Secure Optimization Through Opaque Observations

SON TUAN VU, Sorbonne Université, CNRS, LIP6, France
ALBERT COHEN, Google, France
KARINE HEYDEMANN, Sorbonne Université, CNRS, LIP6, France
ARNAUD DE GRANDMAISON, Arm, France
CHRISTOPHE GUILLON, STMicroelectronics, France

Secure applications implement software protections against side-channel and physical attacks. Such protections are meaningful at machine code or micro-architectural level, but they typically do not carry observable semantics at source level. To prevent optimizing compilers from altering the protection, security engineers embed input/output side-effects into the protection. These side-effects are error-prone and compiler-dependent, and the current practice involves analyzing the generated machine code to make sure security or privacy properties are still enforced. Vu et al. recently demonstrated how to automate the insertion of volatile side-effects in a compiler [52], but these may be too expensive in fined-grained protections such as control-flow integrity. We introduce observations of the program state that are intrinsic to the correct execution of security protections, along with means to specify and preserve observations across the compilation flow. Such observations complement the traditional input/output-preservation contract of compilers. We show how to guarantee their preservation without modifying compilation passes and with as little performance impact as possible. We validate our approach on a range of benchmarks, expressing the secure compilation of these applications in terms of observations to be made at specific program points.

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

Additional Key Words and Phrases: compiler, security, optimization, debugging, LLVM

1 INTRODUCTION

Compilers care about preserving the input/output (I/O) behavior of the program; they achieve this by preserving functional correctness of computations linking input to output values, and making sure these take place in the order and under the conditions specified in the source program.

Interestingly, this is not enough for many compilation scenarios, where optimizations are too aggressive at removing, reordering or otherwise modifying computations that do not result in externally visible I/O. We identified four such scenarios:

(1) **Preventing software and side-channel attacks.** Optimizing compilers are known to interfere with a wide range of security properties. For example, dead store elimination may optimize out procedures specifically designed to erase sensitive data in memory, thereby exposing encryption keys and other secrets to be accessed by an attacker or captured in a memory dump [23, 42, 48]. In cryptography applications, countermeasures against side-channel attacks are even more fragile: optimizations may invalidate a wide range of masking protections—randomizing sensitive data [45]—by reordering elementary operations in masking expressions [14]. Related to this and to forestall timing attacks, encryption/decryption kernels are usually designed to run for a constant amount of time, independent of sensitive inputs [53]. To achieve this, security engineers go to great lenghts to write straight line code, carefully avoiding control flow depending on sensitive data [48]; unfortunately, compilers

*Preprint presented at the PriSC workshop, January 17, 2021 (with POPL 2021).

Authors' addresses: Son Tuan Vu, Sorbonne Université, CNRS, LIP6, 4 place Jussieu, 75252, Paris, France, son-tuan.vu@lip6.fr; Albert Cohen, Google, Paris, France, albertcohen@google.com; Karine Heydemann, Sorbonne Université, CNRS, LIP6, 4 place Jussieu, 75252, Paris, France, karine.heydemann@lip6.fr; Arnaud de Grandmaison, Arm, Paris, France, arnaud.degrandmaison@arm.com; Christophe Guillon, STMicroelectronics, Grenoble, France, christophe.guillon@st.com.

often substitute dataflow encodings of control flow (conditional moves, bitwise logic) with more efficient, performance-wise or size-wise, conditional control flow, defeating the purpose of the constant time implementation [48].

(2) **Preventing fault attacks.** Fault attacks are a growing threat for embedded systems. They can alter the system's correct behavior by means of physical interference [54]. Software countermeasures against such attacks usually involve some form of redundancy, such as computing the same operation on one or more copies of the data then comparing the results [10, 11, 31, 43]. However, this has been a constant fight with compilers as one of the most essential goals of compiler optimizations is to removing redundant code [31, 48]. Another well-known countermeasure which aims at detecting fault attacks altering the program's control flow consists in incrementing counters along side with the execution of individual instructions, source code statements, function calls, etc., then checking their expected values when reaching control-flow merge points [34]. Once again, as compilers do not model the security intention of the countermeasure, they will remove the trivially true counter checks or collect multiple counter incrementations into a single addition.

(3) **Testing, inspecting or verifying machine code.** In security-sensitive applications, these are classical procedures, mandated by certification authorities and standards. It includes checking for the presence of countermeasures against attacks of the form described in the previous two items. There has been a large body of work in software security showing the importance of analysis and verification tools assessing program properties—expressed as a propositional logic formula—at the level of machine code [8, 16, 47]. The need for such analyses derives from the observable mismatch between the behavior intended by the programmer and what is actually executed by the processor [8], or from the essential role played by low-level (micro-)architectural features [16]. More generally, machine code analysis tools and runtime monitors (including debuggers) often need program properties representing high-level program specification, from which the static or dynamic analysis may determine whether the application is secure against a given attack [16]. Still, it is generally challenging to propagate source-level program properties all the way down to machine code [52]. Compilers have no notion of the link between properties expressed as annotations, semantical comments or pragmas, and the semantics of the code they refer to. As a result, compilers generally do not preserve this link or update properties according to the transformations they apply to the code (e.g., updating static bounds on the number of loop iterations when performing loop unrolling). Besides, variables referenced in program properties may also be affected by compiler optimizations, e.g. such variables may be optimized out, thus invalidating the property [52].

(4) **Debugging, unit testing.** Source-level properties can be a powerful debugging tool, helping enforce or detect security violations through the development and deployment process. This is similar to inserting (runtime) assertions, except that the code does not have to carry these runtime checks when running in production. Instead, the ability to propagate source-level properties down to machine code, allows a debugger to trace program execution and evaluate the properties in a testing environment. This brings the promise of using the same executable code in both testing and production environments. Unlike assertions, it is expected that machine code annotations do not consume execution time. On the contrary, their preservation through the compilation flow is not easy (as we have seen earlier in secure applications) and may also force the compiler to occasionally restrain its full optimization potential. We are not aware of any software engineering approach being currently pursued together with an aggressively optimizing compiler, precisely because optimizations are prone to destroying the link between the high-level properties and the machine code where they are meant to

be tested. And indeed, while these are common issues related to the propagation of debug information through optimization passes, the usual tradeoff in such a case is to preserve debug information only as a best-effort strategy, which is insufficient for our debug-only-assert scenario.

This paper is motivated by security applications, and we will focus on the first three scenarios in the following. The effectiveness of our approach in broader software engineering areas such as the fourth scenario is left for future work.

The common pattern in the above scenarios is that program transformations—such as abstraction lowering steps and optimizations—have no means to reason about non-I/O-related observations that the programmer would like to specify and whose preservation should be enforced. As a result, security engineers resort to embedding I/O such as volatile side-effects into security protections. Vu et al. recently demonstrated how to automate this in a compiler [52]; but volatile side-effects may be too expensive. We propose a means to *specify and preserve observations*, to do so in *the most general manner*, avoiding to modify each and every transformation pass in a compiler, extending automatically to future transformations, and with as little performance impact as possible. To this end, we leverage the most essential information modeled by nearly every compiler transformation: *I/O effects and data flow*. And we control this information according to the specified observations through the ability to hide information about an atom of operational semantics—a.k.a. *opacity*. We provide a concrete syntax and formal semantics to specify observations and preserve them through the compilation flow. We show that preserving observations does not require significant modifications to compilers and demonstrate this on the LLVM compiler with aggressive optimizations turned on. We validate our approach and implementation on a range of security-sensitive benchmarks, expressing the secure compilation of these applications in terms of observations to be made at specific points of the computation.

In this paper, we contribute the following:

- We define the notion of observation and its preservation through program transformations (Section 4).
- We present our mechanism to preserve observations down to machine code with minimal interference with compiler optimizations, and formally prove its correctness using a simplified intermediate program representation (Section 5).
- We detail our LLVM-based implementation with almost no modification to individual optimization passes (Section 6).
- We study concrete applications of our approach, to preserve security protections introduced at source level (Section 7).
- We validate the preservation of these security properties according to a range of criteria, to establish the correctness of our implementation (Section 8).
- We evaluate the performance and compilation time impact of our approach, and further compare with alternative mechanisms (Section 9).

2 RELATED WORK

There is a large body of research and engineering on secure compilation [1, 3, 4, 22, 29]. The correctness of a compiler is defined w.r.t. to a notion of behavioral equivalence, which may take different forms from full abstraction to more specific type and isolation properties [3, 18, 40] or even hyperproperties not directly captured in terms of behavioral equivalence [5]. Behavioral equivalence is generally defined with respect to the capabilities of an attacker. Our secure compilation problem targets a wide class of properties not directly expressible within a source language semantics: properties of the machine state resulting from the compilation of the program, including logical

properties of side-channels and countermeasures to physical attacks. Like Blazy et al. [12] we thus extend the semantics of a host language with state and denotations to reason about these extra-functional properties. But unlike Blazy et al. we do not focus on a specific kind of property (execution time in their case). We provide a general means to express observations at deterministic points of the execution, even though the program is subject to aggressive transformations. As presented by Vu et al. [52], these observations may then enable the expression and validation of a wide range of logical properties [13].

This section discusses the most closely related work, starting with Vu et al. [52] which laid the groundwork for this paper. To prevent interference from compiler optimizations, security engineers resort to embedding I/O such as volatile side-effects into security protections. Vu et al. automated this process [52], but as our experiments will show, side-effects may be too expensive in scenarios such as fine-grained control-flow integrity. This fact motivates our effort to distinguish observations from regular I/O mechanisms, and not encoding observations as fake I/O instructions. In addition, Vu et al. relied on a restrictive notion of behavioral equivalence by enforcing the equality of I/O and observation traces. In this paper, we provide security engineers with finer-grained control on the preservation of observations across transformations, and on the partial ordering of these observations.

Compilers for hard real-time systems are designed to carry detailed control flow information in order to bound the worst-case execution time of a reactive method as accurately as possible. This information is called *flow information* and takes the form of source code annotations about, for instance, loop trip counts, infeasible paths and program points that are mutually exclusive during a given run [6, 9]. There is no attempt at formalizing the preservation of control-flow information as a correctness requirement. Instead, CompCert relies on known and implementation-specific limitations of the compiler: it introduces a builtin function modeled as a call to an external function producing an observable event, without emitting it as machine code [46].

The ENTRA (Whole-Systems ENergy TRAnsparency) project Deliverable D2.1 [25] describes a similar mechanism to transfer information from source to machine code. Data and control flow properties are encoded as comments written as inline assembly expressions, relying on the compiler to preserve the local variables listed in the assembly expression. These expressions are declared as volatile I/O side-effecting to maintain their position in control flow relative to other code. This mechanism can be used to observe values and preserve them, but cannot be used to chain these observations and implement a partial ordering specification as the properties do not produce any value (i.e. no opacification). As a result, they cannot be used to preserve security protections.

Another safety-minded approach encodes flow information using IR extensions and external transformations to update loop trip count information [37]. This approach incurs significant changes to optimization passes: it comes with a set of rules to transform control flow information along code transformations.

The introduction already listed applications and motivating scenarios in security and software engineering. The reader may refer to Vu et al. [52] for a more extensive discussion of security-related work.

3 MOTIVATING EXAMPLE

Let us consider a cryptography application as a motivating example. The leakage of confidential information such as secret keys is a major threat. A common countermeasure consists in erasing sensitive data from memory once they are no longer needed [42], including keys, seeds of random generators, and temporary encryption or decryption buffers.

However, this may not be as easy as it seems: security engineers have been painfully fighting optimizing compilers to achieve their goal [42, 48]. Consider the example in Listing 1. The `secret`

buffer containing sensitive information is allocated on the stack and should be erased before returning from the function; this is implemented through a call to `memset()`. However, compilers will spot that `secret` goes out of scope, meaning that access after the function returns is an error or has unspecified behavior; since many optimizing compilers model `memset()` as a builtin function, they are aware of its semantics and will consider the call as dead stores, removing the erasure as part of "dead store elimination". While this completely subverts the security protection, it is perfectly correct with regard to the C semantics: the observable effects of the program are not modified by this optimization.

```
1 void process_sensitive(void) {
2   uint8_t secret[32];
3   ...
4   memset(secret, 0, sizeof(secret));
5 }
```

Listing 1. Erasing a secret buffer on the stack.

Different solutions have been implemented in OpenSSL [50] and mbedTLS [41], however none of them is guaranteed to work in all cases: compilers might still recognize the tricks and optimize them away [42, 48], thus potentially allowing unauthorized access to sensitive data.

At this point, we would like to emphasize that the security protection can be equivalently expressed as a write of the zero value to the secret buffer. This hints at a more general challenge of preserving specific values at specific points of the program execution. These points and values are associated with protection schemes and countermeasures dictated by security concerns, and they have to be preserved and traced down to machine code. As shown by this motivating example, applications are commonly secured by inserting protections at source level, but compilers may fail to implement the programmers' intentions as these protections often do not alter the program observable behavior, resulting in unsafe machine code. Our work aims to enable the programmer to instruct compilers into preserving specific computations and values, thus enforcing the associated security protections and properties. In the next section, we will formally define this observation specification mechanism and the value preservation problem.

4 PROBLEM STATEMENT AND DEFINITIONS

We introduce a simple language, called Mini IR, representative of the levels of Intermediate Representation (IR) typical of the compilation of imperative languages. It models control flow at a relatively low level—linearized three-address code— while supporting the usual memory abstractions, SSA values, intra- and inter-procedural constructs. For the sake of simplicity, we will use it to model not only optimizing compiler IR but also source code and assembly code.

4.1 Mini IR Syntax

Figure 1 presents the grammar of Mini IR (it will be extended in the next section).

In Mini IR, control flow is implemented as flat Control Flow Graph (CFG) of blocks and branches. Unlike traditional CFG- and SSA-based compilers (GCC and LLVM), we use branch and block arguments following continuation-passing style [7]. Like in MLIR [35], single-assignment variables are declared and scoped in a region (introduced by a function or macro) and captured in dominated blocks; as a consequence, branch arguments only need to carry SSA values, as opposed to explicitly carrying all live variables. This choice makes use-def chains more uniform across an entire function without implicitly referring to control flow edges, which in turns simplifies our formalization of a happens-before relation later in this section.

const	::=	*integer*	integer constants
var	::=	*ident*	identifier for an SSA value
expr	::=	*const*	
	|	*un-op var*	unary operator
	|	*var bin-op var*	binary operator
	|	*ident* `(` *var** `)`	function application or macro expansion
	|	`io(` *var*, *var** `)`	I/O effect with ordering descriptor
	|	`snapshot(` *var*$^+$ `)`	identity function observing its arguments into a partial state
instr	::=	*expr*	expression with no associated definition
	|	`[` *var*$^+$ `=]` *expr*	define a value from an expression
	|	*ref* `<-` *var*	store a value to a reference
	|	*var* `=` *ref*	load a value from a reference
	|	`mem[` *var* `]` `<-` *var*	store a value to a memory address
	|	*var* `=` `mem[` *var* `]`	load a value from a memory address
	|	`br` *var* `,` *ident* `(` *var** `)`	branch with condition, target block identifier and arguments
	|	`return(` *var** `)`	return from function or macro
block	::=	*ident* `[ (` *var** `)] :[` *instr* `;]`*	block labeled by a unique identifier, composed of arguments and an instruction sequence, branch- or return-terminated
region	::=	`{` *block*$^+$ `}`	return-terminated region with one or more blocks
func-decl	::=	`function` *ident* `(` *var** `)` *region*	function definition
macro-decl	::=	`macro` *ident* `(` *var** `)` *region*	macro definition

Fig. 1. Grammar of our Mini IR. The terminals *ident*, *un-op*, *bin-op*, *integer* are the same as the corresponding C lexical tokens.

Note that we did not include indirect branches and calls in the syntax. Supporting these would include making identifiers first class and holding them as additional SSA arguments of branch and call instructions. This does not impact the following formalization and expressing our secure compilation benchmarks.

When clear from the context, we will write "instruction *expr*" when referring to a instruction defining, assigning or returning a value from an expression *expr*.

4.2 Operational Semantics

All expressions and instructions have fairly standard semantics, except for `snapshot`, which will be presented in the next subsection.

As a simplifying assumption, we only consider *sequential*, *deterministic* programs with *well defined behavior*. In particular, we avoid cases where the compiler may take advantage of undefined behavior to trigger optimizations. This assumption is consistent with widespread coding standards for secure code. Our formalization also assumes *no exceptions* at the source language level, but precise machine exceptions at the instruction level are supported.

Definition 4.1 (Name-value domains). Every value manipulated during the execution of a program belongs to one of these four *Name-value domains*:

- $\mathcal{V}$ is a set of (*Var*, *Val*) pairs where *Var* is an SSA variable name (e.g. an SSA value in LLVM IR or a variable in a functional language) and *Val* is the value of *Var*; all uses of *Var* are dominated by an unique definition associating *Var* with its value *Val*;
- $\mathcal{C}$ is a set of (*Val*, *Val*) pairs where *Val* is a constant value also standing as the name of the constant;
- $\mathcal{R}$ is a set of (*Ref*, *Val*) pairs where *Ref* is a reference name (e.g. a C variable, a reference in a functional language, or a register in a low-level representation) and *Val* is the value referenced by *Ref*;
- $\mathcal{M}$ is a set of (*Mem*, *Val*) pairs where *Mem* is a memory address and *Val* is the value stored at *Mem*.

We define an operational semantics for our Mini IR in terms of a state machine, where every IR instruction defines a transition referred to as an *event*.

Definition 4.2 (Program state). A *program state* is defined by a tuple $(Vals, \pi)$ with $Vals = V \cup C \cup R \cup M$, where $V \subseteq \mathcal{V}$, $C \subseteq \mathcal{C}$, $R \subseteq \mathcal{R}$, $M \subseteq \mathcal{M}$, and the *program point* π holds the value of the program counter pointing to the next instruction to be executed.

Definition 4.3 (Event). An *event* e is a state machine transition, associated with the execution of an instruction i, from a state σ into a state σ'. It is denoted by $e = \sigma \overset{\mathrm{i}}{\rightsquigarrow} \sigma'$.

For any given event e, let $Inst(e)$ denote the instruction executed by event e.

Definition 4.4 (Program execution). A program execution E is a—potentially infinite—ordered sequence of program states and events:

$$E = \sigma_0 e_0 \sigma_1 e_1 \sigma_2 \ldots \text{ with } e_0 \text{ a special initial event defining all constant values } c \in C,$$

$$\sigma_0 \text{ the initial state, and } \sigma_k \overset{\mathrm{i_k}}{\rightsquigarrow} \sigma_{k+1}, \text{ where } \forall k \geq 0, i_k = Inst(e_k)$$

When executing a non-branch, non-`return` instruction in a basic block, the next state σ_{k+1} points to the next instruction in the block. Executing a branch instruction makes the next state point to the first instruction of the target block. Executing a `return` instruction makes the next state point to the next instruction following the function call instruction that led to the currently executing function. When executing a function call instruction, the next state σ_{k+1} points to the first instruction of the function's enclosed region.

Unlike a function call, macro expansion takes place in an earlier, offline stage, prior to program execution. The macro's region is expanded in place, with effective arguments substituted in place of the formal ones, renaming the region's local variables and references to avoid conflicts with variables and references of the parent region, and implementing `return` as copying some of the macro's variables into variables defined in the parent. As a result, macro expansion never occurs on a program execution.

Starting from an initial state σ_0, the execution proceeds with calling the special `main` function, taking no argument and returning no value. Instead the program conducts input and output operations through I/O instructions involving the `io` expression. A given program input and output is modeled as a—potentially infinite—list of independent—potentially infinite—sets of values, each set identified with an unique descriptor, the first argument of the `io` expression. Every value in an I/O set is uniquely tagged to distinguish it from any other I/O value from the same set.

The semantics of P is a function from input sets to outputs sets. Given an input I, the semantics of P applied to I is denoted by $C[\![P]\!](I)$, and P produces an unique execution denoted by $\mathcal{E}[\![P]\!](I)$.

The execution of an I/O instruction instantiates an I/O event. Every I/O event reads or writes one or more values. For an I/O event e we note $IO(e)$ its input or output values.

The sets *Inputs* (resp. *Outputs*) represent the sets of all possible inputs of P (resp. outputs of P), and *Executions* is the set of executions produced by P.

Definition 4.5 (I/O ordering). Any pair of distinct events associated with the execution of `io` instructions *with the same descriptor* are ordered by a so-called *I/O ordering relation*, denoted by $\xrightarrow{\text{io}}$. Formally, given an execution $E = \mathcal{E}[\![P]\!](I)$ of P on some input I, $\xrightarrow{\text{io}}$ is the reflexive and transitive closure of the following relation:

$$\forall \ldots e_1 \ldots e_2 \ldots \in E,\ \mathit{Inst}(e_1) = \texttt{io}(\mathit{desc}, \mathit{IO}(e_1)) \\ \wedge\ \mathit{Inst}(e_2) = \texttt{io}(\mathit{desc}, \mathit{IO}(e_2)) \implies e_1 \xrightarrow{\text{io}} e_2$$

This relation on events induces a relation on input and output sets, also denoted by $\xrightarrow{\text{io}}$

$$\forall \ldots e_1 \ldots e_2 \ldots \in E,\ \mathit{Inst}(e_1) = \texttt{io}(\mathit{desc}, \mathit{IO}(e_1)) \\ \wedge\ \mathit{Inst}(e_2) = \texttt{io}(\mathit{desc}, \mathit{IO}(e_2)) \ \wedge\ e_1 \xrightarrow{\text{io}} e_2 \implies \mathit{IO}(e_1) \xrightarrow{\text{io}} \mathit{IO}(e_2)$$

In addition, when a single I/O event reads or writes multiple values, they are ordered from left to right in a given `io` instruction and sequentially over successive `io` expressions associated with the same event.

The $\xrightarrow{\text{io}}$ relation on input and output data models streaming I/O as well as unordered persistent storage in computing systems, and any middle-ground situations such as locally unordered streaming I/O and locally ordered storage operations.

4.3 Program Transformations

Let us first define a notion of program transformation, as general as possible, and without considering validity (correctness) issues for the moment. This notion is inseparable from a mapping that relates semantically connected events across program transformations.

Definition 4.6 (Program transformation). Given a program P, a transformation τ maps P to a transformed program P'. Every transformation τ induces an *event map* $\propto_\tau$ relating some events before and after transformation. The event map notation $e \propto_\tau e'$ reads as "τ maps e to e'" or "e maps to e' through τ", or "e maps to e'" when τ is clear from the context, or "τ preserves e" when the event after transformation does not need to be identified. The mapping is partial and neither injective nor surjective in general, as events in P may not have a semantically relevant counterpart in P' and vice versa.

In the following, we will incrementally construct a $\propto_\tau$ relation for an arbitrary transformation τ. Being a constructive definition, it will serve as a tool to prove the existence, ordering, and properties of values across program transformations.

The set of hypotheses on what is considered a valid program transformation is minimal, covering as many compilation scenarios as possible. This constitutes a major strength of our proposal: we make no assumptions on the analysis and transformation power of a compiler, covering not only the classical scalar, loop and inter-procedural transformations (optimization, canonicalization, lowering), but also hybrid static-dynamic schemes, including control and value speculation. The only constraint on transformations is to preserve the I/O behavior of a program *on all possible inputs*.

Definition 4.7 (Valid program transformation). Given a program P, a program transformation τ that applies to P is valid if it produces a program $P' = \tau(P)$ such that $\forall I \in \mathit{Inputs}, C[\![P]\!](I) = C[\![P']\!](I)$, i.e. P and P' have the same I/O behavior.

The set of all valid transformations of P is denoted by $\mathcal{T}(P)$.

Let us now prove that I/O events as well as their relative ordering are preserved by all valid program transformations. We first introduce a class of events that are always related through $\propto_\tau$ for any valid transformation τ, then prove I/O events belong to this class.

Definition 4.8 (Transformation-preserved event). Given a program P and input I, an event e_{tp} is *transformation-preserved* for execution $\mathcal{E}[\![P]\!](I)$ if all valid program transformations are guaranteed to preserve it. The set of transformation-preserved events for a program P and input I is denoted by $TP(P, I)$. Formally,

$\forall e_{tp} \in \mathcal{E}[\![P]\!](I),\ e_{tp}$ is a transformation-preserved event if and only if

$$\forall \tau \in \mathcal{T}(P),\ \exists e'_{tp} \in \mathcal{E}[\![\tau(P)]\!](I), e_{tp} \propto_\tau e'_{tp}$$

Let us now show that one may construct a $\propto_\tau$ relation that preserves I/O events.

Lemma 4.9 (Unicity of transformed I/O events). *For an execution $E = \mathcal{E}[\![P]\!](I)$ of a program P on some input I, an event e from E reading or writing a value v from/to an input/output set, and a valid program transformation τ, there exists a unique event $e' \in \mathcal{E}[\![P']\!](I)$ such that e' reads or writes v.*

Proof. By definition of transformation validity (Definition 4.7), v also belongs to an input or output set associated with the transformed program $P' = \tau(P)$. As a consequence, $E' = \mathcal{E}[\![P']\!](I)$ also holds an event e' reading or writing v. Since v is uniquely tagged among I/O values, semantical equality $C[\![P]\!](I) = C[\![P']\!](I)$ implies that e' is the only event reading or writing v in the execution E'. □

Definition 4.10 (Preservation of I/O events). For an execution $E = \mathcal{E}[\![P]\!](I)$ of a program P on some input I and a valid program transformation τ, we define $\propto_\tau$ to include all pairs (e, e') such that e is an I/O event in E reading or writing a value *val* from/to an input/output set, and e' is the unique I/O event in $\mathcal{E}[\![P']\!](I)$ such that e' reads or writes *val*.

Lemma 4.11 (Preservation of I/O event ordering). *Any valid program transformation preserves the partial ordering on I/O events.*

Proof. Consider the execution E of a program P on some input I, and a valid program transformation τ.

Given two events e_1 and e_2 in E, each of which is associated with an `io` expression such that $e_1 \xrightarrow{\text{io}} e_2$. From Definition 4.10, there exists two events e'_1 and e'_2 in $E' = \mathcal{E}[\![\tau(P)]\!](I)$ such that $e_1 \propto_\tau e'_1$ and $e_2 \propto_\tau e'_2$. By definition of $\xrightarrow{\text{io}}$ induced by I/O events on input and output sets, any values $v_1 \in IO(e_1)$ and $v_2 \in IO(e_2)$ are such that $v_1 \xrightarrow{\text{io}} v_2$. Since τ is a valid transformation, events e'_1 and e'_2 also have to be ordered such that $v_1 \xrightarrow{\text{io}} v_2$, hence $e'_1 \xrightarrow{\text{io}} e'_2$. □

Finally, one may lift the notion of transformation preservation to a program instruction, collecting events associated all or a subset of the executions of this instruction.

Definition 4.12 (Transformation-preserved instruction). Given a program P, i_{tp} is a *transformation-preserved instruction* of P if all valid program transformations are guaranteed to preserve its associated events, for all inputs. Formally,

$\forall i_{tp} \in P,\ i_{tp}$ is a transformation-preserved instruction if and only if

$$\forall \tau \in \mathcal{T}(P), \forall I \in \mathit{Inputs}, \forall e_{tp} \in \mathcal{E}[\![P]\!](I), i_{tp} = \mathit{Inst}(e_{tp}),\ \exists e'_{tp} \in \mathcal{E}[\![\tau(P)]\!](I), e_{tp} \propto_\tau e'_{tp}$$

And i_{tp} is *transformation-preserved conditionally on the preservation of an instruction* i_c if for all executions of P, the preservation of some event e_c associated with the execution of i_c implies the preservation of any event e_{tp} associated with the execution of i_{tp}. Formally,

$$\forall i_{tp} \in P,\ i_{tp} \text{ is conditionally transformation-preserved on } i_c \text{ if and only if}$$
$$\forall \tau \in \mathcal{T}(P), \forall I \in \mathit{Inputs}, \forall e_{tp} \in \mathcal{E}[\![P]\!](I), i_{tp} = \mathit{Inst}(e_{tp}), \exists e_c \in \mathcal{E}[\![P]\!](I), e'_c \in \mathcal{E}[\![\tau(P)]\!](I),$$
$$e_c \propto_\tau e'_c \wedge \mathit{Inst}(e_c) = i_c \implies \exists e'_{tp} \in \mathcal{E}[\![\tau(P)]\!](I), e_{tp} \propto_\tau e'_{tp}$$

We will use these notions to validate the preservation of security protections, either for all possible executions, or conditionally on the execution of a given secure expression/function.

4.4 Observation Semantics

Let us now consider the last expression in our Mini IR syntax. `snapshot` expressions introduce a specific mechanism to observe values along the execution of the program. Vu et al. [52] defined a *partial observation trace* as a sequence of sets of (variable, value) and (address, value) pairs. To increase the reach of compiler optimizations while preserving the user's ability to attach logical properties to specific values and instructions, we extend the observation semantics to *partially ordered partial states* defined by the execution of instructions involving `snapshot` expressions.

Definition 4.13 (Partial state). Any event involving a `snapshot` expression define a *partial observation state* of the operational semantics, or *partial state* for short. These partial states are modeled by the following *observation function*:

$$\mathit{Obs} : \mathit{Events} \longrightarrow \mathit{States}$$

extracting from an event e a *partial state* $(\mathit{ObsV}, \mathit{ObsC}, \mathit{ObsR}, \mathit{ObsM}, \pi)$ such that π is the program point of the instruction associated with e and $\mathit{ObsV} \subseteq V$, $\mathit{ObsC} \subseteq C$, $\mathit{ObsR} \subseteq R$, $\mathit{ObsM} \subseteq M$ are the $(\mathit{name}, \mathit{value})$ pairs observed by all arguments of `snapshot` expressions involved in event e.

In addition, an individual instruction involving a `snapshot` expression returns all its arguments in addition to capturing these arguments' $(\mathit{name}, \mathit{value})$ pairs into a partial state.

Let us now define *observation events* associated with the execution of instructions involving `snapshot` expressions.

Definition 4.14 (Observation event). We call *observation event* any event associated with the execution of an instruction involving a `snapshot` expression.

The following definitions introduce the observation-ordering relation as a precise tool in the hand of the programmer to define an ordering between observation events; this relation on observation events is derived from def-use, reference-based and in-memory data-flow relations, and control dependences.

Definition 4.15 (Dependence relation). We define relations $\xrightarrow{\text{du}}$, $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{cd}}$ as partial orders on def-use pairs, in-reference/in-memory data flow and control dependences, respectively. Formally, let $\mathit{def}(v, i)$ and $\mathit{use}(v, i)$ be the predicates evaluating to true if and only if instruction i defines variable v and instruction i uses variable v, respectively, and let $\mathit{postdom}$ denote the post-domination

binary predicate:

$$
\begin{aligned}
e_1 \overset{\text{du}}{\rightarrow}{}^{1}\ e_2 \quad & \text{if and only if} \quad def(v, Inst(e_1)) \wedge use(v, Inst(e_2)) \\
e_1 \overset{\text{rf}}{\rightarrow}{}^{1}\ e_2 \quad & \text{if and only if} \quad \big(Inst(e_1) = (ref\ \texttt{<-}\ v) \vee Inst(e_1) = (\texttt{mem[}addr\texttt{]}\ \texttt{<-}\ v)\big) \\
\wedge \quad & \big(Inst(e_2) = (var\ \texttt{=}\ ref) \vee Inst(e_2) = (var\ \texttt{=}\ \texttt{mem[}addr\texttt{]})\big) \\
\wedge \quad & \nexists e_s,\ E = \ldots e_1 \ldots e_s \ldots e_2 \ldots, \\
& \quad \big(Inst(e_s) = (ref\ \texttt{<-}\ v') \vee Inst(e_s) = (\texttt{mem[}addr\texttt{]}\ \texttt{<-}\ v')\big) \\
e_1 \overset{\text{cd}}{\rightarrow}{}^{1}\ e_2 \quad & \text{if and only if} \quad \exists e_s,\ E = \ldots e_1 \ldots e_s \ldots e_2 \ldots, \\
& \quad postdom(Inst(e_2), Inst(e_s)) \wedge \neg postdom(Inst(e_2), Inst(e_1))
\end{aligned}
$$

and

$$
\overset{\text{du}}{\rightarrow} = \left(\overset{\text{du}}{\rightarrow}{}^{1}\right)^{*} \qquad \overset{\text{rf}}{\rightarrow} = \left(\overset{\text{rf}}{\rightarrow}{}^{1}\right)^{*} \qquad \overset{\text{cd}}{\rightarrow} = \left(\overset{\text{cd}}{\rightarrow}{}^{1}\right)^{*}
$$

The dependence relation, denoted by $\overset{\text{dep}}{\rightarrow}$, is defined as the union of the def-use, reference-based and in-memory data-flow, and control dependence relations:

$$
\overset{\text{dep}}{\rightarrow}{}^{1} = \overset{\text{du}}{\rightarrow}{}^{1} \cup \overset{\text{rf}}{\rightarrow}{}^{1} \cup \overset{\text{cd}}{\rightarrow}{}^{1} \qquad \text{and} \qquad \overset{\text{dep}}{\rightarrow} = \left(\overset{\text{dep}}{\rightarrow}{}^{1}\right)^{*}
$$

Definition 4.16 (Observe-from relation). Given an execution E, observation events induce a relation called *observe from* and denoted by $\overset{\text{of}}{\rightarrow}$, mapping a definition to an observation event e_{obs}:

$$
\forall \ldots e_1 \ldots e_{obs} \ldots \in E,\ \left(Inst(e_{obs}) = (var, \_\ \texttt{=}\ \texttt{snapshot(\_,\_)})\ \wedge\ e_1 \overset{\text{du}}{\rightarrow}{}^{1}\ e_{obs}\right) \implies e_1 \overset{\text{of}}{\rightarrow} e_{obs}
$$

Definition 4.17 (Observation ordering relation). Any pair of distinct events associated with the execution of instructions involving `snapshot` expressions related through a dependence relation are ordered by a so-called *observation ordering* relation denoted by $\overset{\text{oo}}{\rightarrow}$. Formally, given an execution E of P, $\overset{\text{oo}}{\rightarrow}$ is the restriction of $\overset{\text{dep}}{\rightarrow}$ to observation events:

$$
\begin{aligned}
& \forall \ldots e_1 \ldots e_2 \ldots \in E, \\
\big(Inst(e_1) = (var_1, \_\ \texttt{=}\ \texttt{snapshot}(v_1, \_))\ \wedge\ Inst(e_2) = (var_2, \_\ \texttt{=}\ \texttt{snapshot}(v_2, \_))\ \wedge\ e_1 \overset{\text{dep}}{\rightarrow} e_2\big) & \\
\implies e_1 \overset{\text{oo}}{\rightarrow} e_2 &
\end{aligned}
$$

We chose to only include data flow relations (through SSA values, references or memory) and control dependences into $\overset{\text{oo}}{\rightarrow}$. This is a trade-off between providing more means to the programmer to constrain program transformations to enforce observation ordering, and freedom left to the compiler in presence of such observations. Data-flow paths between `snapshot` expressions enable the expression of arbitrary partial orders of observation events, and they are easily under the control of programmers—if necessary by inserting dummy or token values as we will see in the next section—hence they appear to be expressive enough for our purpose. Note that control dependences are not directly useful at capturing partial ordering (that would not be otherwise expressible using data dependences), and it is sufficient to *not* make them dependent on the result of `snapshot` expressions to avoid having to unduly constrain the ordering of observations (e.g., forbidding legitimate hoisting of loop-invariant expressions). On the other hand, control dependences are important to model the effect of program transformations converting data dependences into control dependences: for example, boolean logic may be converted into control flow, yielding a single static truth value for some boolean variables occurring in a dependence chain linking two observations. Conversely, adding more relations into $\overset{\text{oo}}{\rightarrow}$ such as non-data-flow write-after-write

(memory-based) dependences would not enhance the ability to represent more partial orders while severely restricting the compiler's ability to reorder loop iterations or hoist observations from loops.

4.5 Happens-Before Relation

We now define a partial order on both I/O and observation events, capturing not only the I/O semantics of the program but also its associated observations.

Definition 4.18 (Happens-before relation). For a given program execution E, one may define a partial order $\xrightarrow{\text{hb}}$ over pairs of events called a *happens-before relation*. It has to be a sub-order of the total order of events in E.

Definition 4.19 (Preservation of happens-before). Given a valid program transformation τ, for any input $I \in \mathit{Inputs}$, P produces an execution $E = \mathcal{E}[\![P]\!](I)$, and the transformed program $P' = \tau(P)$ produces an execution $E' = \mathcal{E}[\![P']\!](I)$. τ is said to preserve the happens-before relation if any events in happens-before relation in P have their counterparts through $\propto_\tau$ in happens-before relation in P'. Formally,

$$\forall e_i, e_j \in E,\ \forall e'_i, e'_j \in E',\ e_i \xrightarrow{\text{hb}} e_j \ \wedge\ e_i \propto_\tau e'_i \ \wedge\ e_j \propto_\tau e'_j \implies e'_i \xrightarrow{\text{hb}} e'_j$$

The preservation of the happens-before relation is a property that has to be proven in general. Depending on how sparse the $\propto_\tau$ and $\xrightarrow{\text{hb}}$ relations are, it may be more or less difficult to enforce and establish. In the following, we use the following happens-before relation:

$$\xrightarrow{\text{hb}} = \left(\xrightarrow{\text{io}} \cup \xrightarrow{\text{of}} \cup \xrightarrow{\text{oo}} \right)^*$$

Thanks to Lemma 4.11 one will only have to prove the preservation of the $\xrightarrow{\text{of}}$ and $\xrightarrow{\text{oo}}$ components of $\xrightarrow{\text{hb}}$ in the following. On the contrary, unlike I/O instructions, instructions involving `snapshot` are *not* preserved by valid program transformations in general.

Let us now provide two important definitions to reason about the preservation of observations.

Definition 4.20 (Observation-preserving transformation). Given a program P, a transformation τ that applies to P is *observation-preserving* if it produces a program $P' = \tau(P)$ such that the four following conditions hold:

(i) it is a valid transformation (see Definition 4.7);

(ii) it preserves the existence of observation events:

$$\forall I \in \mathit{Inputs}, \forall e \in \mathcal{E}[\![P]\!](I),\ \mathit{Inst}(e) = (\mathit{var}, \_ \texttt{ = snapshot}(v, \_)) \implies \exists e' \in \mathcal{E}[\![P']\!](I),\ \mathit{Inst}(e') = (\mathit{var}', \_ \texttt{ = snapshot}(v', \_)) \ \wedge\ e \propto_\tau e'$$

(iii) it preserves all happens-before relations:

$$\forall I, \forall e_1, e_2 \in \mathcal{E}[\![P]\!](I),\ e_1 \xrightarrow{\text{hb}} e_2 \implies \exists e'_1, e'_2 \in \mathcal{E}[\![P']\!](I), e'_1 \xrightarrow{\text{hb}} e'_2$$

(iv) it preserves the observed values:

$$\forall I \in \mathit{Inputs}, \forall e \in \mathcal{E}[\![P]\!](I),\ \mathit{Inst}(e) = (\mathit{var}, \_ \texttt{ = snapshot}(v, \_)), e' \in \mathcal{E}[\![P']\!](I),\ \mathit{Inst}(e') = (\mathit{var}', \_ \texttt{ = snapshot}(v', \_)) \ \wedge\ e \propto_\tau e' \implies \mathit{Obs}(e) = \mathit{Obs}(e')$$

Given a program P, a transformation τ that applies to P is *observation-preserving conditionally on* instruction i_c in P if it produces a program $P' = \tau(P)$ such that the conditions (i), (iii) and (iv) above hold, and also:

(ii_c) it preserves the existence of observation events conditionally on the preservation of i_c:

$$\begin{aligned}\forall I \in \mathit{Inputs}, \forall e \in \mathcal{E}[\![P]\!](I),\ \mathit{Inst}(e) = (\mathit{var}, \_ \texttt{ = snapshot}(v, \_)) \wedge \\ \exists e_c \in \mathcal{E}[\![P]\!](I), e'_c \in \mathcal{E}[\![P']\!](I), \mathit{Inst}(e_c) = i_c, e_c \propto_\tau e'_c \\ \implies \exists e' \in \mathcal{E}[\![P']\!](I),\ e \propto_\tau e'\end{aligned}$$

Let us now define a notion of observation that is preserved over all possible valid transformations.

Definition 4.21 (Protected observation). An observation in a program P is *protected* if and only if all valid transformations that apply to P are observation-preserving.

An observation is *protected conditionally on instruction* i_c if and only if all valid transformations are observation-preserving conditionally on i_c.

As expected, preservation (resp. protection) imply conditional preservation (resp. protection) on all instructions.

Note that the composition of two valid transformations yields a valid transformation, according to Definition 4.7. As a consequence, Definition 4.21 covers compositions of valid transformations along a compilation pass pipeline.

In the next section, we will provide a constructive method to implement programs with *protected observations* complying with Definition 4.21. This will allow us to prove the preservation of $\overset{\text{hb}}{\longrightarrow}$ on a class of programs with carefully defined protections of `snapshot` expressions, as a partial fulfillment of the requirements for a valid program transformation to be observation-preserving.

5 VALUE PRESERVATION MECHANISMS

As noted earlier, valid transformations do not preserve the happens-before relation in general. This section introduces a mechanism to achieve this, involving a minor extension of the Mini IR with expressions that are defined to be opaque to any program analysis.

5.1 Opaque Expressions

extended-expr	::=	*expr*	
	|	`opaque` *region*	atomic opaque region: make I/O, side-effects and definitions visible atomically outside the region and vice versa; the compiler sees statically unknown yet functionally deterministic values
	|	`yield(` *var** `)`	return from atomic opaque region

Fig. 2. Extension of Mini IR to implement event and happens-before preservation.

To implement the preservation of observation events and the associated happens-before relation, we extend Mini IR with an *opaque expression* syntax. The `opaque` keyword introduces a region of control flow, as shown in Figure 2. The opaque expression syntax gets its name from the "opacity" of its enclosed region w.r.t. program analyses and transformations. An instruction defining a value from an opaque expression is called an *opaque instruction*.

When executing `opaque`, the associated event gathers the definitions and effects of all instructions in its enclosed region, *atomically and in isolation*. We assume the enclosed region is a *terminating* sequence of instructions. It proceeds with "internal" state transitions without defining events and without exposing intermediate states. When reaching a `yield` instruction, the program state

serves as the resulting state of the atomic event while also defining all values listed in the `yield` instruction. Formally, executing an opaque instruction enclosing a region $\{i_1; \ \ldots; \ i_n\}$ on a state σ yields the event $e = \sigma \overset{a}{\rightsquigarrow} \sigma'$ where $\sigma \overset{i_1}{\rightsquigarrow} \cdots \overset{i_n}{\rightsquigarrow} \sigma'$. The last instruction i_n must be a `yield` instruction. We authorize arbitrary (terminating) control flow in these regions, including conditional memory access and I/O. As a result, the memory and I/O effects triggered by an opaque instruction are input-dependent. Since regions inside opaque expressions always terminate, σ' always exists. This definition guarantees both atomicity and isolation, since states and transitions associated with individual instructions $\{i_k\}_{1 \leq k \leq n}$ are not modeled in the operational semantics. Finally, the semantics of nested opaque expressions is defined inductively from the inside out.

The compiler is very limited in what analyses it may perform on opaque expressions:

- gathering the uses of an opaque expression;
- deciding whether an opaque expression has read or write side-effects;
- deciding whether an opaque expression performs I/O;
- deciding whether two opaque expressions are identical up to variable renaming.

Yet the compiler is not allowed to determine the precise side-effects (references, memory addresses) in an opaque expression, and it may not attempt to establish a correlation between its uses (resp. loads from references or memory) and the values it defines (resp. stores to references or memory).

On the other hand, a valid transformation τ may also delete or duplicate an opaque instruction, and even synthesize completely new ones. This may sound too powerful, as without additional care τ may break the opacity and expose intermediate states in an opaque expression. We will see in the following that the opacity property itself prevents this from happening, thus maintaining opacity, atomicity and isolation of opaque expressions across transformations.

Since opaque expressions can nest multiple instructions (and even nested regions), we introduce a notation to denote the sequence of instructions executed atomically within an event. Let $\mathit{InstList}(e)$ denote the list of instructions associated with event e; it is a single-element list for all events, except for those associated with opaque instructions where it is the sequence of instructions executing within the region for this particular instance e of the opaque instruction. We will write $i \in \mathit{InstList}(e)$ to denote that an instruction i is associated with event e.

In the rest of the paper, we will use revised and extended versions of Definitions 4.5–4.20 operating on *sets of instructions* in $\mathit{InstList}(e)$ rather than a specific instruction $\mathit{Inst}(e)$. All equalities of the form $i = \mathit{Inst}(e)$ in these equations should be rewritten into $i \in \mathit{InstList}(e)$. For convenience, we will also consider all I/O expressions as being opaque; this is consistent with the traditional assumptions about compilers not being able to analyze across system calls.

Informally, opaque expressions have two important consequences on value and event preservation across program transformations: (1) if a valid program transformation preserves an event using a value defined by an opaque instruction or stored by an instruction from its associated region, then it must also preserve the event associated with the opaque instruction, and (2) a valid program transformation has to preserve any value used in the opaque expression, as proceeding with downstream computation would otherwise involve some form of unauthorized guessing of the opaque expression's behavior. Formally, let the predicate $\mathit{Opaque}(e)$ denote that event e is associated with the execution of an opaque instruction; we restrict the effects of program transformations in presence of opaque expressions as follows:

Definition 5.1 (Opaque expression preservation). Given a program P, input I, and valid transformation τ,

$$\forall \ldots e_1 \ldots e_2 \ldots \in \mathcal{E}[\![P]\!](I),\ Opaque(e_1),\ e_1 \overset{\text{dep}}{\rightarrow}{}^{1} e_2,$$
$$\exists e_2' \in \mathcal{E}[\![\tau(P)]\!](I),\ e_2 \propto_\tau e_2' \implies \exists e_1' \in \mathcal{E}[\![\tau(P)]\!](I),\ e_1 \propto_\tau e_1' \wedge Opaque(e_1') \wedge e_1' \overset{\text{dep}}{\rightarrow} e_2' \quad (1)$$

$$\forall \ldots \sigma_e \ldots \in \mathcal{E}[\![P]\!](I),\ \forall \ldots \sigma_{e'}' \ldots \in \mathcal{E}[\![\tau(P)]\!](I),\ Opaque(e),\ e \propto_\tau e',$$
$$use(v, Inst(e)) \wedge (v, val) \in \sigma_e \implies \exists i' \in InstList(e'), use(v', i') \wedge (v', val) \in \sigma_{e'}' \quad (2)$$

This restriction is taken as a definition, capturing formally the intuitive expectations about what the compiler has to enforce in the presence of opaque expressions.

Notice the transitive dependence relation $e_1' \overset{\text{dep}}{\rightarrow} e_2'$ in the transformed program (rather than $e_1' \overset{\text{dep}}{\rightarrow}{}^{1} e_2'$): the immediate dependence may be transformed into a series of instructions (e.g., spilling a value to the stack).

Let us highlight a subtle point in this definition: i' is an instruction belonging to the transformed opaque expression's region, not the opaque instruction itself. Indeed, variable v' may not be a free variable in $Inst(e')$, it may be bound to the opaque expression's internal region. For example, it is always correct to transform `opaque { some_use_of(v) }` into the sequence `t1 = not v; opaque { t2 = not t1; some_use_of(t2) }`. This does not involve any analysis of the opaque expression's semantics—which is explicitly disallowed. While `t2` remains part of the program state and retains the value `v` had in the original program, it is not exposed as a variable captured by the opaque expression.

In the following, we will only use `snapshot` within the region of an opaque expression. As a result, `snapshot` expressions will inherit all properties of opaque expressions, including the conditions for their preservation (1) and the preservation of observed values (2).

5.2 Opaque Chains

Let us now build dependence chains involving opaque instructions. These will be called *opaque chains* and serve two purposes: (1) establishing a transformation-preserved $\overset{\text{oo}}{\rightarrow}$ relation, and (2) linking observations to downstream I/O events to preserve the former through program transformations. We first need additional definitions and notations.

Definition 5.2 (Opaque value set). Given an execution $E = \mathcal{E}[\![P]\!](i)$, consider a chain of dependent events $e_1 \overset{\text{dep}}{\rightarrow}{}^{1} \ldots \overset{\text{dep}}{\rightarrow}{}^{1} e_n$ with $n \geq 2$, an opaque instruction/event $i_j = Inst(e_j)$ on the chain defining a variable var_j and an instruction/event $i_k = Inst(e_k)$ on the chain, with $1 \leq j < k \leq n$ and $\forall j < l < k, \neg Opaque(e_l)$. Let σ_j be the program state e_j transitions into.

OV_j denotes the set of all opaque values that var_j may take according to its data type: for example, an opaque expression yielding a value of boolean type will have $OV_j = \{\text{true}, \text{false}\}$.

We also lift this definition to the set of values used by a downstream expression across a chain of dependent instructions. Consider a value $alt \in OV_j$, and an execution E_{alt} continuing after e_j on program state $\sigma_j\{var_j \mapsto alt\}$.[1] Consider an event $e_{k_{alt}}$ such that $e_j \overset{\text{dep}}{\rightarrow} e_{k_{alt}}$ and $E_{alt} = \ldots e_j \ldots e_{k_{alt}} \ldots$. The function $value_{j,k}$ maps every value $alt \in OV_j$ to a value defined as follows:

[1]The value alt may be identical to the original value of var_j in E, or any alternative value in OV_j. The substitution syntax $\sigma_j\{var_j \mapsto alt\}$ denotes the set $\sigma_j \setminus (var_j, \_) \cup (var_j, alt)$.

- $value_{j,k}(alt)$ is the value used or read by $i_{alt} = Inst(e_{k_{alt}})$ along the $e_j \dots e_{k_{alt}}$ sub-chain if i_k and i_{alt} are identical expressions up to variable renaming, and $\forall e \neq e_{k_{alt}}$ s.t. $e_j \overset{\text{dep}}{\rightarrow} e \overset{\text{dep}}{\rightarrow} e_{k_{alt}}$, $\neg Opaque(e)$;
- $value_{j,k}(alt)$ is the special value $\perp$ otherwise.

We define the opaque value set $OV_{j,k}$ as $value_{j,k}(OV_j)$.

Intuitively, the definition of $value_{j,k}$ allows to reason on the cardinality of the opaque value set $OV_{j,k}$: whether this set is a singleton or not will tell whether the dependent instruction/event $i_k = Inst(e_k)$ is truly sensitive on the opaque value of $i_j = Inst(e_j)$. A non-singleton set tells that evaluating the opaque instruction i_j cannot be avoided by a valid transformation. Let us paraphrase this definition to expose the intuitions behind it. When substituting the value *alt* for the opaque expression i_j, the $value_{j,k}$ function yields the value used or read by i_k if the execution path of the altered execution still reaches i_k. If the execution path is altered when considering the *alt* value and encounters an identical instruction/event $i_{alt} = Inst(e_{k_{alt}})$ before encountering a dependent opaque instruction, it yields the value used or read by i_{alt} (this accounts for program transformations capable of combining two identical instructions, more on this later). And if the altered execution reaches an opaque instruction before reaching an identical instruction, $value_{j,k}$ yields the "undefined" value $\perp$.

Definition 5.3 (Opaque chain). Given an execution $E = \mathcal{E}[\![P]\!](i)$, consider a chain of dependent events $e_1 \overset{\text{dep}}{\rightarrow}{}^1 \dots \overset{\text{dep}}{\rightarrow}{}^1 e_n$; $e_1 \overset{\text{dep}}{\rightarrow}{}^1 \dots \overset{\text{dep}}{\rightarrow}{}^1 e_n$ is an *opaque chain* linking e_1 to e_n if and only if

(i) $i_1 = Inst(e_1)$ and $i_n = Inst(e_n)$ are opaque instructions;
(ii) for $2 \leq k \leq n$, an *opaque* instruction $i_k = Inst(e_k)$, and $1 \leq j < k$ the immediately preceding opaque instruction on the chain, $\text{Card}(OV_{j,k}) \geq 2$.

We note $e_1 \overset{\text{opaque}}{\rightsquigarrow} e_n$ such an opaque chain.

The intuition behind the (ii) condition is the following. Given an instruction/event $i_k = Inst(e_k)$, for the immediately upstream opaque instruction/event $i_j = Inst(e_j)$ on the chain, we consider all values it may define according to its opaque result type. Either i_k is control-dependent on i_j and there exists an alternate execution from e_j bypassing i_k (or an identical expression up to variable renaming), or i_k is data-dependent on i_j and the set of values i_k may use or read is not a singleton, or both.

Opaque chains take the form of an alternating sequence of opaque instructions and sub-chains of regular instructions, starting with an opaque instruction and ending with an opaque instruction (remember I/O expressions are considered opaque). The control- and data-dependence restrictions in Case (ii) serve as "information-carrying" guarantees: the compiler does not have enough information about the possible paths dependent on an opaque value or on the processing of opaque values to break an opaque chain into distinct dependence chains.

Let us consider a few examples of dependence chains that are not opaque chains. First, considering data dependences only, shifting an opaque `uint32_t` by 32 bits to the right would allow the compiler to reason about the resulting zero value, transforming the downstream opaque expression into one applied to the constant zero, hence breaking the chain. The cardinality requirement on data dependences rules such transformations out. As a more complex example, let us illustrate the restrictions on the multiplicity of paths leading and not leading to i_k or an identical opaque expression. Consider the following program

```
1 bb_entry:
2   c = opaque {
3     yield(42);
```

```
  };
  br c, bb_true;
bb_false:
  io(desc, 0);
  br true, bb_join;
bb_true:
  io(desc, 0);
bb_join:
```

forming a dependence chain from the definition of `c` to the I/O instruction—through a control dependence. And consider its transformation into

```
bb_entry:
  c = opaque {
    yield(42);
  };
  io(desc, 0);
```

As a result of *instruction combining*, there is no dependence anymore in the transformed program. The dependence chain in the original program is not an opaque one due to the identical values (constant 0) read by the I/O instruction on both paths leading to an identical instruction.

On the contrary, the following example illustrates the conversion of control into data dependences and vice-versa, preserving opaque chains in the process: consider the programs P_{data}

```
bb_entry:
  c = opaque {
    u = snapshot(boolean_input);
    yield(u);
  };
  br c, bb_true;
bb_false:
  v = 0;
  br true, bb_join;
bb_true:
  v = 42;
bb_join:
  opaque {
    snapshot(v);
  };
```

and P_{control}

```
bb_entry:
  c = opaque {
    u = snapshot(boolean_input);
    yield(u);
  };
  br c, bb_true;
bb_false:
  opaque {
    snapshot(0)
  };
  br true, bb_join;
bb_true:
  opaque {
    snapshot(42);
  };
bb_join:
```

Both form opaque chains from the definition of `c` to the observation of `v`, `42`, `0`; the transformations from P_{data} to P_{control} and vice-versa are both valid, preserving observations (a data dependence is converted into a control dependence, specializing values into constants, and vice-versa for the reverse transformation). Whether it is the multiple values of `v` or the alternative path from the definition of `c` to a consuming `snapshot`, it is impossible for the compiler to break the dependence.

Beyond opaque instructions, important classes of instructions always belong to an opaque chain:

- all instructions that only propagate existing values within or across name-value domains; these include dereference, assignment, load, store, `br` on branch arguments (not on the branch condition), `call` and `return` instructions, and instructions involving `snapshot` or `io` expressions;
- the same applies to the traditional C unary operators `-`, `!`, `~`;
- any binary operator (resp. function call) where one or more operands (resp. arguments) are opaque, and where opaque operands (resp. arguments) are not correlated (feeding multiple times the same opaque value or dependent expressions may degenerate into a singleton value set, such as the subtraction of an opaque value with itself);
- any binary operator (resp. function call) where the operand (resp. arguments) type or the value of the other operand (resp. other arguments) makes the operation bijective; e.g., `+` on unsigned integers, `*` with the constant 1, etc.

More instructions may belong to an opaque chain provided specific constraints hold on its inputs: e.g., left-shifting by 1 an `unsigned int` value if the compiler cannot prove it is always greater than or equal to `UINT_MAX/2`, or dividing a value that the compiler cannot statically analyze to be less than the divisor; in both cases the compiler is forced to consider that the image of the instruction on all possible inputs is not a singleton.

We may now generalize Equation (1) to opaque chains. If a valid program transformation preserves an event at the tail of an opaque chain, then it must also preserve the event associated with the head of the opaque chain. Formally:

Lemma 5.4 (Chained opacity). *Given a program P, input I, and valid transformation τ,*

$$\forall \dots e_1 \dots e_n \dots \in \mathcal{E}[\![P]\!](I),\ e_1 \overset{\text{opaque}}{\rightsquigarrow} e_n, \wedge\ \exists e'_n \in \mathcal{E}[\![\tau(P)]\!](I), e_n \propto_\tau e'_n$$
$$\implies \exists e'_1 \in \mathcal{E}[\![\tau(P)]\!](I),\ e_1 \propto_\tau e'_1\ \wedge\ \mathit{Opaque}(e'_1)\ \wedge\ e'_1 \overset{\text{dep}}{\rightarrow} e'_n \quad (3)$$

Proof. The $\overset{\text{opaque}}{\rightsquigarrow}$ relation implies e_1 is opaque. If $n = 1$, the result stems from the application of Equation (1) to e_1. Consider the case of $n > 1$. The value used by e_n is sensitive to the value defined by e_1, implementing a non-constant function. Since the instructions in our Mini IR capable of implementing a non-constant function are the definition, load, store and conditional branches, this implies the existence of a slice of events in P, spawning backward from e_n; in addition e_1 belongs to this backward slice. These defintion, load, store, branch instructions are exactly those building up $\overset{\text{dep}}{\rightarrow}$. As a result, from Equation 2, e_n being opaque, the mapping of e_n to e'_n implies that the same value computed by a non-constant function is used by e'_n. In P', this non-constant function is computed by a slice in $\tau(P)$ spawning backward from e'_n. The backward slice in P includes one or more instructions depending on the result of e_1, including e_2. This is also the case in the transformed program: $\mathcal{E}[\![\tau(P)]\!](I)$ holds an event e' that uses the opaque value defined by e_1. Either $\mathit{Inst}(e')$ is an opaque instruction lumping together $\mathit{Inst}(e_1)$ with additional instructions including the use of the opaque value defined by e_1 and defining $e'_1 = e'$ and $e_1 \propto_\tau e'_1$ concludes the proof, or one may define $e_2 \propto_\tau e'$ and apply Equation (1) again. □

Let us now introduce an important lemma on the preservation of opaque expressions.

Lemma 5.5 (Preservation of opaque chains). *Given a program P and input I, if e_1 is linked through an opaque chain to a transformation-preserved event e_n, then e_1 is transformation-preserved, and for any transformation τ mapping e_1 to e_1' and e_n to e_n', there is a chain of dependent instructions linking e_1' to e_n'. Formally,*

$$e_1 \overset{\text{opaque}}{\rightsquigarrow} e_n \ \wedge e_n \in TP(P,I) \implies \forall \tau \in \mathcal{T}(P),\ \exists e_1', e_n' \in \mathcal{E}[\![\tau(P)]\!](I),\ e_1 \propto_\tau e_1' \ \wedge \ e_1' \overset{\text{dep}}{\rightarrow} e_n'.$$

Proof. Consider an opaque chain $e_1 \overset{\text{opaque}}{\rightsquigarrow} e_n$. The existence of e_n' derives from the hypothesis that e_n is transformation-preserved.

The case of $n = 1$ also stems immediately from the same hypothesis that e_n is transformation-preserved.

Consider a chain of $n > 1$ events $e_1 \overset{\text{opaque}}{\rightsquigarrow} e_n$. Let $i_n = Inst(e_n)$, and consider the last opaque instruction/event $i_k = Inst(e_k)$ before e_n. Applying Equation (3) to e_k implies the existence of e_k' such that $e_k \propto_\tau e_k'$ and $e_k' \overset{\text{dep}}{\rightarrow} e_n'$. Induction on the length of the chain proves the preservation of e_1 through valid transformations for all chain lengths, and that the transformed events form a dependence chain $e_1' \overset{\text{dep}}{\rightarrow} e_n'$. □

Notice that a typical case of transformation-preserved e_n is an I/O event, but the lemma is not limited to opaque chains terminating in a consuming I/O event. Notice also that events associated with non-opaque expressions along the chain are not necessarily preserved.

Unfortunately, despite our efforts to make the definition as robust to transformations as possible, we have not been able to prove that an opaque chain transforms into an opaque chain in general. Additional hypotheses on opaque expressions are likely to be needed to prove this, and we conjecture these would not require modifying our definition of opaque chains, but we do not have a definite answer at this point. Fortunately, we do not need to prove such a strong preservation result: a weaker-yet-sufficient compositionality result can be used as a work-around.

Lemma 5.6 (Transitive preservation of opaque chains).

$$e_1 \overset{\text{opaque}}{\rightsquigarrow} e_n \ \wedge \ e_n \in TP(P,I)$$
$$\implies \forall \tau \in \mathcal{T}(P), \forall \tau' \in \mathcal{T}(\tau(P)),\ \exists e_1'', e_n'' \in \mathcal{E}[\![\tau'(\tau(P))]\!](I),\ e_1 \propto_{\tau' \circ \tau} e_1'' \ \wedge \ e_1'' \overset{\text{dep}}{\rightarrow} e_n''$$

Proof. The proof of transitive transformation preservation stems from the observation that $\tau \circ \tau'$ is a valid transformation and the application of Lemma 5.5. □

We will use Lemma 5.6 to reason about the preservation of opaque events with a dependent transformation-preserved event (e.g. an I/O instruction) throughout the compilation flow.

5.3 Syntactic Sugar

Let us now introduce a simple pattern implementing a "tokenizing" opaque expression.

```
macro token(v1, ..., vk) { // pure, opaque unit-type value,
                           // not associated with any resource;
                           // the compiler sees a statically unknown
                           // yet functionally deterministic value
bb_macro:
  w = opaque {
    use(v1, ..., vk);
    yield(unit_value);
  };
  return(w);
}
```

where the variadic `use` function uses all its arguments and returns no value.

It is pure, functionally deterministic, and opaque to the compiler. Semantically, the `token` pattern returns a value of the unit type—called a token—irrespectively of the number of arguments. Yet it is not known to the compiler what values a token can take, and in particular the compiler is not told it is a singleton set. As a result one may use `token` as an opaque expression, and to use the token type in dependent instructions forming an opaque chain. Unlike more general opaque expressions, token values do not need hardware resources to store live token values when emitting assembly code: we refer to the unique value `unit_value` of the unit-type to implement "resource-less" opaque chains of tokens.

It may sound paradoxical to return the same `unit_value` in multiple tokens, yet this is not visible to the compiler since it occurs in an opaque expression; the compiler must assume these are different, unpredictable values.

Furthermore, since the unit data type does not carry an informative value, we define `snapshot` to ignore its token arguments, i.e., not to embed them into a partial state.

Finally, we introduce a `tailio` pattern implementing a special case of the `io` instruction with no argument. It is used in opaque chains to prevent them from being eliminated.

```
macro tailio() {
bb_macro:
  desc = tagged_unit_unordered_set_descriptor;
  io(desc);
  return();
}
```

The `tailio` pattern uses a dedicated descriptor of an unordered set of (tagged) unit-valued outputs. Since it embeds an I/O instruction, it is preserved by transformations, yet unlike the more general form of I/O it does not incur any ordering constraint.

5.4 Robust Partial State Observation

Based on the extended syntax, we may now define a series of observation patterns, defining partial states preserved over program transformations. They are ordered by increasing degrees of freedom for the scheduler and optimizations in general.

Monolithic opaque expression. The simplest form of observation preserved over program transformations.

```
macro observe_monolithic(v1, ..., vk) {
bb_macro:
  t2 = opaque {
    w1, ..., wk = snapshot(v1, ..., vk);
    t1 = token(w1, ..., wk);
    tailio();
    yield(t1);
  };
  return(t2);
}
```

Lemma 5.7 (Preservation of monolithic observation events). *If (1) all instructions involving* `snapshot` *expressions of a program P are introduced via* `observe_monolithic`*, and if (2) the happens-before relation on observations in P is enforced through an opaque chain, then observations in P are protected according to Definition 4.21.*

Proof. Consider any valid transformation τ. We need to prove that τ preserves observations, according to Definition 4.20.

The opaque expression's atomic region expands a nested `tailio` pattern. As a result, Definition 4.10 guarantees that all events associated with the execution of such an opaque expression are transformation-preserved. This proves conditions (i) and (ii) in Definition 4.20.

Any valid transformation must preserve the value of (token) `t1` from the definition of opaque expressions.[2] The opaque expression's region holds a backward slice linking atomically the (token) value `t1` to the arguments `v1`, ..., `vk` to be observed. By equation (2) in Definition 5.1, any valid transformation must preserve the values of `v1`, ..., `vk` along with the definition events producing these values. This proves condition (iv) in Definition 4.20.

Let us now prove condition (iii)—the preservation of $\overset{\text{hb}}{\longrightarrow}$. The preservation of $\overset{\text{of}}{\longrightarrow}$ follows the same reasoning as the proof of condition (iv); yet the preservation of $\overset{\text{oo}}{\longrightarrow}$ involves the traversal of opaque chains. Consider a pair of events e_{obs_1}, e_{obs_2}, each of which is associated with the opaque expression expanded from `observe_monolithic`, such that $e_{obs_1} \overset{\text{oo}}{\longrightarrow} e_{obs_2}$, and let e'_{obs_1}, e'_{obs_2} be their counterparts in $P' = \tau(P)$; these transformed events must exist as they involve I/O effects. Since $e_{obs_1} \overset{\text{opaque}}{\rightsquigarrow} e_{obs_2}$, Lemma 5.5 applies to all `opaque` events on the opaque chain, guaranteeing their mapping to events in P' through τ, and that these events form a dependence chain. This proves $e'_{obs_1} \overset{\text{oo}}{\longrightarrow} e'_{obs_2}$. □

Notice that some arguments of `observe_monolithic` associated with e_{obs_2} may be converted by τ into constants, which are defined at the initial event e_0. Still, at least one argument will remain the target of the opaque chain linking e_{obs_1} to e_{obs_2}.

Decoupled I/O region. Cutting out a specific I/O section allows to decouple event preservation from event ordering. Rather than inserting an I/O instruction in every observation, it is sufficient to consider a smaller set, or even a single I/O instruction, and to chain this set backwards to the observations whose events they are meant to preserve.

```
macro observe_decoupled(v1, ..., vk) {
bb_macro:
  u1 = opaque {
    w1, ..., wk = snapshot(v1, ..., vk);
    t1 = token(w1, ..., wk);
    yield(t1);
  };
  return(u1);
}

macro observe_tailio(u1, ..., uk) {
bb_macro:
  v = opaque {
    u = token(u1, ..., uk);
    tailio();
    yield(u);
  };
  return(v);
}
```

The programmer may use `observe_decoupled` to organize observations, setting partial ordering constraints among them using the pattern's resulting token. These will be much less intrusive to compiler transformations than `observe_monolithic` and its embedded I/O instruction.

We can now adapt the Lemma 5.7 to the decoupled preservation pattern.

[2]Remember the compiler assumes tokens can have multiple values, even if these do not consume any resources.

Lemma 5.8 (Preservation of decoupled observation events). *If (1) all* `snapshot` *instructions of a program P are introduced via* `observe_decoupled`*, if (2) the happens-before relation on observations in P is enforced through opaque chains whose tails are I/O events introduced via* `observe_tailio`*, then observations in P are protected according to Definition 4.21.*

Proof. Similarly to Lemma 5.7, we need to prove that any valid transformation τ preserves decoupled observation event, according to Definition 4.20.

Since `observe_tailio` includes a `tailio` instruction, Definition 4.10 guarantees that events associated with instances of the opaque expression holding the `tailio` instruction are transformation-preserved. Lemma 5.5 applied to opaque chains linking `observe_decoupled` opaque expressions to a subsequent `observe_tailio` proves conditions (i) and (ii) in Definition 4.20.

The proof of condition (iii)—the preservation of $\overset{\text{hb}}{\longrightarrow}$—in Lemma 5.7 applies to `observe_decoupled` as it does not refer to the `tailio` instruction.

The proof of condition (iv) is also identical to the proof of Lemma 5.7. □

`tailio` or `observe_tailio` typically occur at the tail of an opaque chain, and a single `tailio`/`observe_tail` may close multiple opaque chains.

5.5 I/O-Based Schemes for Partial State Observation

Let us now use our extended syntax to implement the functional property preservation mechanism described in [52]. The authors define the functional property preservation as the preservation of (1) values occurring in the property predicates and (2) the program observation point at which the property is evaluated. As such, the original mechanism is divided in two parts: Vu et al.'s algorithm inserts an artificial definition for every definition reaching a functional property, which corresponds to inserting the following `artificial_def` pattern, and an observation for the functional property itself, implemented with the following `observe_cc` pattern.

```
1  macro artificial_def_cc(v) {
2  bb_macro:
3    u = opaque {
4      desc = ordered_set_descriptor;
5      io(desc);
6      yield(v);
7    };
8    return(u);
9  }
10
11 macro observe_cc(u1, ..., uk) {
12 bb_macro:
13   opaque {
14     w1, ..., wk = snapshot(u1, ..., uk);
15     desc = ordered_set_descriptor;
16     io(desc);
17     yield();
18   };
19 }
```

Using the first pattern, every value referred in the property's predicate is protected by the compiler using an opacification mechanism. In the original approach, these opaque values next replace the original ones in the subsequent code. Finally, the second pattern snapshots the opaque values at the observation point. Notice that unlike our lightweight approach, all opaque expressions used for property preservation contain an I/O instruction inducing a total order. This total ordering

constraint is cumbersome, and not always wanted by the programmer but rather a downside of the approach.

5.6 Observing Address-Value Pairs

When observing values in memory, several applications require observing not only the value but also the memory address that holds this value. For example, this is important when assessing the proper erasure of a buffer in memory, to avoid leaking sensitive data. The following pattern provides such a functionality, when associated with an `observe_tailio` pattern as in the decoupled or fine-grained schemes above.

```
macro observe_pair(a) {
bb_macro:
  u = opaque {
    v = mem[a];
    b, w = snapshot(a, v);
    t = token(b, w);
    yield(t);
  };
  return(u);
```

The observation of both address `a` and the value stored at this location `a` occurs atomically w.r.t. other observations since they belong to the same partial state. One may use `observe_pair` to check the value at a specific memory address, as required by the abovementioned memory erasure example.

Of course, one may also define a version of this pattern for the monolithic scheme, and versions with a variable number of address-value pairs.

5.7 Value-Preserving Observation-Opacification Pattern

We will see in Sections 6 and 7 that a common pattern to protect secure implementations consists in opacifying specific values, without modifying the data or control flow. In particular, rather than building an opaque chain of tokens—like `observe_decoupled` does—it is natural to chain observations using the original values but hiding them from potentially harmful transformations.

```
macro observe_and_opacify(v1, ..., vk) {
bb_macro:
  u1 = opaque {
    w1, ..., wk = snapshot(v1, ..., vk);
    yield(w1);
  }
  return(u1);
}
```

The `observe_and_opacify` pattern implements the identity function on its first argument. Its other arguments can be used to express data dependence relations. In addition, all arguments are observed. The compiler sees the result of `observe_and_opacify` as a statically unknown yet functionally deterministic value. Any observation introduced via `observe_and_opacify` is called *opacification*.

Notice that `observe_and_opacify` only differs from `observe_decoupled` in the returned value (the first argument rather than a token). As a consequence, the proof of Lemma 5.8 applies directly to the following value-preserving observation and opacification result.

Lemma 5.9 (Preservation of observation-opacification events). *If (1) all* `snapshot` *instructions of a program P are introduced via* `observe_and_opacify`*, if (2) the happens-before relation*

on opacifications in P is enforced through opaque chains whose tails are I/O events introduced via `observe_tailio`*, then observations in P are protected according to Definition 4.21.*

5.8 Protecting Observations and Logical Properties

Collecting the results in this section, the following theorem provides a methodology for the observation and opacification of any value(s) or address-value pair(s), the enforcement of any partial ordering among these observations, and the preservation of both observations and their partial ordering.

Theorem 5.10 (Observation protection). *Let P be a program implementing observations through a combination of* `observe_monolithic`*,* `observe_decoupled` *and* `observe_and_opacify` *on opaque chains enforcing a programmer-specified $\xrightarrow{\text{hb}}$ order, and such that any chain involving* `observe_decoupled` *and* `observe_and_opacify` *leads to a downstream transformation-preserved instruction (such as a trailing* `tailio`*). Then all observations in P are protected according to Definition 4.21.*

Let P be a program implementing observations through a combination of `observe_monolithic`*,* `observe_decoupled` *and* `observe_and_opacify` *on opaque chains enforcing a programmer-specified $\xrightarrow{\text{hb}}$ order, and such that any chain involving* `observe_decoupled` *and* `observe_and_opacify` *leads to a downstream instruction transformation-preserved conditionally on some instruction in a set I_c. Then all observations in P are protected conditionally on instructions in I_c according to Definition 4.21.*

The theorem above relates to observation preservation. More generally, to support the evaluation of a logical property such as an ACSL formula [13], one may need to observe an arbitrary partial state with a variable number of (*name*, *value*) pairs. The monolithic, decoupled and observation-opacification patterns above achieve this, by collecting all name-value domains occurring in the logical property and atomically observing their name- and address-value pairs.

6 PUTTING IT TO WORK

Let us now describe the implementation of our approach across multiple levels of program representation through an optimizing compilation flow, ranging from source code, compiler IR, down to binary code. We focus on C programs, representative of secure embedded applications, and we build our framework on the LLVM compilation infrastructure. It is composed of three main phases. The first one is the front-end which translates high level programming languages into the LLVM IR. As our source programs are written in C, we naturally choose Clang, the LLVM front-end for C, C++ and Objective-C. Then, in a second phase, the LLVM IR generated by Clang is processed by the source-and-target-independent middle-end optimizers. In the last phase, the back-end (or the code generator) lowers the LLVM IR produced by the optimizers to a machine-specific representation called LLVM MIR, which supports both SSA and register-allocated non-SSA forms; the LLVM MIR is subject to a few late machine code optimizations, before finally being converted to assembly code.

6.1 Value Preservation in Source Code

We introduce language extensions to Clang to support observation (`snapshot`) and opacification (`opaque`) expressions. We define three builtins `__builtin_opacify`, `__builtin_observe_mem` and `__builtin_io` corresponding, respectively, to the `observe_opacify`, `observe_pair` and `observe_tailio` macros defined in the previous section.

- `__builtin_opacify` is a variadic function implementing value opacification. It returns the same scalar value as its first argument, but made opaque to the compiler. This opaque value

may then replace the original one in subsequent code. All other arguments are optional and represent additional data dependence relations to implement opaque chains constraining program transformations. The builtin function also observes (snapshots) all its arguments: this provides a means to validate the opacification mechanism by tracing observed values down to the generated machine code.
- `__builtin_observe_mem` implements an address-value pair observation. It reads a value from its pointer argument and observes (snapshots) this value together with the pointer itself. It returns a token to implement downstream opaque chains.
- `__builtin_io` is a variadic function implementing an I/O effect: the arguments serve to extend opaque chains linking upstream observations to the I/O effect. The function returns a token to implement downstream opaque chains.

These language extensions enable the programmer to define additional constraints when transforming the program, in the form of data dependences or ordering relations. As unit type is not natively defined in C, we currently use integer-typed variables to represent tokens produced by `__builtin_observe_mem` or `__builtin_io` and only used by our builtins. There are two main cases: when builtins are removed and this eliminates all uses of a token variable, this variable is eliminated as well and does not incur any resource overhead; when the token variable remains live due to escaping values (in function call or return, or in memory), this variable, will incur low resource usage in the generated machine code, most likely a single stack slot for the whole function and no register usage beyond the token definition on a RISC ISA. A better solution would be to implement a fully expressive token type in LLVM (the current one is limited—it may not be used in ϕ nodes—and has a different, provenance-tracking purpose).

6.2 Value Preservation in LLVM

Let us now describe the transformation of our language extensions to two different compiler intermediate representations: the IR on which the optimizers operate and the MIR which represents the final code to be emitted by the compiler.

6.2.1 Value Preservation in LLVM IR. LLVM IR supports intrinsic (a.k.a. builtin) functions with compiler-specific semantics. Intrinsics require the compiler to follow additional rules while transforming the program. These rules are communicated to the compiler via the *function attributes* which specify the intrinsic function's behavior w.r.t. the program mutable state (memory, control registers, etc.). Intrinsics provide an extension mechanism without having to change all of the transformations in the optimizers.

To implement our preservation mechanism, we introduce three intrinsics to the LLVM IR:

- `llvm.opacify` has the same semantics as `__builtin_opacify`. It is pure, does not access memory and has no I/O or other side-effects: it is valid to optimize away `llvm.opacify` if the opaque value is not used in subsequent code.
- `llvm.observe.mem` has the same semantics as `__builtin_observe_mem`. Unlike `llvm.opacify`, `llvm.observe.mem`'s attributes let it read argument-pointed memory. Other than such reads it has no side effects: it is valid to optimize away `llvm.observe.mem` if the output token is not used in subsequent code. Being able to access argument-pointed memory is actually an optimizing implementation of the `observe_pair` macro presented in Section 5: this avoids having to generate instructions loading from these memory locations.
- `llvm.io` has the same semantics as `__builtin_io`.

We modified Clang to map `__builtin_opacify`, `__builtin_observe_mem` and `__builtin_io` to `llvm.opacify`, `llvm.observe.mem` and `llvm.io` respectively, when generating LLVM IR from C code; observations are represented in LLVM IR as metadata attached to the corresponding intrinsic.

LLVM IR metadata is indeed designed to convey additional information to optimizers and code generators [28], and we defined a new type of metadata carrying information about observed values: (1) the observation builtin source-level identifier such as source line and column number, (2) the program point at which the observation takes place, and (3) the location holding the observed value at this point. We modify a few utility functions commonly used by different optimization passes such as `replaceAllUsesWith()` and `combineMetadata()` to update (e.g. when combining two intrinsics) and maintain (e.g. when duplicating the intrinsic) the metadata attached to the intrinsic throughout the optimization pipeline. An alternative would have been to embed metadata directly into the intrinsic (by passing a metadata operand to the intrinsic), avoiding to modify these utility functions to update and maintain observation metadata. But this would prevent optimizations from combining intrinsics when values are equal but metadata differs (e.g. line and column numbers).

6.2.2 Value Preservation in LLVM MIR. To preserve values throughout code generation we also need to implement our mechanism in the MIR. We achieve this by lowering the intrinsics `llvm.opacify`, `llvm.observe.mem` and `llvm.io` respectively into `OPACIFY`, `OBSERVE_MEM` and `IO` pseudo-instructions, with the same semantics and behaviors w.r.t. memory accesses and side effects. Pseudo-instructions are MIR instructions that do not have machine encoding information and must be expanded, at the latest, before code emission. Nevertheless, our mechanism should not interfere with the emitted machine code; the pseudo-instructions introduced are thus not expanded but completely removed during code emission. To guarantee the correct functional behavior of the program when removing the pseudo-instructions, the `OPACIFY` pseudo-instructions uses the same register as its first operand to hold the opaque value.

The preservation of observation metadata is more challenging: LLVM does not currently support attaching metadata to MIR instructions, we thus have to transform IR metadata into an operand of `OPACIFY` and `OBSERVE_MEM` pseudo-instructions. This may require modifications to passes in the code generator to maintain and update the metadata while not preventing them from optimizing the program: indeed, as discussed in the alternative implementation above, this approach is potentially preventing some optimizations from combining pseudo-instructions with the same arguments but different observation metadata; fortunately we did not have to do so since we did not find any such missed optimizations on our benchmark suite and on the different backends considered.

At the final stage of the code generator, observation metadata is also emitted into machine code in the debug section. This allows to communicate information about the observed values to binary code utilities carrying out the validation of observation and opacification mechanisms (the debugger, binary code verifiers, etc.). To represent this information in machine code, we extend the DWARF format [19], which provides an easily extensible description of the executable program. Yet unlike the more conventional approach relying on debug information generated by the compiler itself [52], we maintain and update the information of observed values ourselves, only using DWARF for its standard encoding of the data, since it is already supported by most binary code utilities.

7 PRESERVING SECURITY PROTECTIONS

It has been shown that there is a correctness-security gap in compilation, which arises when compiler optimizations preserves the functional semantics but violates a security guarantee made by source program [23]. As a consequence, security engineers have been fighting with optimizing compilers for years by devising and introducing complex programming tricks to the source code, though yet found a reliable way to obtain secure binary code [48]. In this section, we demonstrate, on different examples, how our mechanisms can be used to preserve security protections through an optimizing compilation downto the generated binary.

7.1 Sensitive Memory Data Erasure

First, let us consider our motivating example described in Section 3. The security protection consists in erasing a secret buffer allocated on the stack after usage; 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".

To preserve the erasure, we insert an opaque artificial read of values stored in the buffer, after the call to `memset()`. We then use the value produced by the opacification in an I/O effecting operation to guarantee that it does not get removed, as shown in Listing 2. We validate the approach on mbedTLS's RSA encryption and decryption [41], called `erasure-rsa-enc` and `erasure-rsa-dec` in the following. A short opaque chain links `__builtin_observe_mem()` to the final I/O builtin. The former is also an observation that enables the validation of the security property (i.e., effective erasure).

```
1 void process_sensitive(void) {
2   uint8_t secret[32];
3   ...
4   memset(secret, 0, sizeof(secret));
5   __builtin_io(__builtin_observe_mem(secret));
6 }
```

Listing 2. Erasing a buffer with observation.

7.2 Computation Order in Masking Operation

Something as simple as respecting the computation order, as explicitly written in the source code, of associative operations, may be difficult to achieve when compiling with optimizations enabled. Indeed, as long as the generated program produces matching observable behaviors w.r.t. the C standard, compilers have perfectly the right to reorder associative operations, even with proper parenthesizing, and doing so independently of the optimization level.

Now, it may sound like no big deal to reorder associative operations, because after all, this is what associativity really means. Nonetheless, this can be problematic when it comes to using associative operations such as `xor` for masking against side-channel attacks. In fact, masking operation is a commonly used countermeasure to protect block cipher implementations against side-channel attacks [32]. Consider the code excerpt of a masking scheme shown in Listing 3. The secret `k` is first masked with `m` (line 1), then is remasked with `mpt` (line 3). Note how the programmer has intentionally put the parentheses to express the fact that `k` has to be remasked with `mpt` before removing the old mask `m`. Nevertheless, it has been reported that the statement in line 3 has been compiled as `k^(mpt^m)`, which altogether defeats the countermeasure [26].

```
1 k ^= m;
2 ...
3 k = (k ^ mpt) ^ m;
```

Listing 3. Masking example.

In order to preserve the correct order in the masking operation, we propose a solution based on opacification, as shown in Listing 4. We first linearize the compound expression to three-address form by explicitly declaring a temporary variable `tmp` to hold the result of the remasking operation (line 3), which is next used for unmasking (line 5). Obviously, this alone would only guarantee the correct masking operation if no optimizations enabled. To prevent compiler optimizations from removing `tmp` and reodering the masking operation, we opacify the result of the remasking

operation, making it unknown to the compiler (line 4). Furthermore, we assign the opaque value to `tmp` to make sure that subsequent code refers to this value instead of the original one, thus guaranteeing the use of the correct value in the removal of the old mask `m`. This forms an opaque chain from the definition of `tmp` to the definition of `k`. There is no need for a terminal I/O builtin since we already know that the computation of `k` is transformation-preserved, `k` being the value of interest in downstream computation. Notice also that the cardinality constraint on values in opaque chains is trivially satisfied by the bijectivity of the exclusive or operator. The opaque chain enforces the ordering constraint that the opacified value of `tmp` will be observed after the first masking operation and before the second one.

We validate our approach on a masked implementation of Advanced Encryption Standard (AES) [30], named `mask-aes` in the following. In the following, we will also consider a self-written application called `mask-rotate`, which contains a loop of masking operations with the same security property as `mask-aes`, together with I/O instructions; the goal is to evaluate the performance overhead of our lightweight implementation relying on pure intrinsics without side effects.

```
1 k ^= m;
2 ...
3 uint8_t tmp = k ^ mpt;
4 tmp = __builtin_opacify(tmp);
5 k = tmp ^ m;
```

Listing 4. Secure masking using opacification.

7.3 Step Counter Incrementation

Fault attacks are a growing threat for secure devices such as smart cards. Such attacks can alter the system's correct behavior via physical injection means [55]. For example, it has been shown that fault attacks can induce unexpected jumps to any location in the program [15, 39]. One source-level scheme to enhance the resilience against such fault attacks [34] is shown in the code excerpt from Listing 5. The protection consists in defining a step counter at each control construct (line 2), and stepping the counter of the immediately enclosing control construct after every C statement of the original source (lines 4 and 6). Counters are then checked against their expected values at the exit of the enclosing control construct (lines 8 and 9), calling a handler when it fails (line 10). We refer to this technique as Step Counter Incrementation (SCI), which may be seen as a very fine-grained form of Control Flow Integrity (CFI) [2, 17].

```
1 ...
2 unsigned short cnt_if = 0;
3 if (cond) {
4   cnt_if++;
5   a = b + c + 1;
6   cnt_if++;
7 }
8 if (!((cnt_if == 2 && cond) ||
9       (cnt_if == 0 && !cond)))
10  fault_handler();
```

Listing 5. SCI protection.

However, as fault attacks are not modeled in compilers, optimizations are free to transform the program even when it does not preserve the security countermeasures inserted by the programmer. Indeed, counter checks are removed—their conditions are trivially true in a "fault-free" semantics

of the program. Counter incrementations might hence 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 make use of our opacification mechanism to preserve the SCI protection, as shown in Listing 6. In fact, preserving the SCI protection boils down to (1) protecting counter incrementations and checks and (2) guaranteeing the proper interleaving of functional and countermeasure statements. The former can be achieved by opacifying counters at each of their incrementations (lines 4 and 6), and at checks against expected constant values (lines 8 and 9), so that checks can no longer be deduced, while counter values can no longer be constant-propagated and must be incremented instead. As for the latter, we need to create additional data dependences between values defined by the functional code and counter values: we insert an artificial use of the counter value at each functional value definition and inversely, an artificial use of the last functional value defined at each counter incrementation. To achieve this, we opacify non-constant operands used in definitions of functional or counter values and express these artificial uses as token parameters of the opacification operator (lines 5 and 6). This creates an opaque chain linking every counter incrementation to the next counter use, and then again to the next incrementation until the terminating fault handler, while interleaving original program statements in the chain through the bundling of both counter and original variables in opacification builtins. Notice that the opaque chain includes a control dependence when linking with the fault handler. We validated this approach on two well-known smart-card benchmarks: PIN authentication [24] and AES encryption [36], called `sci-pin` and `sci-aes` in the following.

```
1 ...
2 unsigned short cnt_if = 0;
3 if (cond) {
4   cnt_if = __builtin_opacify(cnt_if) + 1;
5   a = __builtin_opacify(b, cnt_if) + __builtin_opacify(c, cnt_if) + 1;
6   cnt_if = __builtin_opacify(cnt_if, a) + 1;
7 }
8 if (!((__builtin_opacify(cnt_if) == 2 && cond) ||
9       (__builtin_opacify(cnt_if) == 0 && !cond)))
10  fault_handler();
```

Listing 6. Secure SCI protection using opacification and data dependences.

7.4 Source-Level Loop Protection

Recent research has shown that loops are particularly sensitive to fault attacks. Indeed, faulted iteration counter in AES cipher could lead to retrieval of secret key [21], while fault injections in the core loop of memory copy operation during embedded system's boot stages may allow an attacker to control the target's execution flow which eventually will lead to arbitrary code execution on the target [51]. Other work highlighted the need to protect the iteration counter of the PIN code verification on smart cards [24].

To enforce the correct iteration counter of sensitive loops, a compile-time loop hardening scheme has been recently proposed and implemented in LLVM [43]. It operates on the LLVM IR and is based on the duplication of loop termination conditions and of the computations involved in the evaluation of such conditions. However, such redundant operations do not impact the program observable semantics and are ideal candidates to be optimized away by downstream optimizations [31]. The authors has originally investigated and analyzed different compilation passes in order to select a relevant position for the loop hardening pass in the compilation flow, so that the countermeasure is preserved in the executable binary. We argue that the investigation of the

```
1 int memcmp(char *a1, char *a2, unsigned n) {
2   for (unsigned i = 0; i < n; ++i) {
3     if (a1[i] != a2[i]) {
4       return -1;
5     }
6   }
7   return 0;
8 }
```

Listing 7. Original `memcmp()` implementation.

positioning of compile-time countermeasure pass can be facilitated, if not dismissed. We implement the loop hardening scheme at source level and leverage our opacification mechanism to preserve the redundancy-based protection through the whole optimizing compilation flow. Consider an implementation of `memcmp()` function, shown in Listing 7.

```
1 int memcmp(char *a1, char *a2, unsigned n) {
2   unsigned i, i_dup, n_dup = n;
3   for (i = 0, i_dup = 0; i < n; ++i, ++i_dup) {
4     if (i_dup >= n)
5       fault_handler();
6     if (a1[i] != a2[i]) {
7       if (a1[i_dup] == a2[i_dup])
8         fault_handler();
9       if (n_dup != n)
10        fault_handler();
11      return -1;
12    }
13  }
14  if (i_dup < n)
15    fault_handler();
16  if (n_dup != n)
17    fault_handler();
18  return 0;
19 }
```

Listing 8. Securing `memcmp()` loop.

Listing 8 demonstrates our approach on the core loop of the above `memcmp()` function. We duplicate the loop counter `i` (line 3) and loop-independent variables being used in the loop body (`n` in this case, line 2). Furthermore, we insert redundant computations of the exit condition at every iteration of the loop (line 4), as well as at the loop exit (lines 7 and 14). We also verify that the values of the duplicated loop-independent variables at every loop exit are correct w.r.t. the values of their original counterparts (lines 9 and 16).

To prevent optimizations from removing the redundant data and code, we opacify every assignment to the duplicated variable (lines 2 and 4 from Listing 9): the compiler can no longer detect the identity relation between the original and its corresponding duplicated variable. Like in the previous example, the resulting opaque chains interleave original computations with checks, and link to a terminating fault handler through a control dependence. We validate the source-level loop hardening scheme on the core loop of PIN authentication [24], named `loop-pin` in the following.

7.5 Constant-Time Selection

Another well-known, yet hard to achieve example of security property is selecting between two values, based on a secret selection bit, in constant time. This means the generated code for the

```
int memcmp(char *a1, char *a2, unsigned n) {
  unsigned i, i_dup, n_dup = __builtin_opacify(n);
  for (i = 0, i_dup = 0; i < n; ++i, ++i_dup) {
    i_dup = __builtin_opacify(i_dup);
    if (i_dup >= n)
      fault_handler();
    if (a1[i] != a2[i]) {
      if (a1[i_dup] == a2[i_dup])
        fault_handler();
      if (n_dup != n)
        fault_handler();
      return -1;
    }
  }
  if (i_dup < n)
    fault_handler();
  if (n_dup != n)
    fault_handler();
  return 0;
}
```

Listing 9. Secure `memcmp()` loop using opacification.

selection operation must not contain any jump conditioned by the secret selection bit, otherwise the execution time of the operation will depend on whether the first or the second value is selected, thus leaking the secret selection bit. Cryptography libraries resort to data-flow encoding of control flow, bitwise arithmetic at source level to avoid conditional branches, but this fragile constant-time encoding may be altered by an optimizing compiler.

```
/// a. Constant-time selection between two values, version 1
uint32_t ct_select_vals_1(uint32_t x, uint32_t y, bool b) {
  signed m = 0 - b;
  return (x & m) | (y & ~m);
}
```

```
/// b. Constant-time selection between two values, version 2
uint32_t ct_select_vals_2(uint32_t x, uint32_t y, bool b) {
  signed m = 1 - b;
  return (x * b) | (y * m);
}
```

```
/// c. Constant-time selection from lookup table
uint64_t ct_select_lookup(const uint64_t tab[8], const size_t idx) {
  uint64_t res = 0;
  for (size_t i = 0; i < 8; ++i) {
    const bool cond = (i == idx);
    const uint64_t m = (-(int64_t)cond);
    res |= tab[i] & m;
  }
  return res;
}
```

Listing 10. Constant-time selection attempts.

Consider different functions from Listing 10. Other than the first two attempts at implementing constant-time selection between two values `x` and `y` based on a secret selection bit `b`, we also consider an example where the programmer wishes to select a value from the lookup table `tab`

while hiding the secret lookup index `idx`. All these functions are carefully designed to contain no branch conditioned by the secret value: a bitmask `m` is created from the secret value using arithmetic tricks, then is in turn used to select the wanted values. Nevertheless, it has been reported that the code generated by LLVM is not guaranteed to be constant-time. For instance, the compiler introduces a conditional jump based on the secret value when compiling, with optimizations enabled, the first two functions for IA-32 [48], or the last function for both IA-32 and x86-64 [38]. The community is desperately in search of a reliable way to prevent the compiler from spotting and optimizing the constant-time idioms. The most certain approach currently available is perhaps to introduce to the compiler a specially-crafted builtin that will be ultimately compiled into a conditional move instruction (if available in the target architecture) [48]. However, this is rather a proof-of-concept and lack of generalizability: it only supports the operation of selecting between two values, and needs to be rewritten in order to implement the constant-time selection from lookup table from Listing 10.c for instance.

We propose in Listing 11 an alternative to the above solution, relying on our opacification mechanism. The intuition is to hide from the compiler the correlation between the bitmask `m` and the secret selection bit `b`. This prevents the compiler from recognizing the selection idioms and turning it into conditional jumps: it embeds bitwise logic into an opaque chain linking select arguments to the return value. Moreover, we do not assume calls to these constant-time selection functions to be part of opaque chains; instead we create an opaque chain inside each function and make sure that it terminates by an opaque expression by opacifying the return value. This is a case of conditional transformation-preservation of instructions (Definition 4.12): individual selection operations may or may not execute depending on (non-sensitive) program input, but as soon as one of these executes, the constant-time expressions it encloses will be transformation-preserved due to the opaque chain forcing the compiler to compute the bitmask (and its complement) then using it for the selection. Furthermore, for Listing 11.c, not only we want to ensure that the compiler does not transform the selection inside the loop into a branch conditioned by the secret index, but additionally we want to preserve the constant-timeness of the whole loop by making sure that the `|` operation takes place at every iteration. This is implemented by opacifying each element of the array. It is worth noting that, unlike the traditional approach trying to reliably generate conditional move instruction whenever available, we accurately generate the expected constant-time code from the programmer's data-flow encoding of control flow. Although this may result in slower code, this can be directly applied to other constant-time operations involving value preservation. We validate this solution on mbedTLS's RSA decryption [41] and a self-written RSA exponentiation using Montgomery ladder [48], respectively called `ct-rsa` and `ct-montgomery` in the following.

Interestingly, this does not work as intended for Listing 11.b: since `b` is of type `bool`, the compiler knows that `x * b` can only yield `0` or `x`, thus generate a conditional jump by enumerating all possible values of `b`. This can easily be fixed by slightly modifying the multiplication as illustrated in Listing 12; however, this exposes the limit of our mechanism: we cannot rely on data opacification and dependences to prevent optimization passes from introducing control-flow to the program. To the best of our knowledge, there exists no real solution to this problem (yet): it has always been valid for compilers to modify the program's control-flow as long as this does not alter the program's behavior, and this is something we usually have no control over.

8 METHODOLOGY AND VALIDATION

In this section, we first validate the functional correctness of our observation-preserving approach and implementation, then describe our validation methodology and use it to establish the preservation of the security protections on the applications presented in Section 7.

```
/// a. Constant-time selection between two values, version 1
uint32_t ct_select_vals_1(uint32_t x, uint32_t y, bool b) {
  signed m = __builtin_opacify(0 - b);
  return __builtin_opacify((x & m) | (y & ~m));
}
```

```
/// b. Constant-time selection between two values, version 2
uint32_t ct_select_vals_2(uint32_t x, uint32_t y, bool b) {
  signed m = __builtin_opacify(1 - b);
  return __builtin_opacify((x * b) | (y * m));
}
```

```
/// c. Constant-time selection from lookup table
uint64_t ct_select_lookup(const uint64_t tab[8], const size_t idx) {
  uint64_t res = 0;
  for (size_t i = 0; i < 8; ++i) {
    const bool cond = (i == idx);
    const uint64_t m = __builtin_opacify(-(int64_t)cond);
    res |= __builtin_opacify(tab[i]) & m;
  }
  return __builtin_opacify(res);
}
```

Listing 11. Secure constant-time selections using opacification.

```
uint32_t ct_select_vals_2(uint32_t x, uint32_t y, bool b) {
  signed m = __builtin_opacify(1 - b);
  return __builtin_opacify((x * (1 - m)) | (y * m));
}
```

Listing 12. Secure constant-time selection version 2.

8.1 Functional Validation by Checking Value Integrity and Ordering

Establishing the preservation of an observation event amounts to proving the existence of an observation point at which all observed values are available (cf. Definition 4.20), at the proper memory address or associated with the appropriate variable, and following the specified partial order. This consists in checking, for a given program execution, (1) the presence of all observation events, and (2) that for each of these events, the observed values of the specified variables and memory locations are the expected ones, and (3) that event ordering is compatible with the specified partial order. To this end, we leverage the concept of *observation trace*, which is the sequence of program partial states defined by all observation events encountered during a given execution of the program [52]. Practically, validation involves comparing, for a given input of the program, two observation traces:

(1) Reference trace: we execute the reference program compiled with optimizations disabled. The reference program is the original program (without our intrinsics) with `printf` inserted to generate the expected observed values. We assume `-O0` preserves the observation events as well as the partial state of the ISO C abstract machine [33] containing the observed values of each event.
(2) Optimized trace: we execute the program with builtins inserted, compiled with our compilation framework at different optimization levels. We modify a DWARF parser library [27] to create a list of breakpoints containing all observation addresses in the binary code, as reported in the DWARF section. At each of these addresses, we record the locations where

the observed values are stored. We finally retrieve these values during program execution, using a debugger.

To compare the traces, we associate each partial state defined by an observation event (`printf` in the reference program or intrinsic in our version) with a unique identifier—a combination of line and column numbers at which the event is defined in the program source. We then verify using an offline validator (a small Python program) that (1) each partial state in the reference trace has a corresponding counterpart (having the same identifier) in the optimized trace, and inversely (2) each partial state in the optimized trace has a correspondence in the reference trace. The validator also verifies that all values in a given partial state from the optimized trace match the expected values reported in its reference counterpart. Now, this leaves us with the question of validating observation ordering: unlike Vu et al. we only enforce a partial ordering on program observation events. There is no particular constraint for the relative order of observation events having no data dependence relation. As a consequence, due to code motion or rescheduling during compiler optimizations, the reference and optimized traces might not be identical. As a result, we propose an incomplete validation methodology aiming for practicality and still providing high confidence. The methodology is twofold: for every benchmark we derive (i) a totally ordered version where a unique temporary variable is used as a written-to and read-from token to chain all observation events (in other words, all events consume and write to the same token), and (ii) a minimally ordered version where a distinct token is produced from every observation event and immediately consumed in a distinct I/O instruction (the latter being decoupled from the former, it does not constrain the ordering of observation events).

Moreover, since we model observation events as side-effect free, pure functions, different observations of the same values may be combined into a single one. As such, during program execution, a single point observing a given value might actually correspond to multiple observations of the same value. Although the transformation is perfectly valid, it leads to false positives reported by the validator when comparing observation traces. To eliminate these erroneous validation results, we update instruction combining support functions in LLVM to embed the metadata representing individual observations to be combined into a single combined observation (referencing both variable names, line numbers, etc.). These embedded observations will eventually be expanded when creating observation breakpoints for the debugger, which allows the corresponding partial states to be logged into the observation trace.

We validate the functional correctness of our implementation on a subset of the test suite of Frama-C, a static analysis framework for C source program [20]. The test suite is designed to validate different Frama-C analyses on a range of C programs representative of the language semantics, using program properties written in the ACSL annotation language [13]. We restrict ourselves to properties verifying the expected values of variables at a given program point, ignoring test cases referring to more advanced ACSL constructs. These properties can easily be expressed as observation events with our intrinsics.

We compile each of these test cases at 6 optimization levels `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`. This results in two sets of 31 applicable test cases featuring 616 observations—one set for totally ordered events (i) and one for minimally ordered events (ii). Notice that these test cases are not meant to be evaluated as performance benchmarks, we only use them to validate the correctness of our implementation.

We automatically verified that in both sets, all 616 properties have been correctly propagated to machine code. In the first set (i) we checked that the observation trace is identical to the reference trace. In the second set (ii) we checked that values are all present and correct, but could not

verify any partial ordering constraint (as there is none to be checked). All of this, at all considered optimization levels.

8.2 Security Protection Preservation Validation

Validating the preservation of security protections is more challenging. While verifying value integrity is enough to prove the preservation of observation events, it is only a necessary condition for preserving security protections. Hence, other than applying value integrity verification to the security applications from Section 7, we also define additional mechanisms to validate specific components of the preserved protections.

Checking Value Utilization. In our security examples, opacification is used to (1) protect key values of the security countermeasure which are subject to program optimizations—such as duplicated variables from the loop hardening scheme or step counters from SCI protection—from being optimized away, and (2) make sure that the protected values are actually used in the subsequent code of the countermeasure (different security checks for instance). Clearly, value integrity verification only guarantees the former, we need a second verification to assess the latter. On the one hand, we first determine the uses of opacified values in the program, then verify that they are indeed part of the original countermeasure in the source program. On the other hand, we first find the critical parts of each of the security protections, such as the removal of the old mask from the masking operation, or various redundant checks. We then determine the operands of these key computations and verify that they indeed are results of opacifications.

The mechanism is undoubtedly protection-dedicated, as important uses of the opacified values really depend on the considered countermeasure scheme, and thus requires manual inspection of the generated code. Nonetheless, the two-phase verification can be implemented as an automated data-flow analysis at the program's MIR or machine code, as long as the analysis tool knows about uses of the opacified values crucial to the considered countermeasure.

Checking Statement Ordering. For SCI, other than preserving the step counter incrementations and the security checks, we also need to guarantee the proper interleaving of functional and countermeasure statements. This requires opacifying operands of every statement with an artificial dependence of the result from previous statement, thus creating an opaque chain. This ensures that, given two consecutive C statements S_1 and S_2, all MIR/assembly instructions between the opacification of the first operand of S_1 and the opacification of the first operand of S_2 correspond to S_1. We manually inspect the generated code to verify this ordering.

Notice that there would be an option to automate this verification of the proper interleaving of functional and countermeasure statements, *if* one trusts the debug information to be sound and accurate. We could verify the line numbers, mapping MIR/assembly instructions to the corresponding source statements and vice versa. If (1) all MIR/assembly instructions between the opacification of the first operand of a statement S_1 and the opacification of the first operand of the next statement S_2 have the same line number, and (2) during the scan over every MIR/assembly instructions of the program, the line number reported for each instruction (from the same basic block) is in an ascending order, it can be concluded that functional and countermeasure statements are correctly interlaced. Unfortunately debug information is not robust enough in general and we prefered to rely on manual inspection for higher confidence.

Checking Constant-Time Selection. As explained in Subsection 7.5, a widely-adopted informal definition of constant-time selection is that there is no conditional branch based on a secret selection bit. To validate the preservation of constant-timeness using our opacification approach, we manually verify that none of the following three values is used to compute branch conditions:

a secret selection bit, a bitmask created from it, or its opacified value. Notice that a conservative verification scheme could be implemented as a static data-flow analysis on the generated machine code, even though the automated determination of whether the branch conditions depend on the secret selection bit might be challenging, notably when the data flow involves memory accesses.

Validation Results. Table 1 summarizes different validation schemes that we apply for each application presented in Section 7, in order to verify the preservation of different security countermeasures. For each application we could verify that the appropriate schemes yield the expected results, validating our approach and implementation.

	`erasure-*`	`mask-*`	`loop-pin`	`sci-*`	`ct-*`
Value Integrity	✓✓	✓✓	✓✓	✓✓	✓✓
Value Utilization	N/A	✓✓	✓✓	✓✓	✓✓
Statement Ordering	N/A	N/A	N/A	✓✓	N/A
Constant-time Selection	N/A	N/A	N/A	N/A	✓✓

Table 1. Validation of different security applications. ✓indicates the scheme is *applied* to the program, N/A indicates the opposite. ✓ indicates the scheme is *validated* for the program.

9 EXPERIMENTAL EVALUATION

Let us now evaluate the proposed mechanisms on the security applications presented in Section 8, focusing on performance and compilation time impact.

9.1 Experimental Setup

For each one of the considered security applications, we first compare our versions against the unoptimized programs—which is also a solution to preserving security protections—to quantify performance benefits. We then compare our versions against other available preservation mechanisms, namely compiler-dependent programming tricks for constant-time selection [48], and Vu et al.'s I/O-based approach for all other applications [52]. For fairness purposes we use the same version of LLVM as Vu et al. Eventually, we also compare our compiler-based implementation with an alternative one that does not involve modifications to Clang and LLVM but relying on inline assembly instead; we will describe it in the following. Finally, we present the compilation time overhead of our implementation.

For all benchmarks, we target two different instruction sets: ARMv7-M/Thumb-2 which is representative of deeply embedded devices, and Intel x86-64 representative of high-end processors with a complex micro-architecture. In addition, since Simon et al. showed that the compilation of source-level constant-time selection code on the IA-32 architecture contained secret-dependent conditional jumps, we also consider IA-32 for the constant-time applications `ct-rsa` and `ct-montgomery`.

Performance evaluation for the ARMv7-M/Thumb-2 ISA takes place on an MPS2+ board with a 32-bit Cortex-M3 clocked at 25 MHz with 8 Mb of SRAM, while our Intel test bench has a quad-core 2.5 Ghz Intel Core i5-7200U CPU with 16 GB of RAM.

We use the Intel platform for compiling for either target. Changing the target only concerns the back-end, a short part of the compilation pipeline, as a result, we only report the compilation time evaluation results for the ARMv7-M/Thumb-2 ISA.

Our experiments cover all common optimization levels (`-O1`, `-O2`, `-O3`, `-Os`, `-Oz`). Performance measurements are based on the average of 10 runs of each benchmark and configuration.

9.2 Comparing to Unoptimized Programs

Figure 3 presents the speed-up of our approach at different optimization levels over unoptimized programs. For all benchmarks, speedup ranges from 1.2 to 12.6, with an harmonic mean of 2.8. Clearly, our observation- and opacity-based approach to preserving security protections enables aggressive optimizations with significant benefits over `-O0`.

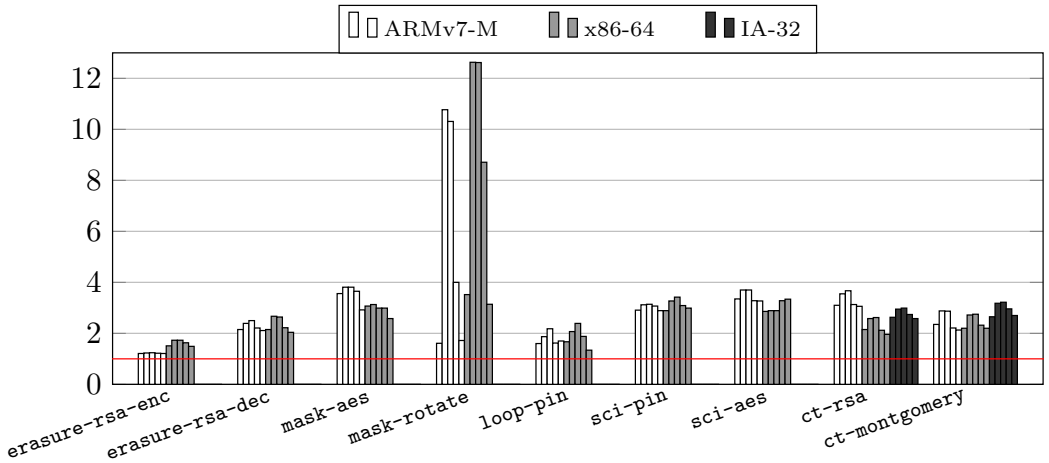

Fig. 3. Speed-up of our approach over unoptimized original programs—ordered by compiler option -O1, -O2, -O3, -Os, -Oz. The red line represents a performance ratio of 1.

9.3 Comparing to Reference Preservation Mechanisms

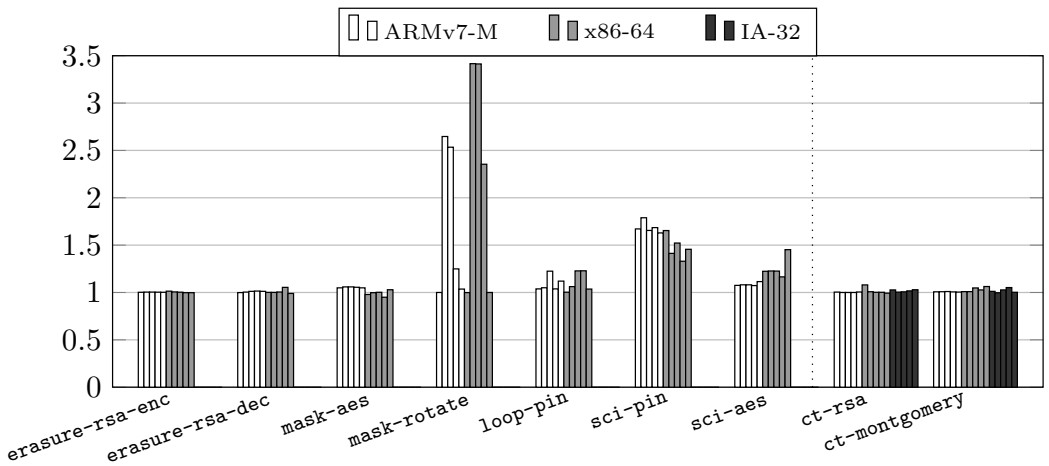

Fig. 4. Speed-up of our approach over reference preservation approaches—compilation-time property-preserving annotations [52] for applications on the left side of the dotted line and programming tricks [48] for the ones on its right side—ordered by compiler option -O1, -O2, -O3, -Os, -Oz.

Figure 4 presents the speedup our approach compared to reference preservation approaches, at different optimization levels. For `erasure-rsa-enc`, `erasure-rsa-dec`, `mask-aes`, `mask-rotate`,

`loop-pin`, `sci-pin` and `sci-aes`, we compare our approach against the property preservation mechanism proposed by Vu et al. [52]. The authors introduced new intrinsics to implement their property preservation mechanism, however, they rely heavily on the I/O side effects of the intrinsics: not only they introduce I/O side-effecting intrinsics to model observation points so that these cannot be removed by optimizations, they also insert I/O side-effecting artificial definitions for every property-observed value to protect these from being optimized out. Furthermore, in order to guarantee the correct debug information for these values, the authors inserted more artificial definitions to prevent multiple live ranges corresponding to the same source variable from overlapping. Furthermore, to ensure the correct values in memory at observation points, these intrinsics also behave like memory fences, i.e. can read from and write to memory. As a consequence, our implementation with pure intrinsics (no side effects), accessing memory only when required, should enable more optimizations and thus result in faster code. Our results clearly confirm this. For example, although `rotate-mask` contains the masking computation, the data used in the operation is passed as function arguments instead of being declared as global variables in reference implementations; this clearly allows more optimizations when the function is inlined (i.e. when compiled at `-O2`, `-O3` or `-Os`), and especially when the function call is inside a loop. More generally, optimizations such as "loop unrolling" and "loop invariant code motion" are the main sources of benefits with our approach at these optimization levels. On the contrary, for `erasure-rsa-enc` and `erasure-rsa-dec`, the function implementing the protection only contains the erasure of the sensitive buffer, we thus observe almost no difference compared to the property preservation mechanism proposed by Vu et al. Similarly for `aes-herbst`, the data required for the masking computation is stored in memory as global variables, there is almost no difference between two versions: the masking operation contains loads and stores to the secret key as well as the different masks in the order defined in the source program, as the protection is correctly preserved. As for other applications, for both targets, we note a clear improvement, ranging from 1.04 to 1.79, with an average of 1.3. Overall, compared to our approach, I/O side-effecting intrinsics restrict compiler optimizations, thus inevitably degrade the performance of generated code.

For `ct-rsa`, we compare our approach against the constant-time selection implementation of mbedTLS [41], which is basically the same as the version from Listing 10.a, but with the computation of the bitmask (line 3) splitted into a separate function. Furthermore, this function must not be inlined in order to prevent the compiler from optimizing it away. As for `ct-montgomery`, we compare our approach against the specially-crafted implementation of OpenSSL [50]. It is worth noting that general-purpose compilers offer no guarantees of preserving constant-timeness: future versions of the same compiler may spot the trick and optimize the constant-timeness away [48]. Although our approach allows the functions implementing constant-time selection to be inlined while still preserving constant-timeness, these only take a small fraction of the execution time; we do not notice a clear difference compared to other constant-time implementations.

9.4 Comparing to Alternative Implementations

Production compilers provide an *inline assembly* syntax to embed target-specific assembly code in a function. The feature is regularly used by C programmers for low-level optimizations and operating system primitives, and also for sensitive applications to avoid interference from the compiler [44]. GCC-compatible compilers implement an extension of the optional ISO C standard syntax for inline assembly, allowing programmers to specify inputs or outputs for inline assembly as well as its behavior w.r.t. memory accesses and I/O effects [49]. This specification stands as a contract between the assembly code and the compiler. Compilers, relying on the contract, are completely agnostic to what happens inside the an inline assembly region. In other words, the assembly code region is opaque to the compiler. We may thus leverage this feature to implement opaque expressions. For

example, to preserve the correct masking order in Listing 4, the call to `__builtin_opacify` at line 4 may be replaced by an inline assembly expression, as shown in Listing 13.

```
1 k ^= m;
2 ...
3 uint8_t tmp = k ^ mpt;
4 __asm__ ("" : "+r" (tmp));
5 k = tmp ^ m;
```

Listing 13. Secure masking using opacification based on inline assembly.

The inline assembly region (line 4) is actually empty: the behavior exposed to the compiler of the whole expression is specified by the `"+r" (tmp)` constraint. This essentially means that `tmp` is both the input and output of the expression, and that the expression neither accesses memory nor does it have any side-effect. As an output of the inline assembly expression, `tmp` is now opaque to the compiler, just as if it was defined by the `__builtin_opacify`.

This example can be generalized to implement any opaque region. In practice, it is sufficient to implement a small set of builtins covering the typical opacification scenarios. A set of preprocessor macros can be designed to cover these typical scenarios and provided as a portable interface across most compilers and targets.

Note that this approach slightly complicates the implementation of observations, carrying precise variable names, memory addresses, line numbers down to machine code. Additional conventions and post-pass on the generated assembly code are required to produce the appropriate DWARF representation, as described in Section 6.

Now, the natural question is to compare the performance of an inline-assembly-based implementation with our compiler-native opaque regions. To this end, we consider a subset of the applications presented in Section 7, containing `erasure-rsa-enc`, `erasure-rsa-dec`, `mask-aes`, `mask-rotate`, `loop-pin`, `ct-rsa` and `ct-montgomery`. We exclude `sci-pin` and `sci-aes`, as these applications would require the manual insertion of inline assembly expressions at every statement of C source programs, which is impractical. Figure 5 presents the speedup our compiler-native implementation w.r.t. inline-assembly, at different optimization levels.

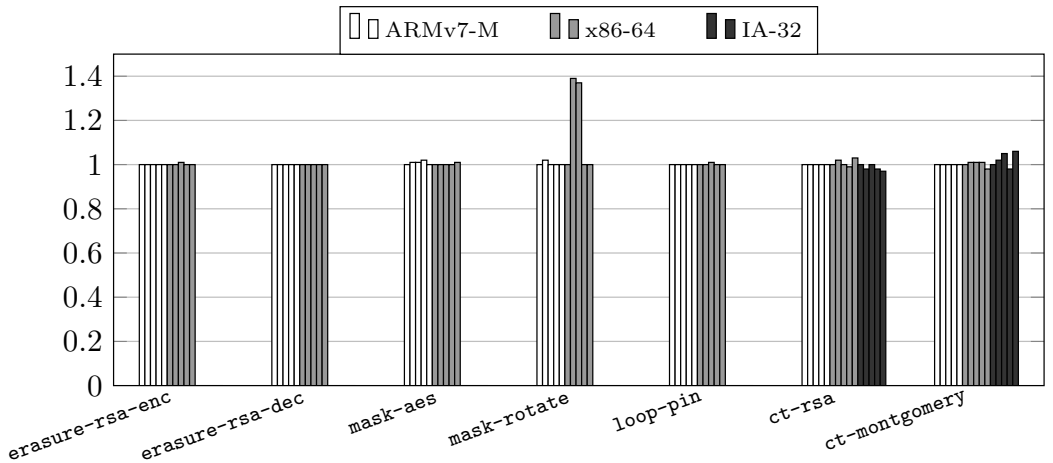

Fig. 5. Speed-up of our approach over inline assembly—ordered by compiler option -O1, -O2, -O3, -Os, -Oz.

In the majority of cases, the two considered implementations generate the same executable code. For `ct-rsa` and `ct-montgomery`, there is a slight difference in performance due to discrepancies in register allocation. This is not visible in other applications because these two programs are larger and demonstrate higher register pressure. The only significant performance difference is for `mask-rotate` compiled with `-O2` and `-O3` for x86-64: our compiler-native implementation is 40% faster than inline assembly. The core loop of the inline-assembly-based version happens not to be unrolled, while compiler-native version's is. Interestingly, this is only the case for x86-64: the same loop is unrolled for both versions when compiling for ARMv7-M. Indeed, the difference disappears when we force loop unrolling using the `-funroll-loops` option together with `#pragma unroll`. As expected, inline assembly occasionally interferes with compiler optimizations, despite the precise specification enabled in its syntax, while compiler intrinsics allow for carrying more precise semantics to the optimizers. Mitigations exist, and make the inline assembly approach interesting to some multi-compiler development environments. The take away from this is that both approaches are sound and leverage the same formalization and secure development scenarios (for opacification purposes, not for observation purposes). Yet this may not always be the case in the future: compilers are not forbidden to analyze inline assembly and take optimization decisions violating the opacity hypothesis; the fact they do not do it today is no guarantee that secure code will remain secure in future versions. On the contrary, our intrinsics in the source language and IR have a explicit and future-proof opacification and observation semantics.

9.5 Compilation Time Overhead

Figure 6 shows the compilation time overhead compared to compiling the original programs at the same optimization level. Note that the optimized original programs are insecure, as protections have been stripped out or altered by optimizations. We consider the Intel platform since it is used for both native and cross-compilation for both targets.

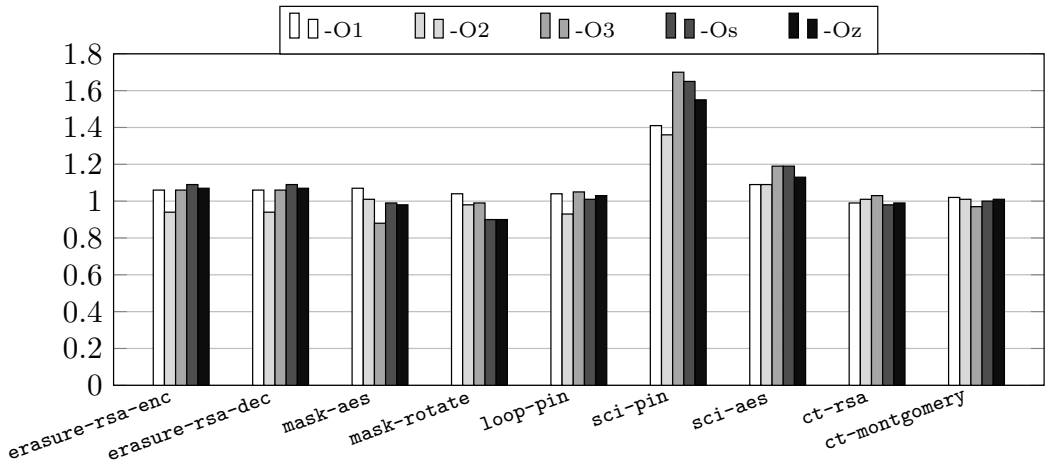

Fig. 6. Compilation time overhead on the Intel platform, compared to compiling the original programs at the same optimization level.

In general, the compilation time overhead is under 10%, except for `sci-aes` and `sci-pin` where it ranges from 13% up to 70%. As discussed in Subsection 7.3, the SCI protection represents a very important part of the whole application (as counter incrementations are inserted after each C instruction), and it is completely stripped out from original programs when optimizations are

enabled without opacification. As a consequence, the code size of the insecure baseline is much smaller than the secure code with fully-preserved countermeasures, which justifies the important compilation time difference.

10 CONCLUSION

We formally defined the notion of observation and its preservation through program transformations. We instantiated this definition and preservation mechanisms through multiple program representations, from C source code down to machine code. The approach relies on two fundamental principles of compiler correctness: (1) the preservation of I/O effects and (2) the interaction of data dependences with program constructs that are opaque to static analyses. We formally proved the correctness of the approach on a simplified intermediate language, and validated it within the LLVM framework with virtually no change to existing compilation passes. Our proposal specifically addresses a fundamental open issue in security engineering: preserving security countermeasures through optimizing compilation flow. Avenues for further research include software engineering scenarios such as testing of production code and more robust debugging of optimized code.

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

[7] Andrew W. Appel. 1998. SSA is Functional Programming. *ACM SIGPLAN Notices* 33, 4 (1998), 17–20. https://doi.org/10.1145/278283.278285

[8] 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

[9] 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.

[10] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. 2004. The Sorcerer's Apprentice Guide to Fault Attacks. *IACR Cryptology ePrint Archive* 2004 (2004), 100. http://dblp.uni-trier.de/db/journals/iacr/iacr2004.html#Bar-ElCNTW04

[11] 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

[12] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proc. ACM Program. Lang.* 4, POPL (2020), 7:1–7:30. https://doi.org/10.1145/3371075

[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] Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. 2013. Sleuth: Automated Verification of Software Power Analysis Countermeasures. In *Cryptographic Hardware and Embedded Systems - CHES 2013*, Guido Bertoni and Jean-Sébastien Coron (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 293–310.

[15] 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

[16] 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

[17] 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

[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] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria. 2013. Electromagnetic glitch on the AES round counter. In *Fourth International Workshop on Constructive Side-Channel Analysis and Secure Design - COSADE'2013 (Lecture Notes in Computer Science, Vol. Lecture Notes in Computer Science)*, Springer Berlin Heidelberg (Ed.). Springer Berlin Heidelberg, Paris, France, pp 17–31. https://doi.org/10.1007/978-3-642-40026-1

[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] V. D'Silva, M. Payer, and D. Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Security and Privacy Workshops*. 73–87.

[24] 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

[25] 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

[26] 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

[27] Eli Bendersky. 2011. pyelftools - Python library for parsing ELF files and DWARF debugging information. https://github.com/eliben/pyelftools

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

[29] 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

[30] 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

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

[32] 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.

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

[34] 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, Vol. 8713)*, Miroslaw Kutylowski and Jaideep Vaidya (Eds.). Springer International Publishing, Wroclaw, Poland, 200–218. https://doi.org/10.1007/978-3-319-11212-1_12

[35] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:2002.11054 [cs.PL]

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

[37] 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

[38] llvm dev. 2020. *Side-channel resistant values.* http://lists.llvm.org/pipermail/llvm-dev/2019-September/135079.html

[39] 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

[40] 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

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

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

[43] 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

[44] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. 2018. An Analysis of X86-64 Inline Assembly in C Programs. *SIGPLAN Not.* 53, 3 (March 2018), 84–99. https://doi.org/10.1145/3296975.3186418

[45] Matthieu Rivain and Emmanuel Prouff. 2010. Provably Secure Higher-Order Masking of AES. In *Cryptographic Hardware and Embedded Systems, CHES 2010*, Stefan Mangard and François-Xavier Standaert (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 413–427.

[46] 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

[47] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

[48] 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

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

[50] The OpenSSL Project. 2003. OpenSSL: The Open Source toolkit for SSL/TLS. (April 2003). www.openssl.org.

[51] 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

[52] 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* (San Diego, CA, USA) *(CC 2020)*. Association for Computing Machinery, New York, NY, USA, 14–26. https://doi.org/10.1145/3377555.3377897

[53] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* 7 (02 2017). https://doi.org/10.1007/s13389-017-0152-y

[54] 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 (01 Jun 2018), 111–130. https://doi.org/10.1007/s41635-018-0038-1

[55] 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