, and its preservation across transformations. To the best of our knowledge, there is no optimizing compiler, taking as input a source program annotated with functional properties, capable of preserving these annotations through a range of agressive optimization passes, and propagating them all the way to machine code.

Furthermore, many properties of interest are related to the *machine state*: they are not naturally captured as logical expressions of the source program variables. We refer to these as *non-functional* since they cannot be directly expressed as functions of source-level denotations or state. For example, in the context of secure code countermeasures such as control flow integrity [2], the properties take the form of a careful encoding of the machine code's branching behavior as functional properties of the source program. In general, the link between such smart functional encoding and the related assumptions on the execution cannot be made explict, given the semantic abstraction gap between source and machine code. These encodings are also as diverse as the (approximate) models of the machine state and transitions instrumented by control flow integrity countermeasures [16]. We are thus looking for a generic solution rather than property- or optimization-specific ones.

In brief, this paper makes the following scientific contributions: capturing non-functional security properties as functional properties of the source program (Section 6.3); propagating functional properties down to machine code without hampering compiler optimizations (Section 3 and Section 4); an LLVM-based implementation with virtually no modification of the optimization passes (Section 5); the end-to-end validation on both generic and secure code properties (Section 6). We would like to emphasize that preserving properties through the compilation flow has been a long-standing open issue in security engineering. This explains the emphasis on security in the following examples and experimental evaluation.

2 Context and Related Work

There is a large body of research and engineering on secure compilation [1, 3, 4, 22, 31]. Program transformations are meant to enforce some notion of behavioral equivalence with respect to the capabilities of an attacker, from contextual equivalence [3]—the compiler is then called fully abstract—to more specific properties such as isolation and safety guarantees enforced by the the source language's type system [18, 51], and to hyperproperties not directly captured in terms of behavioral equivalence [5]. This major trend benefits from and participates to the advances in formally verified compilation, as illustrated by the CompCert project [43, 44]. It is complementary to ours: the properties we care about are finer grained. In the future, one may wish to extend one such mechanically proven framework to express and preserve our properties in a more principled fashion, with a higher level of confidence.

Besides, many program annotations are solely meant to be exploited at source level for static analysis; see e.g. *ANSI/ISO C Specification Language* (ACSL) and *Framework for Modular Analysis of C programs* (Frama-C) for a survey and reference [13, 20]. We focus the rest of the discussion on properties that are meant to be carried along the compilation flow.

Properties carried as program annotations may significantly enhance the quality and the effectiveness of optimization passes. Dynamic properties include information from past executions to influence optimization heuristics, such as *profile-guided optimization.* Static properties generally take the form of logical invariants represented as boolean-valued expressions. However, through the compilation process, the program is transformed and lowered to *intermediate representations* (IRs) on which optimizations operate. When the property is functional and captured natively as a source language expression—like the C `assert`—any correct compiler is meant to preserve its consistency as a dynamically evaluated expression. But when the static expression itself needs to be carried to an IR for the purpose of conducting analysis or optimizations, or when the expression is an annotations external to the language such as ACSL [13], the property must be taken into account in program transformations as a semantic preservation constraint, and also as a subject of transformations itself to preserve its consistency with the transformed code. For example, best-effort statistical methods have been devised to carry control flow information through transformations updating branch probabilities [34, 56], but these are limited to run-time values and non-functional quantitative metrics. In the case of static properties, a methodology has been proposed [50] to augment each optimization pass with a witness relation between the source and the target programs, guaranteeing the property is correctly propagated for that optimization. Compared to our approach, this methodology is rather intrusive and limited to the specific passes it enhances, as demonstrated in a proof of concept LLVM-based implementation [30, 49, 62]. Instead of having to define a witness generator for each optimization pass, we leverage one fundamental property already enforced by every compiler transformation: the preservation of reaching definitions, a.k.a. as use-def chains.

For hard real-time systems, it is not only crucial that the software computes the correct result, but also that this happens in a timely manner. One needs to determine the *Worst-Case Execution Time* (WCET) of critical software parts to get an embedded system certified. WCET estimation takes place on machine code, where the processor micro-architecture can be modeled. Detailed control flow information, typically loop trip counts, infeasible paths, program points that are mutually exclusive during the same run, is needed to calculate WCET as accurately as possible; this information takes the form of source code annotations [7, 11]. Preserving these annotations' consistency with the code being compiled is a correctness requirement. For this purpose, CompCert introduces a builtin function modeled as a call to an external function producing an observable event, without emitting it as machine code [57]. While CompCert events capture values and memory locations individually, we address the preservation of (variable, value) and (memory_location, value) pairs and their association with a specific program point. We take up the additional challenge of embedding such mechanisms in a widely deployed compiler with minimal changes. Other work encode flow information using inline assembly [24] outside the IR [27, 54], or using IR extensions and external transformations to update loop trip count information [46]. These approaches incur significant changes to optimization passes: they all come with a set of rules to transform control flow information along code transformations.

Finally, some security properties do not fall in the functional category listed at the beginning of this section. This is the case of properties associated with software protections: the effectiveness of these protections relies on the implicit consistency of their functional expression with assumptions of the underlying control flow or machine state. For example, control flow integrity protects secure code from branch or call hijacking in compromised software or in the presence of physical attacks [2]. Such protections take many forms, including duplicated execution or tracking the effective flow in auxiliary counters, taint detection variables, trackers, etc. [16]. Compiler optimizations transform the control flow without any knowledge of the implicit link between the protection code and what underlying mechanisms it is meant to track. One key motivation for our work is to make this link explicit through program properties and making sure transformations preserve this link. In addition, there is a growing need for tools to insert such protections at compilation-time [36, 48, 55], but these techniques all face the problem of protecting their hardening instrumentation from downstream transformations. As a result, the current practice is that security engineers have to analyze the compilation pipeline and disable the optimization passes harming the protections [36, 55]. Obviously, the problem might be avoided by applying countermeasures as lately as possible in the compilation pipeline [12, 21], but this is not always desirable or even possible, since optimizations in the compilation pipeline might remove essential information needed for crafting the protection code itself. Our approach allows to express and propagate the properties of the protection code down to the binary.

3 Problem Statement and Definitions

Let us start with the motivating example in Listing 1. Masking is one of the most widespread countermeasures against power analysis attacks on cryptographic algorithms [8, 17, 33, 37]. The idea is to mask the intermediate computations with random noise and to unmask the result at the end. To maintain the statistical independence of intermediate values from the secret, some re-masking can be necessary at some point [35]. This is illustrated in Listing 1.a. A secret key, `mk`, already masked with mask `m`, is re-masked with a random value `r` (line 7), so that previous mask `m` can safely be removed from `mk` (line 9). Re-masking must take place before the removal of the previous mask to avoid exposing the unmasked secret key value. In this case, the wanted property can be expressed with a boolean expression over program variables at a specific point (line 8). Although this property holds at the source level, compiler optimizations can produce an unsafe, semantically equivalent program illustrated at source-level in Listing 1.b: the unmasking operation now takes place before re-masking, making the program vulnerable to power analysis attacks. The property `tmp == mk ^ r` does not hold anymore: the variable `tmp` is optimized out, and `mk` does not contain the expected value it had on line 8 of the original program. The compiler could also reorder instructions, anticipating the erasure of the random value from line 12 to line 8, which would also invalidate the expected value of `r` in the property.

```
1  /// a. Original program.
2  int r, tmp, mk /* masked key */, m /* mask */;
3  r = rand();
4  ... // do something
5
6  r = rand(); // new random value
7  tmp = mk ^ r;
8  PROPERTY(tmp == mk ^ r)
9  mk = tmp ^ m;
10 ... // do more things
11
12 r = 0; // new definition used in follow-up code
1  /// b. After instruction combining
2  ...
3  r = rand(); // new random value
4  mk = mk ^ m ^ r;
5  PROPERTY(tmp == mk ^ r) // invalid property
```

Listing 1. Motivating example.

This example illustrates the difficulty of preserving and propagating program properties down to executable machine code, especially in the presence of compiler optimizations. Our solution involves defining a notion of partial state associated with a program property, from which we derive the notion of functional property preservation.

We use the informal semantics of C (ISO/IEC 9899:2011 [38]) as a reference. As a simplifying assumption, we only consider deterministic, sequential C programs with well defined behavior, avoiding cases where the compiler may take advantage of undefined behavior to trigger optimizations. This assumption is consistent with widespread coding standards for secure code. Given our interest in the compilation and its effects on program properties, we also need to agree on the semantics of every *Intermediate Representation* (IR) in the flow. We focus on the clang/LLVM framework and select a semantics of LLVM IR that is friendly to aggressive optimizations [42]. By doing so, we do not limit ourselves to the optimizations currently available in LLVM or any C compiler, but rely on the more fundamental assumption that optimizations rely on static data-flow properties to validate the correctness of a given program transformation. We believe our problem statement and solutions could be adapted to other source languages and compilers with only minor modifications. In the following, the program may refer to any step of the compilation flow: source, IR or machine code level.

Definition 1 (Program state). *A program state is defined by*

- *a distinguished value of the program counter π denoting a program point;*
- *a finite set V of $(variable, value)$ pairs for all program-defined variables at π;*
- *a large but finite set M of $(memory\_location, value)$ pairs stored in main memory at π.*

Definition 2 (Program partial state). *A program partial state is a subset of a (complete) program state. It is defined by a program counter π, a set $V' \subseteq V$ and a set $M' \subseteq M$. A program partial state holds the $(variable, value)$ pairs in V' and the $(memory\_location, value)$ pairs in M', both at π.*

Definition 3 (Functional property). *A* functional property *specifies the program behavior by exclusively referencing its variables and memory locations. It takes the form of a pair $(Formula, ObsPt)$, where Formula is a propositional logic formula expressing the behavioral properties of the program; ObsPt denotes the program point called* observation point *at which property Formula is expected to hold.*

Given a functional property $(Formula, ObsPt)$, we call *ObsVar* and *ObsMem* the sets of all *observed variables* and *observed memory locations* occurring in *Formula*. The functional property defines a *partial state* containing the (variable, value) pairs and (memory_location,value) pairs of all observed variables and observed memory locations at observation point *ObsPt*.

Considering Listing 1.a again, at line 8 the complete program state would include a pair for variables `r`, `tmp`, `mk`, and `m`; whereas the program partial state defined by the property `tmp == mk ^ r` does not contain a pair for `m`.

A functional property *acts as a barrier at ObsPt for all memory accesses to locations in ObsMem and definitions of variables in $ObsVar$*: no definition of a variable and no access to a memory location observed by *Formula* may be moved across *ObsPt* through program transformations.

It is worth noting that Definition 3 generalizes to invariant properties of a control flow region: the latter may be considered as a set of functional properties $(Formula, ObsPt_i)$ for all program points $ObsPt_i$ belonging to the region.

Definition 4 (Observation trace). *Given a finite set of functional properties FP, an* observation trace *is a finite sequence of program* partial *states defined by the properties in FP.*[1]

Definition 5 (Functional property preservation). *Given a program P and a transformation τ that applies to P producing a semantically equivalent program P'—i.e. with the same I/O behavior—τ is said to* preserve all functional properties *in P if the observation traces produced by P and P', given the same input, are equal—i.e. they have the same sequence of partial states.*[2]

Note that Definition 5 does *not* define a notion of *program point preservation.* Only pairs $(Formula, ObsPt)$ are preserved, as defined by the associated partial states in the observation traces of the original and transformed programs. Still, a program transformation τ maps functional properties $(Formula_i, ObsPt_i)$ in a program P into functional properties $(Formula'_j, ObsPt'_j)$ in a program P'. One logic formula $Formula_i$ in P may correspond to one or more formulas in P' (for instance, when $ObsPt_i$ is inside a loop and transformation τ is loop unrolling, which would duplicate the loop body) up to the renaming of its variables and the rearrangement of its memory locations, as long as evaluating the formula at observation point $ObsPt'_j$ results in the same value at the same position in the trace. This applies to all levels of program representation, from source to machine code. For example, a source variable may be renamed into several SSA variants according to the control and data flow of the variable, and a variable may be promoted to a register later in the backend compilation flow.

4 Functional Properties and Optimizations

Optimizations are free to remove variables, reorder or remove instructions as long as they do not change the observable behavior (cf. Section 1 and Listing 1). Preserving functional properties according to Definition 5 implies (1) preventing the compiler from removing any observed variable or memory location, (2) maintaining the correspondence between the observed variables or memory locations and their values at the above observation points, and (3) blocking the movement of memory accesses and all variable definitions across the functional property observing them (the barrier effect of functional properties). This section describes our solution.

Production compilers such as gcc [59] or LLVM [41] generally have an IR in *Static Single Assignment* (SSA) form, and we first describe our solution tailored for the SSA properties.

[1]Again, we restrict ourselves to sequential, deterministic programs.
[2]This is a rather conservative definition as it does not allow any reordering of partial states. We consider relaxing it in the future, but prefer a simpler definition for now, emphasizing the difference with related notions of semantics preservation proposed in secure compilation and debug-friendly compilation.

The SSA form ensures that any program point has an unique reaching definition for live SSA variables. As a result, a source variable v is represented by multiple SSA variables $V = \{v_1, \ldots, v_n\}$. Note that multiple SSA variables corresponding to the same source variable may be alive at a given program point.

A natural approach to propagate program functional properties consists in tracking reaching definitions of all observed variables across IRs. This would imply finding, after each program transformation, an observation point at which all reaching definitions of the observed variables exist. However, the existence of such program point is not always guaranteed by compiler optimizations, and worse, a given reaching definition might be optimized away, as illustrated in Listing 1.b. Hence we propose an approach that propagates *by construction* those defined in the source program or any IR.

At a given observation point, we need to ensure the correct values of the observed variables and the values stored at the observed memory locations. For memory locations, one may enforce these constraints by inserting a compiler fence, i.e. an inline assembly instruction with a memory side-effect, see e.g. Listing 2.e. Enforcing the same constraints for the observed variables implies preserving the reaching definitions of every observed variable for the observation point: we have to make sure that these definitions exist and take place before the observation point. This is generally not guaranteed by optimizing compilers since optimizations such as code motion do not know about such a property-specific constraint.

As an example, consider the property `tmp == mk ^ r` from Listing 1.a. We illustrate the concern in Figure 1a, which shows the code example in SSA form: the source variable `r` is translated into 3 different SSA variables `%r1`, `%r2` and `%r3`, where `%r2` is the reaching definition of `r` for the observation point of the property. Similarly, variable `mk` from the source program is represented by 2 different SSA variables `%mk1` and `%mk2`, while `tmp` is now `%tmp1`. Variables in the property have also been renamed accordingly into `%tmp1`, `%mk1` and `%r2`. Compiler optimizations can remove the definition `%tmp1` or move down the observed variable definitions according to any one of the arrows, corrupting the program partial state defined by the property.

The key idea is to tie together the reaching definitions of observed variables and the order of memory accesses. We achieve this by inserting a compiler fence and also new *artificial definitions* around the observation point. These artificial definitions must be declared as *opaque, side-effecting functions* that the compiler cannot analyze, such that their relative ordering w.r.t. the compiler fence cannot be altered by the compiler. Such artificial definitions split the live ranges of a variable across the compiler fence. As a result, program transformations respecting reaching definitions cannot move the live range of any observed variable across the compiler fence. It guarantees the presence and the correct value of observed variables at a given observation point.

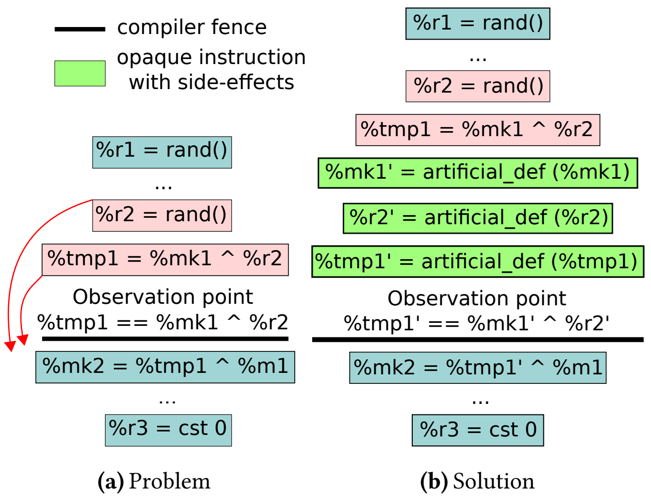


Figure 1. Preserving program partial state in SSA

Algorithm 1. Artificial definitions in SSA

```
input : Properties, list of functional properties
1 for (Formula, ObsPt) ∈ Properties do
2     ObsVar ← variables of Formula;
3     CF ← compiler fence associated with ObsPt;
4     for OV ∈ ObsVar do
5         RD ← reaching definition of OV for ObsPt;
6         insertArtificialDefinition(RD, CF, Before);
```

Algorithm 1 simply inserts artificial definitions before the compiler fence for every element in the set *ObsVar* of observed variables of the functional property. Function *insertArtificialDefinition*(D_v, *Inst*, Before|After) creates an artificial definition of SSA variable v' from D_v the definition of an SSA variable v, it inserts it before (resp. after) instruction *Inst* and replaces all uses of v subsequent to the insertion point by v'. Applying the algorithm to Listing 1.a yields the code in Figure 1b: all observed variables in the property (`%r2`, `%tmp1` and `%mk1`) have been renamed (respectively `%r2'`, `%tmp1'` and `%mk1'`). Artificial definitions opaquely define new values for these observed variables. The compiler cannot make assumptions about these values and is forced to assign concrete locations to hold them. As a result, artificial definitions do prevent the elimination of the observed variables or their replacement by constants at the observation point (they may still be optimized in the rest of the code).

5 Putting It To Work

Let us now survey the implementation of functional properties preservation in a state of the art optimizing compiler.

5.1 Functional Properties in Source Code

We capture functional properties as source program annotations using the following attribute syntax:[3]

```
l: __attribute__((annotate("str")));
```

The label `l` denotes the observation point and the functional property is expressed by a logical formula `str`. Logical formulas use the C language syntax for boolean expressions, which is also a subset of the ACSL specification language [13].

5.2 Functional Properties in Machine Code

When attaching functional properties to machine code, one needs to make sure binary analysis tools are capable of extracting the corresponding representation of the observed variables and memory locations.

Debug information is generated by the compiler for the purpose of communicating source location, type and variable information to the debugger. It already provides machine code with a detailed description of source-level variables as well as of memory locations. It is thus very reasonable to extend the debug information to represent functional properties in the binary: the formulas expressing the functional properties are propagated and emitted into the debug section of the binary, while the binary representations of observation points and observed variables and memory locations are already provided by the standard debug information. It is common knowledge that debug information is a second-class citizen of compiler validation, and may not be accurate in the presence of aggressive optimization. A fortunate side-effect of inserting artificial definitions is to prevent most optimization passes from harming the observed variables' debug information. We still had to fix a few critical remaining bugs, and filed bug reports for others (cf. Section 6.3.3). As an immediate benefit of leveraging (accurate) debug information, compiler passes do not have to worry about renaming variables observed in functional properties: debug information takes care of tracking the mapping of source to IR variables, down to machine code registers and stack locations.

We use the *Debugging With Attributed Record Formats* (DWARF) [19] debug information format that provides an easily extensible description of how a program is translated into executable code. DWARF uses a series of descriptive entities called *Debugging Information Entry* (DIE) to capture the low-level representation of a source program. A DIE has a *tag*, which specifies what it represents and a list of *attributes* which fill in the tag-dependent information and further describes the entity. Hence, a DIE, or a group of DIEs together, provide a description of a corresponding entity in the source program, be it a type, a function, a parameter, a variable or a label.

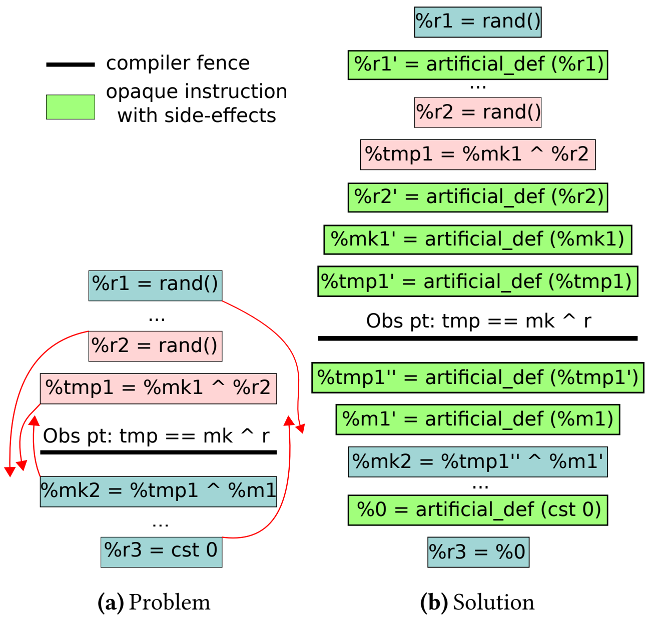
We introduce new tags and attributes to represent source-level functional properties. A property is represented by a DIE which contains the formula, the machine code address corresponding to the observation point and references to the DIEs representing the observed variables and memory locations.

[3]This syntax is not part of ISO C but it is accepted by gcc-compatible compilers, including LLVM. Labeled attributes are better semantically tied to the IR control flow than pragmas.

5.3 Multiple Definitions and Debug Information

So far we only considered SSA form IR for pedagogical reason. However, the source program, machine code and low level IR are typically not in SSA. Our method operates on all these representations. On non-SSA form programs, multiple definitions may exist for a given observed variable v. All of these have to be considered in the algorithm. Furthermore, the algorithm needs to block any assignment to v that *does not reach* the observation point from becoming the reaching definition of v.

Yet this is not the end of the story. As it stands today, debug information can only provide a *single* value for every *source* variable at a given line of code. When multiple live ranges corresponding to the same source variable overlap, only the most recent one is stored in the corresponding DIE. In the same spirit, when multiple live ranges get permuted, debug information only refers to the last definition, irrespectively of the initial program order. Such behavior may be observed after variable renaming and live range splitting, followed by code motion. This simplification is consistent with the common usage of debug information in debuggers, but it conflicts with our application to functional property preservation. This is the second reason why we have to forbid any transformation from reordering definitions of *different occurrences of the same source variable*, as soon as one of these occurrences is observed by a functional property.


compiler fence
opaque instruction with side-effects
%r1 = rand()
...
%r2 = rand()
%tmp1 = %mk1 ^ %r2
Obs pt: tmp == mk ^ r
%mk2 = %tmp1 ^ %m1
...
%r3 = cst 0
%r1 = rand()
%r1' = artificial_def (%r1)
...
%r2 = rand()
%tmp1 = %mk1 ^ %r2
%r2' = artificial_def (%r2)
%mk1' = artificial_def (%mk1)
%tmp1' = artificial_def (%tmp1)
Obs pt: tmp == mk ^ r
%tmp1'' = artificial_def (%tmp1')
%m1' = artificial_def (%m1)
%mk2 = %tmp1'' ^ %m1'
...
%0 = artificial_def (cst 0)
%r3 = %0

(a) Problem **(b)** Solution

Figure 2. Property preservation and debug information

As illustrated in Figure 2a, two SSA variables `%r1` and `%r2` stemming from the same source variable `r` and defined before the functional property will have to remain there (before the observation point); conversely, SSA variables like `%r3` corresponding to later live ranges of `r` can only be defined after the functional property. As a sufficient condition to enforce this additional constraint, as soon as one variable is observed by the property, the algorithm *preserves the relative order* of "sibling" variable definitions stemming from the same source variable,[4] as illustrated in Figure 2b. This may sound as overkill but it is currently necessary to prevent non-reaching definitions from clobbering the variable observed by the functional property. On the bright side, one advantage of this solution is that *variables in functional properties do not need to be renamed* along the compilation flow; this removes the burden of modifying many compilation passes and also helps the interpretation of program binary analysis.

Algorithm 2. Artificial definitions from debug info

```
input :   Properties, list of functional properties
1  for (Formula,ObsPt) ∈ Properties do
2      ObsVar ← variables of Formula;
3      CF ← compiler fence associated with ObsPt;
4      for OV ∈ ObsVar do
5          ReachDefs ← set of reaching definitions of OV for
6          ObsPt;
7          PriorDefs ← set of definitions of OV preceding any
8          reaching definition in ReachDefs in program order;
9          for PD ∈ PriorDefs do
10             insertArtificialDefinition(PD,PD,After);
11         for RD ∈ ReachDefs do
12             insertArtificialDefinition(RD,CF,Before);
13         NextDefs ← set of definitions of OV following
14         ObsPt in program order;
15         for SD ∈ NextDefs do
16             for O ∈ operands of SD do
17                 insertArtificialDefinition(O,SD,Before);
```

These additional precautions, including the extension to non-SSA programs, are implemented in Algorithm 2. This time, all definitions of "sibling" variables are considered, through the iteration over all definitions stemming from a given *source variable* in the functional property via renaming and live range splitting. Notice the "program order" requirement (lines 6 and 13) that prevents reordering of live ranges.

Applying the revised algorithm to Listing 1.a yields the code in Figure 2b. Let us illustrate our approach on the observed variable `r` from the source program. We use debug information to retrieve all definitions of `r` in the IR, with `%r2` being its reaching definition for the observation point. To prevent any preceding definition (i.e. `%r1`) from being moved after `%r2`'s definition, Algorithm 2 inserts artificial definition (of `%r1'`) right after the definition of `%r1` (line 6), then artificial definition of `%r2'` is inserted right before the observation point. `%r2'` also ensures that the reaching definition of the observed variable (`%r2`) cannot be optimized out (line 8). Similarly, artificial definitions are inserted to prevent all succeeding definitions of the observed variable from being moved up before the observation point (line 12): e.g. `%0`'s definition is placed right before `%r3`. According to the use-def property, and as the relative order of artificial definition (e.g. `%r1'`, `%r2'`, `%0`) and the observation point cannot be altered (a property of artificial definitions and observation points), it is guaranteed that the correct value of the observed variable `r` will always reach the observation point (via the variable `%r2'`). Helper function *insertArtificialDefinition* was described with Algorithm 1; it is extended here to operate either on instruction operand or definition of a variable, be it SSA variable or not.

5.4 Implementation in LLVM

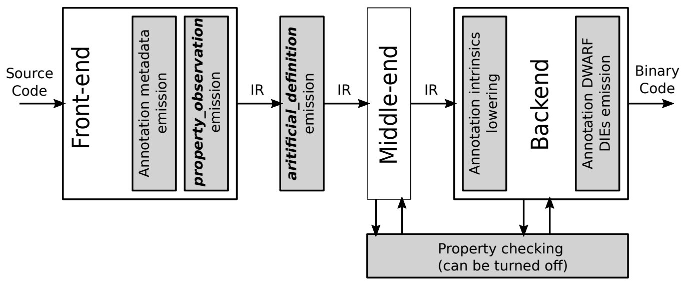

Source Code
Front-end
Annotation metadata emission
property_observation emission
IR
artificial_definition emission
IR
Middle-end
IR
Annotation intrinsics lowering
Backend
Annotation DWARF DIEs emission
Binary Code
Property checking (can be turned off)

Figure 3. Overview of the compilation flow extensions. Grey boxes represent new components.

Let us now introduce our modifications to LLVM to support property preservation. Figure 3 gives an overview of the augmented framework. All developments took place in the latest, continually updated version of clang and LLVM.

The LLVM IR allows metadata to be attached to instructions, to convey additional information to optimizers and code generators [28]. In particular, LLVM debug information is implemented as metadata. It is generally maintained and updated by optimization passes. Since we want to emit functional properties as DWARF DIEs in the binary, we opted for using debug information metadata to represent properties in the IR. We introduce a new type of metadata containing the formula, to which we attach classical debug metadata representing the observed variables and memory locations.

A subtle point about metadata is that while optimizations strive to maintain it—debug information in particular, it is always safe to discard it without affecting correctness. When it comes to transmitting functional properties, we need to ensure the metadata presence and its correctness even at the cost of losing some optimizations. We have to make sure that

[4] This is related to the program dependence web in classical compiler texts [6].

no optimization removes metadata representing functional properties, observed variables or memory locations.

We added new intrinsics to the IR: `property_observation` for observation points—to which the property metadata is attached, as well as `artificial_definition` which acts as a live range splitting mechanism constraining optimizations across a compiler fence. As explained in Section 4, `property_observation` embeds a compiler fence to guarantee the correctness and ordering of the values stored in the observed memory locations. In addition, both intrinsics are declared as having side-effects to preserve the relative order among artificial definitions and with the observation point. Later in the LLVM back-end, these intrinsics are lowered into pseudo-instructions with side-effects; they do not emit any machine instruction. Moreover, the artificial definition pseudo-instructions use the same source and destination register so that they can trivially be removed at instruction emission.

We created an LLVM library that parses the property string to build a list of observed variables and memory locations. This library is used by an extension of clang to generate, from the GNU annotation attributes, the appropriate property metadata along with `property_observation` intrinsics.

We implemented Algorithm 2 as an IR pass that is run before any optimizations. It works on LLVM IR produced by clang, which is not in SSA form yet. This is because the LLVM SSA construction pass `mem2reg` is run only after a few other optimizations, at which point the pristine partial state at the observation point defined by the source-level annotation may already be compromised. We also modified the compiler back-end so that property DIEs are built from the property metadata, and emitted into the object file's debug section. Moreover, we modified LLVM's DWARF reader library to support evaluating properties in binary analysis tools.

Finally, we implemented a mechanism to verify the presence and sanity of functional properties throughout the compilation flow. Before performing any optimization, we insert a LLVM pass that registers all properties within the program into the metadata section. Then, after each optimization pass, we insert a verification pass checking the presence of the metadata representing the property, its observed variables and memory locations. A warning informs the programmer if any verification pass fails; she may react by annotating the program differently, or disabling the optimization. This optional mechanism is only used for validation purposes.

6 Experimental Validation

We now present the experimental methodology, followed by functional validation and applications to security properties.

6.1 Methodology

Property preservation is defined as the equality of observation traces. Our validation approach is based on comparing, for given input data, the observation trace produced when executing the binary compiled with our property-compliant compiler against a reference observation trace. For this purpose, we assume `-O0` preserves the partial state of the ISO C abstract machine [38] containing the properties' observed variables and memory locations.[5]

All traces are obtained using the debugger gdb version 8.3. For the reference trace, we compile the original program *without properties*. We insert a C label instead and set a breakpoint for it in the debugger. We mirror all the variables and memory locations that should be observed into specially-named variables and dump their values using the debugger. Since the reference is compiled with `-O0` these mirrored variables are not optimized out. For all other observation traces, we use the modified DWARF reader to retrieve the addresses of the observation points, as well as the binary representation (constant value, register number or memory address) of the observed variables and memory locations. The binary is executed and the values of the observed variables and memory locations at the different observation points are retrieved using the debugger. Note that instrumentation mirroring is only used for functional validation; it is not activated in any performance or compilation time measurement.

6.2 Functional Validation

We validate our implementation on the test suite of Frama-C, a reference source code analysis platform for C. Properties are written in ACSL as program annotations. The test suite is designed to validate different Frama-C analyses on a range of small C programs representative of the language semantics. We restrict ourselves to boolean expressions as functional properties, ignoring test cases referring to more advanced ACSL built-in constructs. This results in 30 applicable test cases featuring 558 functional properties. Most of these properties verify the expected values of different variables at a given program point. These test cases are not meant to be evaluated as performance benchmarks, we only use them to validate the correctness of our implementation.

The validation platform is a quad-core 2.5 GHz Intel Core i5-7200U CPU with 16 GB of RAM. We target the x86-64 instruction set and compile each of these test cases at 5 optimization levels `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`. We verified that all 558 properties have been correctly propagated to the binaries and produce identical observation traces to the reference one, for all 5 optimization levels considered.

[5]This is not the case in general, as C does not fully specify the ordering of commutative and associative operations, evaluation of function arguments, etc. We mitigate these ambiguities when generating the reference observation trace by linearizing the expressions involved in functional properties to three-address form. Defining one single observed variable at a time is sufficient to fix the order of the partial states.

6.3 Security Properties

Let us now study real-world examples of security properties that would benefit from property preservation. Applications are commonly secured by inserting protections at the source code level. However, compilers may not understand the programmers' intentions, missing the implicit link between secure code protections and the control flow or machine state assumptions underlying its function, optimizing the protection away as a result. As a workaround, security engineers attempt to confuse the compiler by resorting to compiler-dependent coding tricks. This is obviously error-prone, dangerous, and not future-proof as compilers are getting better at removing "unnecessary" code [58].

Instead, following our approach, programmers would have the ability to instruct compilers to preserve the protections and enforce the associated security properties. These properties are *non-functional*: they refer to machine state unaccessible to the source program semantics. More specifically, we will consider 4 use cases covering the following non-functional properties necessary to the effectiveness of the countermeasures:

- proper erasure of sensitive data in memory;
- proper instruction ordering in masked secret key operations;
- proper fine-grained interleaving of functional and protection code;
- presence of redundant code to detect fault injections.

To make our case, *we thus need to encode the implicit assumptions underlying code hardening techniques using functional properties of the source program.* For this purpose, we extend the property language with a minimalistic predicate called `observe()` to determine which are the observed variables and memory locations at a given observation point. This predicate does not encode any logical formula; it takes the variable or memory location to be observed as argument and simply includes it into the partial state defined by the property and thus into the program observation trace.

In the following, for each security use-case, we present the security issue, the associated source-level protection scheme and security property, and how to encode the latter as a functional property using the predicate `observe()`. We also explain, when they exist, any alternate programming tricks to prevent the compiler from invalidating the security property. We check the security property preservation with our approach by comparing traces for all optimization levels `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`. We then analyze the impact of preserving source-code protections on the program performance and compilation time. We target the ARMv7-M instruction set (Cortex-M3 embedded processor). For the trace generation, we emulate the execution of the applications with the QEMU emulator version 3.0.1. The performance results use ARM Fast Models [9].

6.3.1 Secret Erasure.

Cryptographic applications need to erase secret data after usage [53]. For example in Listing 2.a, the sensitive buffer on the stack must be zeroed with a call to `memset` to avoid leaking confidential information; however, most compilers will spot that the buffer is not accessible after the function returns, removing the call to `memset` as part of "dead store elimination".

mbedTLS [52] tries to trick the compiler as shown in Listing 2.b, using a volatile function pointer to the standard `memset`. However, a compiler may still load `memset_func` into a register, comparing it to `memset`, and perform the function call only if they differ [53].

Instead, we insert a property right after zeroing, as illustrated in Listing 2.c. Considering mbedTLS's RSA encryption and decryption, called `rsa-encrypt` and `rsa-decrypt` in the following, at all the optimization levels, the `observe` property forces the effective zeroing of all secret buffers even with optimizations enabled. This is a simple example of combining hardening code and a functional property to enforce a security property (leakage prevention).

Note that this countermeasure deals with leakage from sensitive data in memory only. Leakage from data in registers is not supported; it incurs erasing any possible register where sensitive values may have resided in a secure function, which is not easily expressed as a source or IR-level property [58]. This is left for future work.

6.3.2 Computation Order.

Respecting the computation order of associative operations, as written in the source code, is hard with an optimizing compiler. The C language is defined in terms of an abstract machine producing an *observable behavior.* A compiler can optimize a conforming program, as long as the generated observable behavior matches the one from the C language abstract machine; it is thus free to reorder associative operations, even with proper parenthesizing. To make things worse, this is independent of the optimization level, and programmers usually have no control over it.

This becomes an issue, when for example bitmasking with `xor` operators as a countermeasure against side-channel attacks [37]. It has been reported that the C statement in Listing 2.d has been compiled as `k[0]^(mpt[0]^m)`, which altogether defeats the countermeasure [25].

A frequent mitigation trick is to insert `volatile` keywords and compiler fences, as in Listing 2.e. This may lead to slower code or may be optimized in the future [58].

Instead, we use the functional property `observe(tmp)`, as shown in Listing 2.f, to preserve computation order on a masked implementation of *Advanced Encryption Standard* (AES) [35], named `aes-herbst` in the following. We verified at all optimization levels that the temporary variable at the observation point did hold the expected value, as found in the reference observation trace generated from the Listing 2.e version.

6.3.3 Step Counter Incrementation.

Fault attacks are a growing threat for secure devices such as smart cards. Such

```
1 /// a. attempt to zero a buffer
2 void process_sensitive(void) {
3   uint8_t secret[32];
4   ...
5   memset(secret, 0, sizeof(secret));
6 }
```

```
1 /// b. hidden erasure implementation for mbedTLS
2 static void *(*const volatile memset_func)
3   (void*, int, size_t) = memset;
4 void mbedtls_zeroize(void *buf, size_t len) {
5   memset_func(buf, 0, len);
6 }
```

```
1 /// c. zero a buffer using an annotation
2 void process_sensitive(void) {
3   uint8_t secret[32];
4   ...
5   memset(secret, 0, sizeof(secret));
6   // Property: observe(secret)
7 }
```

```
1 /// d. bitmasking example
2 round_key[0] = (k[0] ^ mpt[0]) ^ m;
```

```
1 /// e. bitmasking with compiler fence
2 volatile uint8_t tmp = k[0] ^ mpt[0];
3 __asm__ __volatile__("":::"memory");
4 round_key[0] = tmp ^ m;
```

```
1 /// f. bitmasking using an annotation
2 uint8_t tmp = k[0] ^ mpt[0];
3 // Property: observe(tmp)
4 round_key[0] = tmp ^ m;
```

```
1 /// g. memcpy using an annotation
2 void secure_memcpy(char *s, char *d, size_t n) {
3   size_t i, j, size = n;
4   for (i = 0, j = 0; i < n; ++i, ++j) {
5     // Property: observe(i); observe(j)
6     if (j >= n) fault_detected();
7     d[i] = s[i];
8   }
9   if (j < n) fault_detected();
10  // Property: observe(n); observe(size)
11  if (n != size) fault_detected();
12 }
```

Listing 2. Secure code examples.

attacks can alter the system's correct behavior via physical injection means [61]. For example, it has been shown that fault attacks can induce unexpected jumps to any location in the program [14, 47]. One source-level scheme to enhance the resilience against such fault attacks consists in defining a step counter at each control construct, and stepping the counter of the immediately enclosing control construct after every C statement of the original source [40]. Counters are checked against their expected values before any incrementation, calling an exception handler when it fails. We refer to this technique as Step Counter Incrementation (SCI); it may be seen as a very fine-grained form of Control Flow Integrity (CFI) [2, 16].

However, as fault attacks are not modeled in compilers, optimizations will remove any counter checks—their conditions are trivially true. Counter incrementations might also be removed or grouped into a single block of code. As a result, practitioners making use of this source-level hardening scheme have to disable compiler optimizations. Instead, we protect counter incrementations and checks using functional properties: we observe the counter value before every check or incrementation. Furthermore, to guarantee that the functional and countermeasure instructions are correctly interlaced, we observe before each incrementation all other program variables and memory locations containing valid values.

We validated this approach on two well-known smart-card benchmarks: PIN authentication [23] and AES encryption [45], called `pin-sci` and `aes-sci` in the following. For the former, all values are correct w.r.t. values from the reference observation trace. For the latter, the debugger was unable to retrieve some values, always less than 0.2% (e.g. 9907 out of 8353591 at `-O2` and `-O3`). All retrieved values were correct. The unavailable values are due to LLVM's back-end generating, for the corresponding observed variables, incorrect location information in the debug information. We did manually verify the presence of the observed variables for all these unavailable values: while they do have their expected values stored in a register or in memory, gdb does not know about it and reports the variables as optimized out (a bug report has been filed).

6.3.4 Control and Data Flow Redundancy.

Loops in sensitive code are important targets of fault attacks. For example, it has been shown that corrupting `memcpy` during the initialization of an embedded system may allow an attacker to escalate privileges and execute arbitrary code [60]. Other work also highlighted the need to protect the iteration count of PIN authentication [23].

There has been recent work to harden sensitive loops at compilation time, by duplicating termination conditions and the computations involved in the evaluation of such conditions [55]. While the original algorithm operates on the IR and takes care of positioning it with respect to downstream and upstream passes, we attempt to make the approach more generic by implementing it at source level. However, since the approach relies on redundant computations, the difficulty lies in preserving the protection from optimizations: redundant operations do not impact the observable semantics and are ideal candidates to be optimized away by the compiler [36].

Given the `memcpy` loop shown in Listing 2.g, we duplicate the loop counter `i` and the computation of the exit condition at every iteration of the loop (line 6), as well as at the loop exit (line 9). We also duplicate loop-independent variables that are used in the loop body, and verify that their values at loop exit are correct with respect to their original values (line 11). To prevent optimizations from altering this protection, we observe both the original and duplicate variables right before the redundant computations, to make sure that their values are always available for use. We implemented the loop hardening scheme on the `memcpy` function above, and on a `memcmp`-like function for PIN authentication [23]; both are included `loop-redundant` in the following. We then validated the preservation of the protections down to machine code.

6.3.5 Performance and Compilation Overhead.

Let us now analyze the run-time performance and compilation overhead for all security applications. We compare our versions with binaries generated with (1) no optimizations at all, (2) with the common-practice (unreliable) programming tricks to prevent the compiler from removing source-code protections and (3) to set an upper bound on the achievable performance, with the unsafe binaries compiled without any property preservation mechanism.

Performance. We measure the program performance in terms of number of instructions executed in the main program, using ARM Fast Models [9]. This is the most consistent choice given the Cortex-M3's very simple pipeline; it is more relevant than execution time given the wide diversity of the processor silicon implementations in real devices. For each application, Figure 4 presents the performance ratio of different versions compiled at different optimization levels w.r.t. the original program compiled at `-O0`, which serves as a baseline. The first version corresponds to the original code without any modification, the second includes programming tricks described per security use-case in previous subsections, and the last one has source-code annotations inserted, also described in previous subsections. These three versions are referred as `Unsafe`, `Tricks` and `Annotations` respectively in Figure 4. It is worth noting that there are no programming tricks to reliably preserve source-level step counter incrementation and loop protection redundancy. Results show that (1) the unsafe versions yield to the fastest but non-secure executables, since protections are modified or removed with optimizations enabled; (2) compared to existing fragile programming tricks, our compiler preserves the source-level protections with similar, if not better performance; (3) when no trick exists (`loop-redundant`, `pin-sci` and `aes-sci`), our compiler provides consistent performance improvement over programs compiled at `-O0` while preserving the source-level protection. The higher cost of preserving the protection of `pin-sci` compared to the one of `aes-sci` can be traced back to the functional/protection code ratio for these 2 programs. Indeed, `pin-sci` features about 4 times more protection code than functional code, with the functional code being too trivial to be optimized within the boundaries of the protection statements; while `aes-sci` contains only twice more protection code than functional code. Moreover, the functional code of `aes-sci` is more complex, leaving potential for classical optimizations. Finally, note that performance is not the primary motivation in these examples: anything better than `O0` is beneficial to security engineers.

Compilation Time. Figure 5 shows the compilation-time overhead for all applications, compared to the original program (without annotations), compiled with the same optimization flag on the same Intel platform described in Section 6.2. RSA encryption and decryption algorithms are not compiled separately, we thus measured the compilation-time of the

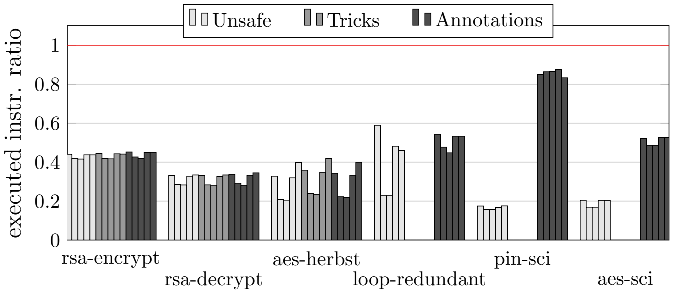


Figure 4. Executed instructions w.r.t. `-O0` baseline, ordered by optimization level `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`

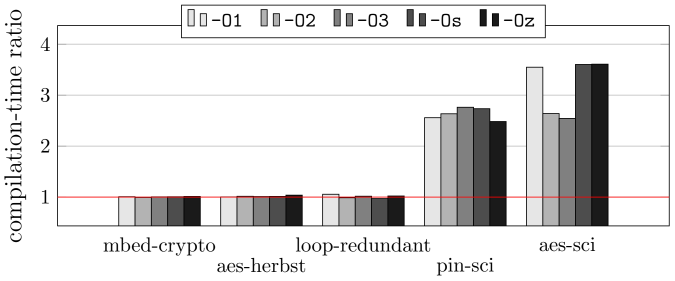


Figure 5. Compilation-time w.r.t. original program without functional property annotations

whole Mbed cryptography library. In general, the compilation overhead is under 5%, though it can sometimes be really important, when *complete* (and not partial) program state is constantly observed. For `pin-sci` and `aes-sci`, the protection scheme introduces at least one annotation for every functional C statement, and the complete program state is observed: this is really a worst case scenario. In fact, overhead depends on intended protections. Moreover, our prototype is not yet tuned and optimized and we believe that compilation-time can be reduced with additional algorithmic and engineering effort.

7 Conclusion

We motivated and proposed an approach to encode, translate, and preserve the semantics of both functional and non-functional properties, across all program representations through the optimizing compilation of C to machine code. The approach relies on a notion of functional property preservation, and its implementation in a compiler such that data and control flow optimizations will preserve the consistency of these properties across transformations. We validated our approach in the LLVM framework, with no changes to existing optimization passes beyond bug fixes related to the propagation of debug information. While the problem we consider may have general applications in software engineering, our proposal specifically addresses a fundamental open issue in security engineering.

References

[1] Martín Abadi. 1998. Protection in programming-language translations. In *Automata, Languages and Programming (ICALP) (LNCS)*, Vol. 1443. Springer.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA) *(CCS '05)*. ACM, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[3] Martín Abadi and Gordon D. Plotkin. 2012. On protection by layout randomization. *ACM Trans. on Information System Security* 15, 2 (2012).

[4] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2018. Exploring Robust Property Preservation for Secure Compilation. *CoRR* abs/1807.04603 (2018). arXiv:1807.04603 http://arxiv.org/abs/1807.04603

[5] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*. 256–271. https://doi.org/10.1109/CSF.2019.00025

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[7] aiT [n.d.]. aiT. https://www.absint.com/ait/index.htm. Accessed 19 May 2018.

[8] Mehdi-Laurent Akkar and Christophe Giraud. 2001. An Implementation of DES and AES, Secure Against Some Attacks. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems (CHES '01)*. Springer-Verlag, London, UK, UK, 309–318. http://dl.acm.org/citation.cfm?id=648254.752562

[9] ARM. 2019. *ARM Fast Models*. https://developer.arm.com/tools-and-software/simulation-models/fast-models

[10] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 23 (Aug. 2010), 84 pages. https://doi.org/10.1145/1749608.1749612

[11] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2010. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *Software Technologies for Embedded and Ubiquitous Systems*, Sang Lyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 35–46.

[12] Thierno Barry, Damien Couroussé, and Bruno Robisson. 2016. Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems* (Prague, Czech Republic) *(CS2 '16)*. ACM, New York, NY, USA, 1–6. https://doi.org/10.1145/2858930.2858931

[13] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. 2008. *ACSL: ANSI/ISO C Specification Language Version 1.4*. https://frama-c.com/download/acsl.pdf

[14] Pascal Berthomé, Karine Heydemann, Xavier Kauffmann-Tourkestansky, and Jean-François Lalande. 2012. High level model of control flow attacks for smart card functional security. In *7th International Conference on Availability, Reliability and Security*. IEEE Computer Society, Prague, Czech Republic, 224–229. https://doi.org/10.1109/ARES.2012.79

[15] Jean-Baptiste Bréjon, Karine Heydemann, Emmanuelle Encrenaz, Quentin Meunier, and Son Tuan Vu. 2019. Fault attack vulnerability assessment of binary code. In *6th Workshop on Cryptography and Security in Computing Systems (CS2)*. Valencia, Italy. https://doi.org/10.1145/3304080.3304083

[16] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* 50, 1, Article 16 (April 2017), 33 pages. https://doi.org/10.1145/3054924

[17] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. 1999. Towards Sound Approaches to Counteract Power-Analysis Attacks. In *Advances in Cryptology — CRYPTO' 99*, Michael Wiener (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 398–412.

[18] Adam Chlipala. 2007. A Certified Type-preserving Compiler from Lambda Calculus to Assembly Language. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. ACM, New York, NY, USA, 54–65. https://doi.org/10.1145/1250734.1250742

[19] DWARF Debugging Information Format Commitee. 2017. *DWARF Debugging Information Format Version 5*. https://dwarfstd.org/doc/DWARF5.pdf

[20] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C: A Software Analysis Perspective. In *10th International Conference on Software Engineering and Formal Methods* (Thessaloniki, Greece). 233–247. https://doi.org/10.1007/978-3-642-33826-7_16

[21] Ronald De Keulenaer, Jonas Maebe, Koen De Bosschere, and Bjorn De Sutter. 2016. Link-time smart card code hardening. *International Journal of Information Security* 15, 2 (01 Apr 2016), 111–130. https://doi.org/10.1007/s10207-015-0282-0

[22] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-Abstract Compilation by Approximate Back-Translation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 164–177. https://doi.org/10.1145/2837614.2837618

[23] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. 2016. FISSC: A Fault Injection and Simulation Secure Collection. 3–11. https://doi.org/10.1007/978-3-319-45477-1_1

[24] Kerstin Eder, John P. Gallagher, Pedro López-García, Henk Muller, Zorana Banković, Kyriakos Georgiou, Rémy Haemmerlé, Manuel V. Hermenegildo, Bishoksan Kafle, Steve Kerrison, Maja Kirkeby, Maximiliano Klemen, Xueliang Li, Umer Liqat, Jeremy Morse, Morten Rhiger, and Mads Rosendahl. 2016. ENTRA. *Microprocess. Microsyst.* 47, PB (Nov. 2016), 278–286. https://doi.org/10.1016/j.micpro.2016.07.003

[25] Hassan Eldib and Chao Wang. 2014. Synthesis of Masking Countermeasures Against Side Channel Attacks. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 114–130. https://doi.org/10.1007/978-3-319-08867-9_8

[26] David Evans. 1996. Static Detection of Dynamic Memory Errors. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. ACM, New York, NY, USA, 44–53. https://doi.org/10.1145/231379.231389

[27] Heiko Falk and Paul Lokuciejewski. 2010. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems* 46, 2 (01 Oct 2010), 251–300. https://doi.org/10.1007/s11241-010-9101-x

[28] LLVM Foundation. 2019. *LLVM Language Reference Manual*. https://llvm.org/docs/LangRef.html#metadata

[29] Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet, and Axel Legay. 2017. An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT. In *2017 IEEE Trustcom/BigDataSE/ICESS*. 293–300. https://doi.org/10.1109/Trustcom/BigDataSE/ICESS.2017.250

[30] Rigel Gjomemo, Kedar S. Namjoshi, Phu H. Phung, V. N. Venkatakrishnan, and Lenore D. Zuck. 2015. From Verification to Optimizations. In *Verification, Model Checking, and Abstract Interpretation*, Deepak D'Souza, Akash Lal, and Kim Guldstrand Larsen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 300–317.

[31] Daniele Gorla and Uwe Nestmann. 2016. Full abstraction for expressiveness: history, myths and facts. *Mathematical Structures in Computer Science* 26, 4 (2016), 639–654. https://doi.org/10.1017/S0960129514000279

[32] Lucien Goubet, Karine Heydemann, Emmanuelle Encrenaz, and Ronald De Keulenaer. 2015. Efficient Design and Evaluation of Countermeasures against Fault Attack with Formal Verification. In *14th International conference Smart Card Research and Advanced Applications (CARDIS) (Lecture Notes in Computer Science)*, Vol. 9514. Springer International Publishing, Bochum, Germany, 177–192. https://doi.org/10.1007/978-3-319-31271-2_11

[33] Louis Goubin and Jacques Patarin. 1999. DES and Differential Power Analysis The “Duplication” Method. In *Cryptographic Hardware and Embedded Systems*, Çetin K. Koç and Christof Paar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 158–172.

[34] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. 2002. Profile Guided Compiler Optimizations. *The Compiler Design Handbook: Optimizations and Machine Code Generation* (02 2002). https://doi.org/10.1201/9781420040579.ch4

[35] Christoph Herbst, Elisabeth Oswald, and Stefan Mangard. 2006. An AES Smart Card Implementation Resistant to Power Analysis Attacks. In *Proceedings of the 4th International Conference on Applied Cryptography and Network Security* (Singapore) *(ACNS’06)*. Springer-Verlag, Berlin, Heidelberg, 239–252. https://doi.org/10.1007/11767480_16

[36] Christoph Hillebold. 2014. *Compiler-Assisted Integrits against Fault injection Attacks*. Master’s thesis. University of Technology, Graz. http://chille.at/articles/master-thesis

[37] Yuval Ishai, Amit Sahai, and David Wagner. 2003. Private Circuits: Securing Hardware against Probing Attacks. In *Advances in Cryptology - CRYPTO 2003*, Dan Boneh (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 463–481.

[38] ISO. 2011. *C11 Standard*. /bib/iso/C11/n1570.pdf ISO/IEC 9899:2011.

[39] Daniel Jackson. 1995. Aspect: Detecting Bugs with Abstract Dependences. *ACM Trans. Softw. Eng. Methodol.* 4, 2 (April 1995), 109–145. https://doi.org/10.1145/210134.210135

[40] Jean-François Lalande, Karine Heydemann, and Pascal Berthomé. 2014. Software countermeasures for control flow integrity of smart card C codes. In *ESORICS - 19th European Symposium on Research in Computer Security (Lecture Notes in Computer Science)*, Miroslaw Kutylowski and Jaideep Vaidya (Eds.), Vol. 8713. Springer International Publishing, Wroclaw, Poland, 200–218. https://doi.org/10.1007/978-3-319-11212-1_12

[41] Chris Lattner and Vikram Adve. 2004. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[42] Juneyoung Lee, Chung-Kil Hur, Ralf Jung, Zhengyang Liu, John Regehr, and Nuno P. Lopes. 2018. Reconciling High-level Optimizations and Low-level Code in LLVM. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 125 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276495

[43] Xavier Leroy. 2006. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL 2006: 33rd symposium Principles of Programming Languages*. ACM, 42–54. https://doi.org/10.1145/1111037.1111042

[44] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom Reasoning* 43, 363 (2009). https://doi.org/10.1007/s10817-009-9155-4.

[45] Ilya Levin. 2007. *A byte-oriented AES-256 implementation*. http://www.literatecode.com/aes256

[46] Hanbing Li, Isabelle Puaut, and Erven Rohou. 2014. Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems* (Versaille, France) *(RTNS ’14)*. ACM, New York, NY, USA, Article 97, 10 pages. https://doi.org/10.1145/2659787.2659805

[47] Nicolas Moro, Amine Dehbaoui, Karine Heydemann, Bruno Robisson, and Emmanuelle Encrenaz. 2013. Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. In *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography*. 77–88. https://doi.org/10.1109/FDTC.2013.9

[48] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. 2012. Compiler Assisted Masking. In *Cryptographic Hardware and Embedded Systems – CHES 2012*, Emmanuel Prouff and Patrick Schaumont (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 58–75.

[49] Kedar S. Namjoshi, Giacomo Tagliabue, and Lenore D. Zuck. 2013. A Witnessing Compiler: A Proof of Concept. In *Runtime Verification*, Axel Legay and Saddek Bensalem (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 340–345.

[50] Kedar S. Namjoshi and Lenore D. Zuck. 2013. Witnessing Program Transformations. In *Static Analysis*, Francesco Logozzo and Manuel Fähndrich (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 304–323.

[51] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.* 37, 2, Article 6 (April 2015), 50 pages. https://doi.org/10.1145/2699503

[52] Paul Bakker, ARM. 2019. mbedTLS. tls.mbed.org

[53] Colin Percival. 2014. *How to zero a buffer*. http://www.daemonology.net/blog/2014-09-04-how-to-zero-a-buffer.html

[54] Adrian Prantl, Markus Schordan, and Jens Knoop. 2008. TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis. In *8th International Workshop on Worst-Case Execution Time Analysis (WCET’08) (OpenAccess Series in Informatics (OASIcs))*, Raimund Kirner (Ed.), Vol. 8. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. https://doi.org/10.4230/OASIcs.WCET.2008.1661 also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-237-3.

[55] Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. 2017. Compiler-Assisted Loop Hardening Against Fault Attacks. *ACM Trans. Archit. Code Optim.* 14, 4, Article 36 (Dec. 2017), 25 pages. https://doi.org/10.1145/3141234

[56] G. Ramalingam. 1996. Data Flow Frequency Analysis. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI ’96)*. ACM, New York, NY, USA, 267–277. https://doi.org/10.1145/231379.231433

[57] Bernhard Schommer, Christoph Cullmann, Gernot Gebhard, Xavier Leroy, Michael Schmidt, and Simon Wegener. 2018. Embedded Program Annotations for WCET Analysis. In *WCET 2018: 18th International Workshop on Worst-Case Execution Time Analysis*, Vol. 63. Dagstuhl Publishing, Barcelona, Spain. https://doi.org/10.4230/OASIcs.WCET.2018.8

[58] Laurent Simon, David Chisnall, and Ross Anderson. 2018. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 1–15. https://doi.org/10.1109/EuroSP.2018.00009

[59] Richard M. Stallman and GCC DeveloperCommunity. 2009. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA.

[60] Niek Timmers, Albert Spruyt, and Marc Witteman. 2016. Controlling PC on ARM Using Fault Injection. In *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*. 25–35. https://doi.org/10.1109/FDTC.2016.18

[61] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. 2018. Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation. *Journal of Hardware and Systems Security* 2, 2 (2018), 111–130. https://doi.org/10.1007/s41635-018-0038-1

[62] Niko Zarzani. 2013. *Improving the Compilation process using Program Annotations*. Master’s thesis. Politecnico Di Milano.