Reconciling optimization with secure compilation

Son Tuan Vu, Albert Cohen, Arnaud de Grandmaison, Christophe Guillon, Karine Heydemann

Reconciling Optimization with Secure Compilation

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

Software protections against side-channel and physical attacks are essential to the development of secure applications. Such protections are meaningful at machine code or micro-architectural level, but they typically do not carry observable semantics at source level. This renders them susceptible to miscompilation, and security engineers embed input/output side-effects to prevent optimizing compilers from altering them. Yet these side-effects are error-prone and compiler-dependent. The current practice involves analyzing the generated machine code to make sure security or privacy properties are still enforced. These side-effects may also be too expensive in fine-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 input/output semantics-preservation contract of compilers. We introduce an opacification mechanism to preserve and enforce a partial ordering of observations. This approach is compatible with a production compiler and does not incur any modification to its optimization passes. We validate the effectiveness and performance of 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: compilation, security, optimization, debugging, LLVM

ACM Reference Format:
Son Tuan Vu, Albert Cohen, Arnaud De Grandmaison, Christophe Guillon, and Karine Heydemann. 2021. Reconciling Optimization with Secure Compilation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 142 (October 2021), 30 pages. https://doi.org/10.1145/3485519

1 INTRODUCTION

Programmers expect compilers to preserve the Input/Output (I/O) behavior of a program. "I/O behavior" itself is loosely defined as any value or side-effect that is "externally visible". Compilers achieve this by preserving the semantics of input or output operations, and functionally linking input to output values. In particular, optimizing compilers make sure that computations resulting in I/O take place in the order and under the conditions specified in the source program, or that they are replaced by observationally equivalent alternatives.

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; Arnaud De Grandmaison, Arm, Paris, France, arnaud.degrandmaison@arm.com; Christophe Guillon, STMicroelectronics, Grenoble, France, christophe.guillon@st.com; Karine Heydemann, Sorbonne Université, CNRS, LIP6, 4 place Jussieu, 75252, Paris, France, karine.heydemann@lip6.fr.

2475-1421/2021/10-ART142
https://doi.org/10.1145/3485519

Interestingly, solely preserving I/O behavior is not enough. In many scenarios, computations that do not result in I/O also need to be preserved, but optimizations are too aggressive at removing, reordering or otherwise modifying such computations. We focus on two such notable scenarios:

(1) **Securing applications.** Optimizing compilers are known to interfere with a wide range of security protections. 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 [D'Silva et al. 2015; Percival 2014; Simon et al. 2018]. In cryptography applications, countermeasures against side-channel attacks are even more fragile: optimizations may invalidate masking protections—randomizing sensitive data [Rivain and Prouff 2010]—by reordering elementary operations in masking expressions [Bayrak et al. 2013]. Related to this and to forestall timing attacks, encryption/decryption kernels are usually designed to run for a constant amount of time, independent of secret data [Yarom et al. 2017]. To achieve this, security engineers go to great lengths to write straight line code, carefully avoiding control flow depending on sensitive data [Simon et al. 2018]; unfortunately, compilers 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 [Simon et al. 2018]. As for fault attacks, which can alter the system's correct behavior by means of physical interference [Yuce et al. 2018] and are a growing threat for embedded systems, the situation is even worse. Software countermeasures against such attacks usually involve some form of redundancy, such as replicating data and operations on the data then comparing the results [Bar-El et al. 2004; Barry et al. 2016; Hillebold 2014; Proy et al. 2017]. Preserving these redundant computations fuels a permanent fight with compilers striving to remove redundant code [Hillebold 2014; Simon et al. 2018]. Another countermeasure detecting fault attacks altering the program's control flow consists in incrementing and checking counters alongside with the execution of individual instructions, source code statements, function calls, etc. [Lalande et al. 2014]. Once again, compilers will remove such trivially true counter checks or coalesce multiple incrementations into a single addition.

(2) **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 item. There has been a large body of work showing the importance of analysis and verification tools to query program properties—expressed in propositional logic—at the level of machine code [Balakrishnan and Reps 2010; Bréjon et al. 2019; Shoshitaishvili et al. 2016]. 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 [Balakrishnan and Reps 2010], or from the essential role played by low-level (micro-)architectural features [Bréjon et al. 2019]. More generally, machine code analysis tools and runtime monitors (including debuggers) often need program properties associated with a high-level specification, from which the static or dynamic analysis may determine whether the application is secure against a given attack [Bréjon et al. 2019]. Still, it is generally challenging to propagate source-level properties all the way down to machine code [Vu et al. 2020]. 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 fail to preserve this link or to 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 properties may also be affected by compiler optimizations, e.g. such variables may be optimized out, thus invalidating the property.

The common pattern in the above scenarios is that program transformations along the compilation flow—such as abstraction lowering steps and optimizations—have no means to reason about high-level properties that the programmer would like to enforce and to convey down to the machine code. Indeed observational equivalence is conventionally defined w.r.t. I/O effects, and unless high-level properties are captured together with I/O effects, the compiler does not have to preserve them. As a result, security engineers resort to embedding I/O effects into security protections and assert-like statements for testing.

1.1 Motivating Example

Let us illustrate the challenge on Listing 1, a (simplified) string comparison function with a source-level protection against fault attacks induced by physical injection means such as laser beams or electromagnetic pulses [Yuce et al. 2018]. The protection aims to detect register corruption or instruction skip inside the loop, avoiding early loop termination [Proy et al. 2017]. This is implemented by checking a duplicate counter `i2` and duplicate bound `n2`. Note that the copies and checks take place before and after the loop, respectively. This is safe according to a fault model that only considers attacks on the loop itself [Proy et al. 2017]. We will use this function as a running example. An optimizing compiler will easily eliminate all computations and checks involving these duplicate variables. Inserting I/O effects dependent on the values of `i`, `i2`, `n` and `n2` will help preventing their elimination. But it is not sufficient to preserve the redundant incrementation and checks as the compiler still has access to equality properties such as `i = i2` (in the absence of faults). As a result, security engineers also resort to "opacifying" the duplicate values, using coding and compilation tricks [Paul Bakker, ARM 2019; The OpenSSL Project 2003]; yet these tricks remain fragile [Percival 2014; Simon et al. 2018].

```
1  unsigned compare(char* a, char* b,
2                   unsigned n) {
3    unsigned res, i, i2, n2 = n;
4
5
6    for (i = 0, i2 = 0; i < n;
7         ++i, ++i2)
8      res |= a[i] ^ b[i];
9    if (i2 < n2 || n2 != n)
10     fault_handler();
11   else return res;
12 }
```

Listing 1. Secure comparison

```
1  unsigned compare(char* a, char* b,
2                   unsigned n) {
3    unsigned res, i, i2, n2, a0, ai2, an;
4    barrier(n, &an); n2 = an;
5    barrier(0, &a0); i2 = a0;
6    for (i = 0; i < n;
7         ++i, barrier(i2, &ai2), i2 = ai2 + 1)
8      res |= a[i] ^ b[i];
9    if (i2 < n2 || n2 != n)
10     fault_handler();
11   else return res;
12 }
```

Listing 2. Secure comparison with side-effecting barriers

Vu et al. [2020] recently demonstrated how to reduce the correct compilation of security protections to the *observation* of values and memory locations. These observations are preserved automatically across program transformations through the insertion of *barriers* isolating upstream values from downstream ones using I/O effects. In addition, the inserted I/O effects induce a total ordering of all observations, which is a conservative solution but also incurs significant performance overhead (note that preserving one observation requires more than one I/O effect).

Back to our string comparison example, Vu et al. would implement the observation of the successive incrementations of the duplicate `i2` by means of I/O side-effecting barriers. We represent these with the `barrier` call in Listing 2. These barriers instruct the compiler to preserve the

evaluation of their first argument, while opacifying and duplicating its value into the second argument.[1] In this example, `a0` and `an` hold an opaque duplicate of the constants `0` and `n`, respectively, avoiding forward substitution of these constants into arithmetic expressions or comparison on `i2`. The incrementation of `i2` itself is protected with the same mechanism using `ai2`. This approach is effective, but forces the sequential execution of all barrier calls and introduces many I/O effects restricting the effectiveness of compiler optimization. For example, even when the loop is unrolled, the loads from `a` and `b` and the definition of `res` would not be vectorized over multiple iterations.

1.2 Introducing Opaque Observations

This paper refines the notion of *observation* and its preservation. *We propose a means to specify and preserve observations down to machine code. We do so in a very general and portable manner, avoiding to modify each and every compiler pass, extending automatically to future transformations, and with as little performance impact as possible.* We control how information about program values is made visible to the compiler's static analyses by means of an *opacification* mechanism, i.e., the ability to hide information about an atomic expression of operational semantics. Opacity means the compiler sees the result of such opacification operations as statically unknown yet functionally deterministic values. We tie observations and opacification together as a single operation called an *opaque observation*. In general, program transformations may delete, duplicate or create an opaque observation, but the compiler is restricted in what analyses it can perform on them. We will formalize this notion in the following, but one may informally reflect on what analyses a compiler may perform:

- gathering the uses of an opaque observation;
- determining whether it loads from references or memory, and from which memory addresses;
- deciding whether it has I/O effects;
- deciding whether two opaque observations are identical up to variable renaming;

and on what analyzes a compiler may *not* perform:

- establishing a correlation between the uses of an opaque observation and the values it defines;
- or a correlation between observed references/addresses and the data they hold in memory.

In this model, I/O is a special case of opaque observation that compilers must preserve.

Eventually, we let security engineers build dependence chains of opaque observations called opaque chains, serving two purposes: they provide a portable and efficient means to protect the evaluation of security-sensitive expressions by chaining them to downstream I/O effects; they also provide a portable and efficient means to enforce a partial ordering on the evaluation of security-sensitive values. Opaque chains take the form of an alternating sequence of opaque observations and regular instructions, implementing a dependence chain. An opaque chain starts with an opaque observation and ends with an opaque observation. There are subtle "information-carrying guarantees" restricting what instructions may occur on an opaque chain: the compiler should not have enough information on opaque values or on the possible control-flow paths dependent on an opaque value to break an opaque chain into distinct dependence chains. We will detail this when formalizing the concept.

1.3 Back to the Example

Let us informally illustrate these concepts on the string comparison example again. A security engineer may observe the initialization and successive incrementations of the duplicate `i2` of `i`.

[1]Note: Vu et al. [2020] do not directly write to the duplicate through a pointer, but insert an *artificial* SSA definition instead (hence the prefix `a` in the second argument); we use this syntax for improved readability.

They may insert three observe expressions as shown on Listing 3,[2] instructing the compiler to preserve the evaluation of specific values. The observe returns the value of its first argument, made opaque to compiler analyses. It also takes an arbitrary number of additional arguments. This allows to bundle the observation of one or more variables with the construction of use-def chains serving as dependence constraints. For example, `i2 = observe(0, n2)` defines i2 to an opacified version of the constant 0, while also forcing the initialization of i2 to happen after the one of n2 (this forces the duplication of n to take place before the loop, which is important to detect a fault corrupting n while the loop executes). The programmer also expresses the need to evaluate individual incrementations of i2 in sequence, rather than substituting i2 with a closed form expression. This conjunction of observation, opacification and (partial) ordering is the tool we offer to security engineers to implement a semantics-preservation contract with the compiler.

```
1 unsigned compare(char* a, char* b, unsigned n) {
2   unsigned res, i, i2, n2 = observe(n);
3   for (i = 0, i2 = observe(0, n2); i < n; ++i, i2 = observe(i2) + 1)
4     res |= a[i] ^ b[i];
5   if (i2 < n2 || n2 != n) fault_handler();
6   else return res;
7 }
```

Listing 3. Secure comparison with observations

Finally, while our simple example only considered observations of scalar values, many scenarios involve preserving the evaluation and the storage of a specific value at a specific memory location. The volatile keyword in C serves that purpose, but opaque observations extended to operate on references and memory addresses provide a more efficient mechanism. Consider the following sequence: `*p = 42; v = *p; observe(v)`. The compiler may implement a store-load forwarding transformation, replacing the sequence with `observe(42)`. But this is not the case of `*p = 42; observe(p)`: the compiler can determine that the opaque observation reads memory at address p, but it is not allowed to determine what data is stored at that address.

1.4 Outline

The paper is structured as follows. It starts with a discussion of the most closely related work (Section 2). Then we illustrate our approach on programming patterns that include security hardening and software protections typical of security-sensitive applications (Section 3); these patterns range from straight-line scalar code to interprocedural control flow and capture common security properties that optimizing compilers frequently end up corrupting. We express the secure compilation of these programming patterns in terms of opaque observations to be made at specific points of the computation. Considering clang/LLVM with aggressive optimizations turned on, we also show that preserving observations does not require significant modifications to existing compilers (Section 4). Based on this motivation and experience, we provide a formal semantics for observations (Section 5), and for the proposed opacification and chaining mechanisms (Section 6). This allows us to prove the main contribution of this paper: the protection of opaque observations through opaque chains for arbitrary valid transformations. We validate that the implementation preserves opaque observations and the security properties expressed on top of them (Section 7). Finally, we evaluate the performance and compilation time of our approach, and discuss alternative implementations (Section 8).

[2]More observations could be inserted according to the security properties to be enforced and validated—e.g. the values of conditional expressions.

2 RELATED WORK

There is a large body of research and engineering on secure compilation [Abadi 1998; Abadi and Plotkin 2012; Abate et al. 2018; Devriese et al. 2016; Gorla and Nestmann 2016]. The correctness of a compiler is defined w.r.t. a notion of behavioral equivalence. It may take different forms, from full abstraction to more specific type and isolation properties [Abadi 1998; Abadi and Plotkin 2012; Chlipala 2007; Patrignani et al. 2015] or even hyperproperties not directly captured in terms of behavioral equivalence [Abate et al. 2019]. Behavioral equivalence is generally defined w.r.t. the capabilities of an attacker. We are interested in enforcing properties of the machine state resulting from the compilation of the program, including logical properties of side-channels and countermeasures to physical attacks. Like Barthe et al. [2020] we extend the semantics of a host language 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). Instead, we provide a means to express observations at deterministic points of the execution and to preserve these while subjecting program to aggressive transformations; Vu et al. [2020] previously showed such observations enable the expression and validation of a wide range of logical properties [Baudin et al. 2008].

To prevent interference from compiler optimizations, security engineers resort to embedding I/O or volatile side-effects into security protections [Vu et al. 2020]. But as our experiments will show, I/O 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. As discussed in the introduction, Vu et al. [2020] proposed a means to automate the embedding of I/O effects into a general-purpose compiler; yet they 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.

In order to bound the worst-case execution time of a reactive method, compilers for hard real-time systems are often capable of carrying loop trip count information down to the emitted assembly code. These so called *flow facts* also capture infeasible paths and program points that are mutually exclusive during a given run [aiT 2003; Ballabriga et al. 2010]. CompCert provides such a mechanism, but does not formalize 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 [Schommer et al. 2018].

The ENTRA (Whole-Systems ENergy TRAnsparency) project Deliverable D2.1 [Eder et al. 2016] 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 preserve security protections due to the lack of opacification and chaining of observations.

Another safety-minded approach encodes flow facts using IR extensions and external transformations to update loop trip counts [Li et al. 2014]. It incurs significant changes to optimization passes, as it requires a set of transformation-specific rules to update flow facts along the compilation flow.

As introduced earlier, our main motivations and applications are related to application security. The reader may refer to Vu et al. [2020] for an extensive discussion of security-related applications.

3 CHALLENGES WITH SECURE OPTIMIZING COMPILATION

Let us now review typical software protections, some of the associated pitfalls with their correct optimizing compilation, as well as the remedies enabled by opaque observations.

3.1 From Security Properties to Opaque Observations

At this point, it is natural to ask about the general form of software protections, and about the process to derive opaque observations from higher level security properties. Vu et al. [2020] informally tackled this issue, translating the so-called non-functional behavior of the machine (physical effects such as side-channels and faults) in terms of functional properties of source-level variables and memory, when considering a program equipped with security countermeasures. We realize that the issue is actually twofold. First, one needs to determine what to observe (i.e. which values have to be preserved by the compiler), then how to observe it (in this paper, we propose to create an opaque chain to prevent the elimination of observations and force a partial ordering). Solving the first problem requires a deep understanding of the attack models and the associated software/hardware protections. For example, duplicate variables and computations involving these variables have to be preserved in order for a redundancy-based protection to be effective (as shown in the motivating example). Once the observed values are identified, our main theorem (Theorem 6.3) provides security engineers with a clear path: observations will be preserved if they are implemented through opaque chains leading to a downstream instruction that is guaranteed to be preserved by compiler optimizations. Unfortunately, the process of creating such opaque chains may involve an abstraction of data and control flow beyond the reach of static analyses. And especially so if one ought to limit the impact of ordering constraints, to take full advantage of compiler optimizations. Also, we do not believe it is currently realistic to propose a logic of the physical machine that captures a wide range of side-channels and fault models. This makes it difficult to imagine an automated, multi-leak, multi-fault translation of security properties into opaque observations. While this may be a long-term goal shared among physical attack experts, it is not achievable in the short term. As a pragmatic step forward, we assume that a security expert is willing to identify sufficient conditions for the prevention of side channels and faults according to a given leakage and fault model, and we propose a contract with the compiler to express such sufficient conditions in terms of security protections/countermeasures leveraging opaque observations and ordering constraints. By design, any program transformation preserving I/O events and ordering will implement this contract.

3.2 Security Patterns

The selection of a suite of standard secure applications happens to be an immediate concern. While numerous benchmark suites provide a common ground to compare metrics such as code size or run-time performance (SPEC, EEMBC, etc.), there does not exist well-established benchmarks dedicated to security; instead, researchers study representative coding patterns associated with typical vulnerabilities and attack scenarios [Dureuil et al. 2016; Witteman 2018]. Also, while code size or performance are also important for secure applications, the correctness of the implementation w.r.t. an actual security property comes first (privacy, confidentiality, authentication, etc.). We thus selected a range of security patterns from widely-used libraries (openSSL, mbedTLS, etc.) and from the literature. Let us now present these patterns to illustrate our approach (we will again refer to those in the correctness validation and performance evaluation sections (Section 7 and Section 8)).

3.2.1 Control Flow Integrity. Our first security pattern illustrates fault attacks, a growing threat for secure devices such as smart cards. Such attacks can alter the system's correct behavior by means of physical injection [Yuce et al. 2018]. For example, it has been shown that fault attacks can induce

jumps to arbitrary locations in the program [Berthomé et al. 2012; Moro et al. 2013]. One natural solution to enhance the resilience against such fault attacks consists in embedding counters into secure code, stepping them after every program statement of the source code and regularly checking against their expected (usually constant) values [Lalande et al. 2014]. This technique is used in the FISSC benchmark suite dedicated to fault injection analysis [Dureuil et al. 2016]. We refer to it as Step Counter Incrementation (SCI) in the following; it may be seen as a very fine-grained form of Control Flow Integrity (CFI) [Abadi et al. 2005; Burow et al. 2017]. However optimizations are free to transform the program even when it does not preserve the security protections. And this is what we observe in practice: counter checks are removed, being trivially true in a "fault-free" semantics of the program. This is one typical scenario where practitioners have to disable compiler optimization when implementing security protections.

Preserving the SCI protection boils down to (1) protecting counter stepping and counter checks, while (2) guaranteeing the proper interleaving of functional and counter-checking statements. The former can be achieved by opaquely observing every counter-stepping expression, so that counter checks can no longer be simplified at compilation time. As for the latter, we introduce additional dependences between values defined by the functional code and counter values. For example, consider the original code in Listing 4 and its transformation into Listing 5.

```
1 unsigned short cnt = 0;
2 a = b + c;
3 cnt++;
```

Listing 4. Fragile step counters

```
1 unsigned short cnt = 0;
2 a = observe(b, cnt) + observe(c, cnt);
3 cnt = observe(cnt, a) + 1;
```

Listing 5. Opaque step counter observations

We opaquely observe non-constant operands of both functional code (e.g. `b` and `c` in the definition of `a` from line 2) and counter incrementations (line 3). We express artificial data dependences through additional arguments of `observe`. This creates an opaque chain linking every counter incrementation to the next use of a counter, and then again to the next incrementation until the terminating counter check. We also interleave original program statements in the chain through the bundling of both counter and original variables in opaque observations. In Section 7 and Section 8, we will validate this approach on two classical smart-card benchmarks protected with SCI: PIN authentication [Dureuil et al. 2016] and AES encryption [Levin 2007], named `sci-pin` and `sci-aes`.

3.2.2 Fault Detection Through Redundancy. As we have seen in the motivating example (Listing 1), detecting fault attacks often uses some form of redundant computation (time, information, or space redundancy) [Hillebold 2014; Moro et al. 2013; Proy et al. 2017]. One of the most common and effective countermeasures involves double-computing every sensitive value. If the results of redundant computations do not match, a fault is detected and reported by calling a handler. Such redundant operations do not impact the program observable semantics and are ideal candidates to be optimized away by optimizations [Hillebold 2014; Proy et al. 2017]. To prevent optimizations from removing this specific form of redundancies, we opaquely observe every duplicate definition or assignment so that the compiler can no longer identify the copy with the original. By doing so, we also create an opaque chain including a control dependence linking to the fault handler. In Section 7, we will validate this scheme on the core loop of PIN authentication [Dureuil et al. 2016], hardened with a source-level loop protection scheme based on redundant computations [Proy et al. 2017],[3] named `loop-pin`.

[3] A comprehensive protection scheme for loop control flow in the spirit of the motivating example.

3.2.3 Erasure of Sensitive Data. The leakage of confidential information such as secret keys is a major threat and has been extensively studied in secure compilation. A common countermeasure consists in erasing sensitive data from memory once they are no longer needed [Percival 2014]; such data include secret keys, seeds of random generators, or temporary encryption or decryption buffers. However, this may not be as easy as it seems. For example, given a stack-allocated buffer containing sensitive information, it should be erased before returning from the function, typically via a call to `memset`. However, compilers will spot that the buffer goes out of scope, and will consider the `memset` as dead store, removing erasure as part of "dead store elimination". Security engineers resort to different programming tricks [Yang et al. 2017], including platform-supplied variants of `memset` which are guaranteed to be preserved by compilers.[4] These workarounds have multiple drawbacks: on the one hand, they are not portable due to the poor adoption of some of the `memset` variants; on the other hand, even when portability is not an issue, resetting the secret memory with a constant value is a bad idea in terms of side channel leakage, so one should rather store random values to effectively erase the memory. In order to solve the root of the dead store (either with `memset` or with random values) elimination problem, we insert an opaque observation of values stored in the buffer, after the erasure of the latter and before the function return, so that the compiler cannot consider the erasure as "dead store" and remove it anymore. To guarantee that the observation itself does not get removed, we may use the value it defines in a subsequent I/O instruction, either immediately after the call or linking it to a longer opaque chain. This simple scheme is illustrated in Listing 6. In this example, `observe` has been lifted to arrays, and (conventionally) returns the address of the array; `io` stands for a dummy I/O side-effect enforcing the evaluation of `p`, hence the evaluation of `observe(secret)` itself. Note that the observation also enables validating the effectiveness of the erasure in a debugger. In Section 7 and Section 8, we will validate the approach on mbedTLS's RSA encryption and decryption [Paul Bakker, ARM 2019], named `era-rsa-enc` and `era-rsa-dec`.

```
1 void process_sensitive(void) {
2  uint8_t secret[32];
3  ...
4  memset(secret, 0, sizeof(secret));
5  uint8_t *p = observe(secret);
6  io(p);
7 }
```

Listing 6. Erasing a buffer with observation.

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

Listing 7. Secure masking using opacification.

3.2.4 Computation Order in Masking Operation. Respecting the source's evaluation order of associative operations may be difficult to achieve with an optimizing compiler. Indeed, as long as the generated program produces matching observable behaviors w.r.t. the C standard, compilers have the right to reorder associative operations, even with proper parenthesizing, and doing so independently of the optimization level. This can be problematic when it comes to using associative operations such as exclusive-or for masking purposes, a common countermeasure to protect block ciphers against side-channel attacks [Ishai et al. 2003]. Consider the masking scheme `k=(k^mpt)^m`. Assuming the secret `k` is already masked with `m`, it is now remasked with `mpt`, so that the old mask `m` can safely be removed. It has been reported that this statement has been compiled as `k^(mpt^m)`, which altogether defeats the countermeasure [Eldib and Wang 2014]. To preserve the correct masking order, we opaquely observe the result of the re-masking operation, making it

[4] In fact, the C library recently introduced a `memset_s` call that specifically provides such a guarantee, yet its implementation remains platform-specific and such a case-by-case approach is not scalable.

unknown to the compiler. The opaque value is used in the removal of the old mask `m`. In order to form an opaque chain, we also opaquely observe the result of the unmasking operation, making the definition of `k` the tail opaque instruction of the chain. The resulting implementation is shown in Listing 7. There is no need for a terminal I/O instruction since we already know that `k` is the value of interest in downstream computation, and the computation of `k` will thus be preserved by transformations. The opaque chain enforces the ordering constraint that the observed value will be computed after the first `^` operation and before the second one. In Section 7 and Section 8, we will validate the approach on a masked implementation of AES encryption [Herbst et al. 2006], named `mask-aes`. We also derive a synthetic benchmark from `mask-aes`, to facilitate performance comparisons across property preservation approaches: `mask-rotate` is a simple loop over masking operations with the same security property as `mask-aes`.

3.2.5 Constant-Time Selection. Another well-known, yet hard to achieve example of security property is selecting between two values, based on a secret boolean value, in constant time. This means the generated code for the selection operation must not contain any branch conditioned by the secret selection boolean value, otherwise the execution time of the operation would depend on whether the first or the second value is selected, thus leaking the secret. Cryptography libraries resort to data-flow encoding of control flow, bitwise arithmetic at source level to avoid conditional branches, but this encoding may be altered by an optimizing compiler.

```
1 uint32_t ct_select_vals(
2     uint32_t x, uint32_t y, bool b) {
3   signed m = 0 - b;
4   return (x & m) | (y & ~m);
5 }
```

Listing 8. Fragile constant-time selection

```
1 uint32_t ct_select_vals(
2     uint32_t x, uint32_t y, bool b) {
3   signed m = observe(0 - b);
4   return (x & m) | (y & ~m);
5 }
```

Listing 9. Robust version using opaque observations

Listing 8 implements constant-time selection between `x` and `y` based on a secret boolean value `b`. Special care is taken to avoid any branch conditioned by `b`: instead, one derives a selection bitmask `m` from the secret using boolean arithmetic, then use it to select the appropriate value (line 3). Nevertheless, it has been reported that the code generated by LLVM is not guaranteed to be constant-time. For instance, on IA-32, the compiler recognizes the selection idiom and transforms it into branch conditional on the secret value [Simon et al. 2018]. The currently available solution is to extend the compiler with a specially-crafted builtin that will be ultimately compiled into a conditional move instruction (if available on the target architecture) [Simon et al. 2018]. However, it is rather specific to constant-time selection between two values and is not sufficient to implement, for example, the constant-time memory lookup operation [Sprenkels 2020]. We propose a more general alternative, relying on opaque observations, shown in Listing 9. Hiding the correlation between the bitmask `m` and the secret boolean value `b` prevents the compiler from recognizing the selection idiom and turning it into a conditional branch. We thus embed the bitwise logic into an opaque chain linking the constant-time selection to the function return value. This is a case of conditional transformation preservation: individual selection operations may or may not execute depending on (non-sensitive) program input, but as soon as one of these execute, its enclosed constant-time expressions will be transformation-preserved thanks to the opaque chain forcing the compiler to evaluate the bitmask. Our formalization will provide a definition and preservation theorem for such conditional preservation cases. In Section 7 and Section 8, we will validate this solution on mbedTLS's RSA decryption [Paul Bakker, ARM 2019] and a version of RSA exponentiation using the Montgomery ladder [Simon et al. 2018], respectively named `ct-rsa` and `ct-mgmr`.

4 PUTTING IT TO WORK

Let us now investigate the practicality of opaque observation concepts by describing how these can be implemented in a real-world compiler. This is also historically how we came up with the formalization, building and refining it from experience.

Our framework is built upon the LLVM infrastructure. Compilation traverses multiple levels of program representation: the Clang front-end lowering C to LLVM IR, the middle-end optimizers on LLVM IR in SSA form, and the back-end lowering the IR to a machine-specific register-level LLVM MIR.

4.1 C Language Extensions

We extend Clang with opaque observation syntax, introducing token values and 3 variadic builtins:

- A *token* is a value of an abstract type, opaque to the compiler, only used for building opaque chains. It is implemented as an integer of unknown value in LLVM, and meant to be eliminated in the back-end when its role in building opaque chains is over.
- `__blt_obs_var` implements `observe` on C variables and constants. It returns the same scalar value as its first argument made opaque to the compiler; this opaque value may replace the original one in subsequent code. Arguments may be tokens. The builtin also observes its arguments, allowing to validate the integrity of the partial observation state at every stage of the compilation flow. Additional (optional) arguments link with upstream opaque chains.
- `__blt_obs_mem` implements `observe` on the region of memory pointed to by its first argument and returns a token. Additional (optional) arguments link with upstream opaque chains and may be tokens.
- `__blt_io` implements an unordered I/O effect. The function returns a token. All (optional) arguments link with upstream opaque chains.

The tokens produced by the last two builtins serve to initiate downstream opaque chains. Remember tokens are opaque to the compiler: LLVM is not able to equate token values resulting from distinct calls, and distinct calls to these builtins initiate distinct downstream opaque chains.

4.2 LLVM Extensions

We may now describe the lowering 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.

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 compiler passes. We thus introduce three IR intrinsics `llvm.obs.var`, `llvm.obs.mem` and `llvm.io`, corresponding to the Clang builtins above; `llvm.io` is defined as I/O-effecting, `llvm.obs.mem`'s attributes let it read argument-pointed memory (this is actually an optimization feature as it avoids having to generate instructions loading from these memory locations), and `llvm.obs.var` is pure (it does not have any memory or I/O-effect).

To track observations effectively across compilation passes, and to let security engineers validate program properties using binary utilities (debuggers, monitors, model checkers of machine code, etc.), we need to embed so-called *observation metadata* into the IR, and preserve it all the way down to machine code. Observation metadata is formed of the observed variable/memory-address at a given program point (source code location). Interestingly, tracking such metadata is precisely what is happening with debug information. Unfortunately, tracking debug information is a best-effort

process in presence of optimizations, and experience with the validity of debug metadata in machine code can be chaotic. Fortunately, we do *not* need to burden the compiler with keeping track of renaming, combination, duplication, relocation effects, etc. for *general instructions*. Instead, we only need such a tracking mechanism for *opaque observations*. This can be implemented as a modular aspect of SSA variable renaming, and does not require instrumenting each and every pass of the compiler. Practically, we attach observation metadata to observation intrinsics via LLVM metadata [LLVM 2019], and we modify the utility functions `replaceAllUsesWith` (resp. `combineMetadata`) to handle the combination (resp. duplication) of observation intrinsics; the modified functions update (resp. maintain) the attached metadata throughout the compilation flow.

To preserve observations down to machine code generation, we lower the `llvm.obs.var`, `llvm.obs.mem` and `llvm.io` intrinsics into the MIR pseudo-instructions `OBS_VAR`, `OBS_MEM` and `IO`, respectively, with the same semantics. This is necessary since MIR is subject to a few late machine code optimizations before emitting assembly code. The three pseudo-instructions are eliminated before emitting assembly code. To facilitate this elimination, `OBS_VAR` holds the opaque value in the same register as its first operand; this includes opaque tokens implemented as integer variables, allowing for a zero-cost implementation for tokens.

Carrying observation metadata down to machine code is more challenging as LLVM does not support attaching metadata to the MIR. We choose to encode observation metadata into an operand of `OBS_VAR` and `OBS_MEM` pseudo-instructions. In theory, this could prevent some optimizations, such as combining pseudo-instructions with the same arguments but different metadata. Yet we did not observe performance regressions due to missed pseudo-instruction combining on our benchmark suite and the back-ends considered (see Section 8).

Finally, when eliminating MIR pseudo-instructions, we emit their observation metadata to the assembly code's debug section. We extend the DWARF format accordingly [DWARF 2017], capturing observation variables/addresses and line number with dedicated DWARF entities. The choice of a custom encoding of metadata differs from the more conventional approach relying on debug information [Vu et al. 2020] and offers stronger correctness and traceability guarantees.

5 SPECIFYING OBSERVATIONS

This section and the following one formalize the observation, opaque observation and opaque chain concepts introduced in the previous sections. 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 or any form of non-local control flow in the source language.[5]

5.1 Operational Semantics

We introduce a simple operational semantics, generic enough to model the side-effects, data and control flow of a range of imperative languages and intermediate representations (IRs), all the way down to assembly code. It is not meant to be faithful to a specific language. Instead, it serves as an abstraction of the essential properties of observations and their preservation. It takes the form of a state machine where every program instruction defines a transition referred to as an *event*.

A *program state* is defined by a tuple $\sigma = (Vals, \pi)$, where *Vals* denotes a set of (*name*, *value*) pairs; *name* may be an SSA variable (e.g. an SSA value in LLVM IR or a variable in a functional language), a constant, a reference name (e.g. a C variable, a reference in a functional language, or a

[5]Modeling non-local control flow would involve additional SSA arguments in branch and call instructions. This does not add interesting behavior to opaque observations and opaque chains but would make the formalization more verbose.

register in a machine-level representation) or a memory address; the *program point* π holds the value of the program counter pointing to the next instruction. A state σ is also interpreted as a function mapping names (variable, constant, reference, address) to the corresponding values.

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

A program execution E is a—potentially infinite—ordered sequence of program states and events: $E = \sigma_0 e_0 \sigma_1 e_1 \sigma_2 \ldots$, with e_0 a special initial event defining all constant values, σ_0 the initial state, and $\sigma_k \overset{i_k}{\rightsquigarrow} \sigma_{k+1}$ such that $\forall k > 0, i_k = Inst(e_k)$.

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 I/O operations through dedicated `io` instructions. Program input (resp. output) is modeled as a list of independent—potentially infinite—partially ordered sets of values, called input (resp. output) sets. Each set is identified with a unique descriptor. Every value of a given data type in an I/O set is uniquely tagged to distinguish it from any other value from the same set. A totally ordered set models streaming I/O; an unordered set models persistent storage; partial orders model middle-ground scenarios. The `io` instruction takes a descriptor and an I/O value as arguments; its execution yields an *I/O event*.

Given a program P, the sets *Inputs* (resp. *Outputs*) represent of all possible inputs (resp. outputs) of P. 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 $\mathcal{IO}[\![P]\!](I)$, and P produces a unique execution $\mathcal{E}[\![P]\!](I)$.

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

$$\forall \ldots e_1 \ldots e_2 \ldots \in E, \quad \big(Inst(e_1) = io(desc, v_1)\big) \wedge \big(Inst(e_2) = io(desc, v_2)\big) \implies e_1 \overset{\text{io}}{\rightarrow} e_2$$

This relation on events induces a relation on values in input and output sets, also denoted by $\overset{\text{io}}{\rightarrow}$:

$$\forall \ldots e_1 \ldots e_2 \ldots \in E, \quad \big(Inst(e_1) = io(desc, v_1)\big) \wedge \big(Inst(e_2) = io(desc, v_2)\big) \wedge e_1 \overset{\text{io}}{\rightarrow} e_2 \implies v_1 \overset{\text{io}}{\rightarrow} v_2$$

5.2 Modeling Program Transformations

We are now able to define a very general notion of program transformation. This definition is meant to cover as many compilation scenarios as possible and this constitutes a major strength of our proposal. In particular, we make no assumption on the analysis and transformation power of a compiler. The definitions covers any scalar, loop and inter-procedural optimization, canonicalization, lowering, etc. as well as dynamic schemes such as control and value speculation.

A transformation τ is valid if it preserves the I/O behavior of a program P on all possible inputs: $\forall I \in Inputs, \mathcal{IO}[\![P]\!](I) = \mathcal{IO}[\![\tau(P)]\!](I)$.

A transformation τ induces a relation between events before and after transformation; it is called the *event map* and denoted by $\propto_\tau$. The event map notation $e \propto_\tau e'$ reads as "τ maps e to e'". The mapping is partial and neither injective nor surjective in general (events in P may not have semantic counterparts in P' and vice versa). The event map will serve as a *transformation-independent translator to reason about the existence and ordering of events across program transformations.*

Let us immediately illustrate this notion, observing that for any valid transformation τ, one may construct an event map $\propto_\tau$ that preserves I/O events.

Lemma 5.1 (Transformation of 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, (1) there exists a unique event $e' \in \mathcal{E}[\![\tau(P)]\!](I)$ such that e' reads or writes v; and (2) τ preserves the partial ordering on all I/O events from E.

Proof. By definition of transformation validity, 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 $\mathcal{IO}[\![P]\!](I) = \mathcal{IO}[\![P']\!](I)$ implies that e' is the only event reading or writing v in the execution E'. This proves the unicity of transformed I/O events.

Given two I/O events e_1 and e_2 in E such that $e_1 \xrightarrow{\text{io}} e_2$, the unicity of transformed I/O events guarantees the existence of two events e'_1 and e'_2 in E' 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 input/output values v_1 from e_1 and v_2 from 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 $e'_1 \xrightarrow{\text{io}} e'_2$. This proves the preservation of I/O event ordering. □

Beyond I/O events, we will grow in the next section a larger set of events that are always guaranteed to have a counterpart through $\propto_\tau$, for all transformations.

In the following, we will characterize the events that are always related through $\propto_\tau$ for any valid transformation τ as *transformation-preserved events*. Let $TP(P, I)$ denote the set of transformation-preserved events for a program P and input I.

5.3 Observation Semantics

Let us now extend the operational semantics with a notion of *observation*. Vu et al. [2020] defined an *observation trace* formed of all observations in a program, as a sequence of sets of (*variable*, *value*) and (*address*, *value*) pairs called *partial observation states*, or *partial states* for short. Enforcing a total ordering of partial states and its preservation across program transformations limits the reach of compiler optimizations. To relax this restriction while still preserving the user's ability to attach logical properties to specific values and instructions, we extend the semantics with *partially ordered partial states* induced by a specific `observe` instruction. It takes as arguments an arbitrary number of values to be observed and returns the value of its first argument. Arguments may be constants, SSA variables, references or memory addresses. We call *observation event* any event associated with the execution of an `observe` instruction.

Formally, considering a program P and input I, with $E = \mathcal{E}[\![P]\!](I)$, the partial observation states of E are modeled by the following *observation function*:

$$
\begin{array}{rccl}
Obs_E : & Events(E) & \rightarrow & States(E) \\
 & \sigma \overset{\text{observe}(V)}{\rightsquigarrow} \sigma' & \mapsto & \left(\left\{(v, \sigma(v))\right\}_{v \in V}, \pi\right) \\
 & \sigma \overset{i \neq \text{observe}}{\rightsquigarrow} \sigma' & \mapsto & \left(\varnothing, \pi\right)
\end{array}
$$

Paraphrasing this definition, $Obs_E(e)$ returns the partial state made of $\{(name, value)\}$ pairs associated with observation e, such that π is the program point in P of $Inst(e)$, the instruction associated with e. $Obs_E(e)$ yields an empty partial state if e is not an observation event.

In the following, we will consider all I/O events as observations of their input and output values; this is consistent with the fact I/O yields "externally visible" effects and values.

As described Section 4, we also need to track observation metadata through every level of IR, down to machine code. To formalize this, we lift the observation function to return partial states *relatively to a program of reference* P_{ref}. Consider a program P deriving from P_{ref} through a series of program transformations and let $E = \mathcal{E}[\![P]\!](I)$. We define $Obs_{E/P_{ref}}(e)$ to return the partial

state $(\{(name, value)\}, \pi)$ where the values are those of $Obs_E(e)$, but the program point (π) and all variable names (*name*) refer to (and so are tracked back to) the ones of P_{ref}.

We are now able to formalize the preservation of program properties, binding them to the observation of source-level partial states and tracking these observations down to machine code.

5.4 Happens-Before Relation

Beyond the preservation of individual observations, we have seen that security properties can involve *the specification of an ordering among observation events*. Such an ordering relation must be respected by program transformations. We call it the *observation-ordering* relation. It is derived from def-use chains, value-based dependences through references and memory, and control dependences. Formally, we define relations $\overset{\text{du}}{\rightarrow}$, $\overset{\text{rf}}{\rightarrow}$ and $\overset{\text{cd}}{\rightarrow}$ as partial orders on def-use pairs, in-reference/in-memory data flow and control dependences, respectively. Let the predicate $def(v, i)$ (resp. $use(v, i)$) determine if instruction i defines (resp. uses) value v, and $write(addr, v, i)$ (resp. $read(addr, v, i)$) determine if instruction i writes v to (resp. reads v from) the memory address or reference name $addr$, and let *postdom* denote the post-domination binary predicate [Cytron et al. 1991]:

$$e_1 \overset{\text{du}}{\rightarrow} e_2 \text{ iff } def(v, Inst(e_1)) \wedge use(v, Inst(e_2))$$

$$e_1 \overset{\text{rf}}{\rightarrow} e_2 \text{ iff } write(addr, v, Inst(e_1)) \wedge read(addr, v, Inst(e_2)) \\ \wedge \forall e, E = \ldots e_1 \ldots e \ldots e_2 \ldots, \neg write(addr, v', Inst(e))$$

$$e_1 \overset{\text{cd}}{\rightarrow} e_2 \text{ iff } \exists e, E = \ldots e_1 \ldots e \ldots e_2 \ldots, postdom(Inst(e_2), Inst(e)) \wedge \neg postdom(Inst(e_2), Inst(e_1))$$

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

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

Given an execution E, one may now define the additional relations involving observation events that program transformations have to preserve.

Observations induce an *observe-from* relation, denoted by $\overset{\text{of}}{\rightarrow}$, which maps a definition to an observation event. Formally, $e_1 \overset{\text{of}}{\rightarrow} e_{obs}$ if and only if $e_1 \overset{\text{du}}{\rightarrow} e_{obs} \wedge Inst(e_{obs}) = \texttt{observe}$.

Symmetrically, observations induce a *from-observe* relation, denoted by $\overset{\text{fo}}{\rightarrow}$, which maps an observation event to a use. Formally, $e_{obs} \overset{\text{fo}}{\rightarrow} e_2$ if and only if $e_{obs} \overset{\text{du}}{\rightarrow} e_2 \wedge Inst(e_{obs}) = \texttt{observe}$.

Any pair of observation events in dependence relation are ordered by a so-called *observation ordering* relation denoted by $\overset{\text{oo}}{\rightarrow}$. Given an execution E of P, $\overset{\text{oo}}{\rightarrow}$ is the restriction of $\overset{\text{dep}}{\rightarrow}$ to observation events. Formally, $e_1 \overset{\text{oo}}{\rightarrow} e_2$ if and only if $e_1 \overset{\text{dep}}{\rightarrow} e_2 \wedge Inst(e_1) = \texttt{observe} \wedge Inst(e_2) = \texttt{observe}$.

$\overset{\text{oo}}{\rightarrow}$ only includes data-flow and control-dependences. 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 `observe` instructions enable the expression of arbitrary partial orders of observation events. Conversely, adding more relations into $\overset{\text{oo}}{\rightarrow}$ such as write-after-write and write-after-read dependences would severely restrict compiler optimizations—e.g. the compiler's ability to hoist loop-invariant observations like Vu et al.—with no expressiveness benefit.

One may now define a partial order on all observation events, including but not limited to I/O, called the *happens-before relation*. It has to be a sub-order of the total order of events in E. We use the following happens-before relation in the following: $\overset{\text{hb}}{\rightarrow} = \left(\overset{\text{io}}{\rightarrow} \cup \overset{\text{of}}{\rightarrow} \cup \overset{\text{fo}}{\rightarrow} \cup \overset{\text{oo}}{\rightarrow} \right)^*$.

Consider a valid program transformation τ and let $E' = \mathcal{E}[\![\tau(P)]\!](I)$. τ is said to preserve the happens-before relation if any events in happens-before relation in E have their counterparts through $\propto_\tau$ in happens-before relation in E' (if such counterparts exist).
Formally, $\forall e_i, e_j \in E, \forall e'_i, e'_j \in E', e_i \overset{\text{hb}}{\rightarrow} e_j \wedge e_i \propto_\tau e'_i \wedge e_j \propto_\tau e'_j \implies e'_i \overset{\text{hb}}{\rightarrow} e'_j$.

Unlike I/O ordering enforced by any valid transformation (Lemma 5.1), preserving the happens-before relation requires additional effort. This is where opacity will come into play; it will be described in Section 6.

Back to our running example, each observation of an incrementation of `i2` happens before the next incrementation of `i2`, and before the call to the fault handler which is a control-dependent I/O (both are modeled in $\overset{\text{oo}}{\rightarrow}$). This allows to reason about loop termination in the presence of faults. Conversely, notice the absence of a happens-before relation linking incrementations of `i`. Optimizations are free to replace `i` with a closed form expressions (analysis of induction variables) or to reorder the respective arithmetic operations on `i` and `i2`. Such optimizations would not be possible with the I/O-effecting observations proposed by Vu et al. [2020].

5.5 Putting It All Together: Protected Observations

Let us now provide two important definitions to characterize the protection of observations against interference from program transformations. By protection, we refer to the preservation of observation events, the preservation of the observed partial states, and the preservation of the happens-before relation, for all valid transformations.

Definition 5.2 (Protected observation). For a given program P, an observation instruction i_{obs} is *protected* if and only if every valid transformation τ satisfies the three following conditions:

(i) τ preserves the existence of observation events:

$$\forall e \in \mathcal{E}[\![P]\!](I), \quad i_{obs} = Inst(e) \implies \exists e' \in \mathcal{E}[\![\tau(P)]\!](I), e \propto_\tau e'$$

(ii) τ preserves the partial states observed in P:

$$\forall e \in E, \forall e' \in \mathcal{E}[\![\tau(P)]\!](I), \quad i_{obs} = Inst(e) \wedge e \propto_\tau e' \implies Obs_E(e) = Obs_{\mathcal{E}[\![\tau(P)]\!](I)/P}(e')$$

(iii) τ preserves the happens-before relation.

An observation instruction i_{obs} is *protected conditionally on instruction* i_c if and only if every valid transformation τ satisfies (ii) and (iii) above, as well as the following condition replacing (i):

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

$$\forall e \in \mathcal{E}[\![P]\!](I), \quad \exists e_c \in \mathcal{E}[\![P]\!](I), \exists e'_c \in \mathcal{E}[\![\tau(P)]\!](I),$$
$$i_c = Inst(e_c) \wedge e_c \propto_\tau e'_c \wedge i_{obs} = Inst(e) \implies \exists e' \in \mathcal{E}[\![\tau(P)]\!](I), e \propto_\tau e'$$

Note that both e and e' are observation events in (i), (i_c) and (ii). Also, the observation of e' in (ii) is defined with P as a reference: the values observed in e' are those of variables in $\mathcal{E}[\![\tau(P)]\!](I)$ tracked back to their original (*name*, *value*) pairs in $\mathcal{E}[\![P]\!](I)$.

In addition, since composing two valid transformations yields a valid transformation, the characterization of observation protection covers compositions of valid transformations along a compilation pass pipeline. The next section provides a portable method to implement protected observations in an optimizing compiler.

6 EFFECTIVELY PRESERVING OBSERVATIONS

As noted earlier, valid transformations do not preserve observations and the happens-before relation in general. This section formalizes opaque observations as a mechanism to achieve this.

6.1 Opaque Observations

To implement the preservation of observation events and the associated happens-before relation, we complement the observation semantics by making observation events *opaque*: in addition to capturing all arguments into a partial observation state and implementing the identity function in its first argument, the `observe` instruction makes the returned value *opaque*. From now on, observation events are called *opaque observation events*, or *opaque observations* for short. Unlike Vu et al. [2020] embedding I/O effects into all observation events, `observe` is a pure function. Since we consider I/O events as observations, we also consider them as opaque in the following; unlike `observe`, `io` provides an I/O-effecting opaque observation.

As informally introduced, opacity means the compiler sees a statically unknown yet functionally deterministic value. In particular, it does not know that `observe` returns its first argument. Given an input I and valid transformation τ, opacity introduces two validity restrictions supporting more cases of value and event preservation:

upstream opacity: if τ preserves an event e_2 that uses a value defined by an opaque observation e_1, it must also preserve e_1 itself (as a transformation has no other means to produce the opaque value defined by e_1):

$$\begin{aligned}\forall \ldots e_{obs} \ldots e_{use} \ldots \in \mathcal{E}[\![P]\!](I),\\ e_{obs} \xrightarrow{\text{fo}} e_{use} \wedge \exists e'_{use} \in \mathcal{E}[\![\tau(P)]\!](I), e_{use} \propto_\tau e'_{use}\\ \implies \exists e'_{obs} \in \mathcal{E}[\![\tau(P)]\!](I), e_{obs} \propto_\tau e'_{obs} \wedge Inst(e'_{obs}) = \texttt{observe} \wedge e'_{obs} \xrightarrow{\text{dep}} e'_{use}\end{aligned} \quad (1)$$

downstream opacity: τ must preserve any value used by a preserved opaque observation (otherwise downstream computation would need to guess the opaque observation's behavior, which is not allowed):

$$\begin{aligned}\forall \ldots \sigma e_{obs} \ldots \in \mathcal{E}[\![P]\!](I), \forall \ldots \sigma' e'_{obs} \ldots \in \mathcal{E}[\![\tau(P)]\!](I),\\ Inst(e_{obs}) = \texttt{observe} \wedge e_{obs} \propto_\tau e'_{obs} \wedge use(v, Inst(e_{obs})) \wedge (v, val) \in \sigma\\ \implies \exists (v', val) \in \sigma', use(v', Inst(e'_{obs}))\end{aligned} \quad (2)$$

These restrictions are taken as a definition, formalizing the intuitive expectations about what the compiler has to enforce in the presence of opaque observations. Notice the transitive dependence relation $e'_{obs} \xrightarrow{\text{dep}} e'_{use}$ in the transformed program (rather than $e'_{obs} \xrightarrow{\text{dep}}{}^{1} e'_{use}$): the immediate dependence may be transformed into a series of instructions (e.g., spilling a value to the stack).

6.2 Opaque Chains

Let us now build a specific class of dependence chains involving opaque observations. As informally discussed in the introduction, these are called *opaque chains* and serve two purposes: (1) preserving observations by linking them to downstream transformation-preserved events (e.g., I/O events), and (2) establishing a transformation-preserved happens-before relation. We first need a technical definition, the *opaque chain value set*, holding all possible values observed by an instruction $i_{observe}$ after traversing a chain of dependences linking an upstream opaque observation i_{opaque} to a downstream one $i_{observe}$. Whether this set is a singleton will tell if the dependent instruction $i_{observe}$ is sensitive on the (opaque) value of i_{opaque}. Intuitively, a non-singleton set tells that evaluating the upstream opaque observation i_{opaque} is the only way to provide the downstream observation with a correct value, hence that i_{opaque} must be preserved by any valid transformation.

Let $Dom(v)$ denote the set of values that a variable v may take according to its data type. E.g., for a value v of boolean type, $Dom(v) = \{\text{true}, \text{false}\}$.

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 $n \geq 2$, two *opaque observations* $i_j = Inst(e_j)$ and $i_k = Inst(e_k)$ on the chain with $1 \leq j < k \leq n$ such that $\forall j < l < k, Inst(e_l) \neq \texttt{observe}$. Let v_j be the variable defined by i_j and used along the chain, and consider the state σ_{j+1} after executing e_j. For any value $alt \in Dom(v_j)$, we note $E_{alt} = \ldots e_j \sigma_{j+1}\{v_j \mapsto alt\} \ldots$ the execution continuing after e_j on program state $\sigma_{j+1}\{v_j \mapsto alt\}$[6].

In the following, opaque observations are considered *equivalent* if swapping them yields the same partial state for every input; e.g. observations are considered equivalent up to variable renaming.

We define the *opaque chain value set* $OCVS_{j,k}$ from all the values derived from $Dom(v_j)$ according to one of the two cases below (and only these):

- if there exists an opaque observation event $e_{k_{alt}}$ such that $E_{alt} = \ldots e_j \ldots e_{k_{alt}} \ldots$ and $i_{alt} = Inst(e_{k_{alt}})$ and i_k are equivalent with $e_j \overset{\text{dep}}{\rightarrow} e_{k_{alt}}$ and $\forall e,\ e_j \overset{\text{dep}}{\rightarrow}^+ e \overset{\text{dep}}{\rightarrow}^+ e_{k_{alt}}, Inst(e) \neq \texttt{observe}$ then the value used or read by i_{alt} along the $e_j \ldots e_{k_{alt}}$ sub-chain belongs to $OCVS_{j,k}$;
- if there is no such event $e_{k_{alt}}$ before reaching another (non-equivalent) opaque observation or the program terminates then $\perp$ belongs to $OCVS_{j,k}$.

Let us paraphrase this definition. When substituting the value of the opaque observation i_j with *alt* three situations may occur: (1) the $OCVS_{j,k}$ set yields the value used or read by i_k if the execution path of the altered execution is not changed; (2) if the execution path is altered and reaches an equivalent opaque observation $i_{alt} = Inst(e_{k_{alt}})$ before encountering any other dependent opaque observation, it yields the value used or read by i_{alt}; and (3) if the altered execution does not reach an equivalent instruction before reaching an opaque observation (or the program terminates), $OCVS_{j,k}$ holds the "undefined" value $\perp$. Note that the execution path may be altered when the opaque chain traverses a control dependence on an opaque observation i_j: alternate opaque values may exercise different paths, some of which may not reach the next opaque observation i_k anymore.

We may now define an opaque chain as an alternating sequence of opaque observations and sub-chains of regular instructions, starting with an opaque observation and ending with an opaque observation (remember I/O events are considered opaque observations).

Definition 6.1 (Opaque chain). Given an execution $E = \mathcal{E}[\![P]\!](i)$, consider a chain of dependent events $e_1 \overset{\text{dep}}{\rightarrow}^1 \ldots \overset{\text{dep}}{\rightarrow}^1 e_n$; $e_1 \overset{\text{dep}}{\rightarrow}^1 \ldots \overset{\text{dep}}{\rightarrow}^1 e_n$ 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 observations;
(ii) for any opaque observation $i_k = Inst(e_k)$, $2 \leq k \leq n$, with i_j, $1 \leq j < k$, the immediately preceding opaque observation on the chain, $|OCVS_{j,k}| \geq 2$.

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

Case (ii) serves as an "information-carrying" guarantee: the compiler lacks information about the possible paths or computations dependent on an opaque value to break an opaque chain into distinct dependence chains. Given an instruction/event $i_k = Inst(e_k)$, for the immediately upstream opaque observation $i_j = Inst(e_j)$ on the chain, we consider all values it may define according to its opaque result type. If i_k is data-dependent on i_j, the set of values i_k may use or read must not be a singleton; unless i_k is control-dependent on i_j, in which case there must exist an alternate execution from e_j bypassing i_k and any equivalent instruction.

6.3 Opaque Chain Examples and Counterexamples

Let us consider examples of dependence chains that are not opaque chains. First of all, 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 observation into one applied to the constant

[6] *alt* may take any value in $Dom(v_j)$. The substitution syntax $\sigma_{j+1}\{v_j \mapsto alt\}$ denotes the set $\sigma_{j+1} \setminus (v_j, orig) \cup (v_j, alt)$.

zero, hence breaking the chain. The cardinality requirement on data dependences forbids such information-erasing instructions to occur on opaque chains.

As a more complex example, consider the code snippet in Listing 10 illustrating the subtleties of dealing with multiple paths. `STDOUT` is the standard output stream, and `val` is some statically unknown value. The program forms two dependence chains from the definition of `c` (line 1) to the output I/O instructions (lines 2 and 3) through a control dependence. Now consider its "tail merging" transformation into Listing 11: there is no dependence anymore in the transformed program. The dependence chain in the original program is not opaque: due to the equivalent I/O instruction `io(STDOUT, 0)` on both paths, $OCVS_{1,2} = OCVS_{1,3} = \{0\}$, hence $|OCVS_{1,2}| = |OCVS_{1,3}| = 1$.

```
1 bool c = observe(val);
2 if (c) io(STDOUT, 0);
3 else io(STDOUT, 0);
```

Listing 10. Control dependence

```
1 bool c = observe(val);
2 io(STDOUT, 0);
```

Listing 11. Eliminated by tail merging

```
1 bool c = observe(val);
2 if (c) v = 42;
3 else v = 100;
4 io(STDOUT, v);
```

Listing 12. P_{data}

```
1 bool c = observe(val);
2 if (c) io(STDOUT, 42);
3 else io(STDOUT, 100);
```

Listing 13. $P_{control}$

On the contrary, Listing 12 and Listing 13 illustrate the robust back-and-forth conversion of data and control dependences in opaque chains. Both programs form opaque chains from the definition of `c` to the output of the value of `v` (42 or 100). The transformation from P_{data} to $P_{control}$ and vice-versa are both valid, preserving the opaque observation (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` (P_{data}) or the alternative path from the definition of `c` to an output I/O instruction ($P_{control}$), it is impossible for the compiler to break the dependence. Formally, on P_{data}: $OCVS_{1,4} = \{42, 100\}$; while on $P_{control}$: $OCVS_{1,2} = \{42, \bot\}$ and $OCVS_{1,3} = \{100, \bot\}$; in all cases the cardinal is 2.

Beyond opaque observations, important classes of instructions are always compatible with the hypotheses of an opaque chain:

- all instructions that only propagate existing values; these include dereference, assignment, load, store, return instructions;
- the same applies to the traditional C unary operators `-`, `!`, `~`;
- 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;
- any binary operator or function call with non-correlated opaque arguments (feeding the same opaque value multiple times may degenerate into a singleton *OCVS* set, such as the subtraction of an opaque value with itself).

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 finally illustrate these definitions on the observation-extended running example in Listing 3. The dependences linking incrementations of `i2` among themselves and with the downstream fault handler form an opaque chain: all operations on opaque values belong to the above-mentioned

classes of instructions that are always opaque-chain-compatible. These chains extend to the `return` instructions and fault handler through control dependences; the existence of an alternate control flow path makes these extended chains opaque.

6.4 Application to Observation Protection

Let us now generalize Equation (1) to opaque chains. If a valid transformation preserves the tail event of an opaque chain, then it must also preserve the event at its head.

Theorem 6.2 (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 valid transformation τ mapping e_1 to e'_1 and e_n to $e'_{n'}$, there is a dependence chain linking e'_1 to $e'_{n'}$. Formally, let $\mathcal{T}(P)$ denote the set of all valid transformations of P,*

$$e_1 \overset{\text{opaque}}{\rightsquigarrow} e_n \wedge e_n \in TP(P,I) \implies \forall \tau \in \mathcal{T}(P), \exists e'_1, e'_{n'} \in \mathcal{E}[\![\tau(P)]\!](I), e_1 \propto_\tau e'_1 \wedge e_n \propto_\tau e'_{n'} \wedge e'_1 \overset{\text{dep}}{\rightarrow} e'_{n'}$$

Proof. The $\overset{\text{opaque}}{\rightsquigarrow}$ relation implies e_1 and e_n are opaque. The case of $n = 1$ is trivial.

Consider the case of $n \geq 2$. Let $i_k = Inst(e_k)$ be the immediately upstream opaque observation of e_n in the chain. Consider the $e_k \dots e_n$ sub-chain (which is trivially an opaque chain). By definition of the $\overset{\text{opaque}}{\rightsquigarrow}$ relation, $|OCVS_{k,n}| \geq 2$. In other words, the sub-chain (excluding e_k and e_n) implements the function mapping $Dom(v_k)$ to $OCVS_{k,n}$, and it is sensitive to the value d defined by e_k (i.e., non-constant along the values d may take). This implies the existence of a slice of events in $\mathcal{E}[\![P]\!](I)$, spawning backward from e_n and including an event e_u using d. It is trivial that $e_k \overset{\text{dep}}{\rightarrow}{}^1 e_u$. From Equation (2), e_n being opaque, the mapping of e_n to $e'_{n'}$ implies that the same value in $OCVS_{k,n}$ is used by $e'_{n'}$. As a result, $\tau(P)$ also compute a function sensitive to d mapping $Dom(v_k) = Dom(v_{k'})$ to $OCVS'_{k',n'}$, which is in turn computed by a slice in $\mathcal{E}[\![\tau(P)]\!](I)$ spawning backward from $e'_{n'}$. The sensitivity of this function to d implies that the backward slice holds an event $e'_{u'}$ using d, such that $e_u \propto_\tau e'_{u'}$. We may apply Equation (1) to $e_k \overset{\text{dep}}{\rightarrow}{}^1 e_u$ and $e_u \propto_\tau e'_{u'}$, which proves the existence of an opaque observation event $e'_{k'} \in \mathcal{E}[\![\tau(P)]\!](I)$ such that $e_k \propto_\tau e'_{k'}$ and $e'_{k'} \overset{\text{dep}}{\rightarrow} e'_{n'}$.

An induction on the number of opaque observations of the chain proves the preservation of e_1 for all chain lengths, and that transformed events form a dependence chain $e'_1 \overset{\text{dep}}{\rightarrow} e'_{n'}$. □

Notice that events associated with non-opaque instructions along the chain are not necessarily preserved. Furthermore, we did not prove that an opaque chain transforms into an opaque chain in general: Theorem 6.2 only establishes that the transformed observation events in an opaque chain are dependent, not that they form another opaque chain. Stronger hypotheses on opaque chains are likely to be needed if we wished to do so. Fortunately, we do not need to further constrain (and complicate) the definition of opaque chains. Theorem 6.2 applies to compositions of valid transformations, which are valid transformations themselves. In practice, it is sufficient to refer to the source program, modeling a sequence of compilation passes as a single transformation.

We may finally present the formalization's main result: the protection of opaque observations through opaque chains for arbitrary valid transformations.

Theorem 6.3 (Protected opaque observation). *Let P be a program implementing observations through opaque chains enforcing a programmer-specified $\overset{\text{hb}}{\rightarrow}$ order, and such that any chain leads to a downstream protected observation (resp. a downstream opaque observation protected conditionally on some instruction in a set $\mathcal{I}_c$). Then all opaque observations in P are protected (resp. protected conditionally on instructions in $\mathcal{I}_c$) according to Definition 5.2.*

Proof. Consider any valid transformation τ. We need to prove that τ preserves observations, according to Definition 5.2. Since opaque chain terminate with a protected observation (conditionally on some instruction i_c or not), Theorem 6.2 applied to opaque chains linking opaque observation instructions to a subsequent protected (resp. conditionally protected) observation instruction proves condition (i) (resp. condition (i_c)). From Equation (2), any valid transformation must preserve the values of opaque observation arguments $v_1, \ldots, v_k$, along with the definition events producing these values. This proves condition (ii). Let us now prove condition (iii)—the preservation of $\overset{\text{hb}}{\longrightarrow}$. The preservation of $\overset{\text{fo}}{\longrightarrow}$ is an immediate corrolary of Equation (1). The preservation of $\overset{\text{of}}{\longrightarrow}$ is analogous to the proof of condition (ii). The preservation of $\overset{\text{oo}}{\longrightarrow}$ involves the traversal of opaque chains: consider a pair of opaque observation events e_{obs_1}, e_{obs_2}, such that $e_{obs_1} \overset{\text{oo}}{\longrightarrow} e_{obs_2}$. Since $e_{obs_1} \overset{\text{opaque}}{\rightsquigarrow} e_{obs_2}$, Theorem 6.2 applies to all opaque events on the opaque chain, guaranteeing their mapping to events in $P' = \tau(P)$ and that these events form a dependence chain. This proves the existence of e'_{obs_1} and e'_{obs_2}—the counterparts of e_{obs_1} and e_{obs_2} in P'—as well as $e'_{obs_1} \overset{\text{oo}}{\longrightarrow} e'_{obs_2}$. □

7 VALIDATION: FUNCTIONAL CORRECTNESS AND SECURITY

This section analyzes the correctness our approach and implementation.

7.1 Functional Validation by Checking Partial State 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, at the proper memory address or associated with the appropriate variable, and ordered w.r.t. other observations according to the happens-before relation. To this end, we leverage *observation traces*, the sequence of partial states associated with observation events encountered during a given execution [Vu et al. 2020]. Practically, validation involves comparing, for a given program input, two observation traces:

(1) the *reference trace*, obtained by executing an instrumented version of the source program with `printfs` in place of observations, compiled without any optimization;[7]
(2) the *optimized trace*: obtained by executing the program with opaque observation builtins, compiled at different optimization levels; we extend a DWARF parser [Eli Bendersky 2011] to map all observation points to breakpoints in machine code, as reported in the DWARF section, recording the values and their storage locations at every observation point; we use a debugger to save these (address, value) pairs during program execution.

To compare the traces, we associate every partial state 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 each partial state in the reference trace has a corresponding counterpart with the same identifier in the optimized trace, and vice versa. 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.

We validate functional correctness on a subset of the Frama-C test suite, a static analysis framework for C programs [Cuoq et al. 2012]. The test suite exercises Frama-C analyses on a range of programs representative of the C language semantics, using program properties written in the ACSL annotation language [Baudin et al. 2008]. We restrict ourselves to properties assessing the values of variables at a given program point—easily be expressed as observation events—ignoring more advanced ACSL constructs. Unlike Vu et al., our approach only enforces a partial ordering on program observation events: there is no particular constraint for the relative order of observation

[7]We assume `-O0` preserves the observation events as well as the partial state of the ISO C abstract machine [ISO 2011] containing the observed values of each event.

events having no data dependence relation. Furthermore, for the considered programs, there is no dependence relation between observation events. As a result, we propose a dual validation methodology: for every test case in the suite, we derive (i) an unordered version where a distinct token is produced from every observation event and immediately consumed in a distinct I/O instruction (the I/O effect being decoupled from the observation, it does not constrain the ordering of observation events), and (ii) a totally ordered version where all events are chained by reading from and writing to the same variable. The totally ordered version is used to validate the preservation of observation events and their ordering, by verifying that reference and optimized traces are *identical*. As for the unordered version, we verify the presence of all observation events in the compiled programs by checking the two-way mapping of partial states in reference and optimized traces.

We compile both versions of every test case at 6 optimization levels `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Oz`. This results in two sets—undordered (i) and ordered (ii)—of 31 applicable test cases featuring 616 observations. Notice that these test cases are not meant to be evaluated as performance benchmarks: they are code snippets (a dozen lines of code) and are not provided with validation data sets; we only use them to validate the correctness of our implementation. We automatically verified that in both sets, all 616 observations have been correctly propagated to machine code, and for the first set (ii), the observation trace is identical to the reference trace.

7.2 Security Protection Preservation Validation

Validating the preservation of security protections is more challenging. While verifying partial state integrity is enough to prove the preservation of observation events, it is only a necessary condition for preserving security protections. Hence, considering our selection of security applications from Section 3, we also define additional mechanisms to validate specific security properties.

- Value utilization: beyond the observation of partial states, we also need to make sure that the observed values are effectively used in downstream security protections. To this end, we implement a two-phase verification: (1) determine the uses of observed values in the program, verifying that they are indeed part of the source code protection, (2) for every observed operand of every C expression of a security protection—e.g. a masked key or SCI counter check—verify that machine code uses a value defined by an opaque observation.
- Statement ordering: for SCI, we also need to check the proper interleaving of functional and countermeasure statements.
- Constant-time selection: branching conditionally on a sensitive value is forbidden; as a result, we verify that none of the conditional branches (indirectly) depend on the secret value, including the selection bitmask derived from the boolean secret or its observed value.

These validation mechanisms have all been applied manually, by inspecting the source code and disassembling the generated machine code.

Table 1 summarizes the results of applying the validation schemes to our selected security benchmarks. For every benchmark we could verify that the schemes appropriate to the benchmark's security properties yield the expected results, validating our approach and implementation.

Table 1. ✓ indicates the scheme is *validated* for the benchmark and n/a that it is not relevant.

	`era-*`	`mask-*`	`loop-pin`	`sci-*`	`ct-*`
Partial state 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	✓

8 PERFORMANCE EVALUATION

Let us now turn to performance and compilation time experiments.

8.1 Experimental Setup

For every security application presented in Section 7, we first compare our opaque observation approach against the unoptimized version—which is also a solution to preserving security protections. We then compare against other preservation mechanisms available, namely compiler-dependent programming tricks for constant-time selection [Simon et al. 2018], and Vu et al.'s I/O-based approach for all other applications. 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 inline assembly approach that does not involve modifications to Clang and LLVM.

We target two different instruction sets: ARMv7-M/Thumb-2, representative of deeply embedded devices, and Intel x86-64, representative of high-end processors with a complex micro-architecture. To evaluate our approach when conditional moves may be expanded into branches [Simon et al. 2018], we also consider IA-32 for constant-time applications `ct-rsa` and `ct-mgmr`. Performance evaluation for the ARMv7-M/Thumb-2 ISA takes place on an MPS2+ board with a Cortex-M3 clocked at 25 MHz with 8 MB of SRAM, while the Intel test bench (used for both x86-64 and IA-32 targets) and has a quad-core 2.5 Ghz Core i5-7200U CPU with 16 GB of RAM. We use the Intel platform to compile for all targets. Switching targets only concerns the back-end, a short part of the compilation pipeline; as a result, compilation times are very similar and we only report compilation time for the ARMv7-M/Thumb-2 ISA. Our experiments cover all common optimization levels (`-O1`, `-O2`, `-O3`, `-Os`, `-Oz`). Table 2 collects the number of repetitions of the kernel function, the size and execution time of all compiled binaries. Sizes are for the unstripped binaries, accounting for all debug and observation metadata.

8.2 Performance Evaluation

Figure 1 presents the speedup 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 opaque observation approach to preserving security protections enables aggressive optimizations with significant benefits over `-O0`.

Figure 2 presents the speedup our approach compared to reference preservation mechanisms, at different optimization levels. For `era-rsa-enc`, `era-rsa-dec`, `mask-aes`, `mask-rotate`, `loop-pin`, `sci-pin` and `sci-aes`, we compare our approach against Vu et al. The authors introduced new intrinsics to implement their property preservation mechanism, relying heavily on I/O effects. In particular, these intrinsics chain upstream definitions to downstream observations through I/O-effecting artificial definitions. In order to maintain the consistency of the debug information for observed values, the authors inserted additional 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. Overall, this results in spurious ordering constraints and missed optimizations due to the high density of I/O-effecting instructions. As a consequence, our implementation with pure intrinsics (no side effects), enables more optimizations, resulting in faster code. 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

Table 2. Repetitions, execution time and binary size for all benchmarks and targets.

		era-rsa-enc	era-rsa-dec	mask-aes	mask-rotate	loop-pin	sci-pin	sci-aes	ct-rsa	ct-mgmr
Number of repetitions of the security-sensitive function										
	x86-64	10^5	10^4	10^7	10^{10}	3×10^8	10^8	10^5	10^4	200
	ARMv7-M	1	1	1	10	120	2	1	1	1
	IA-32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	10^4	200
Execution time (in seconds for x86-64 and IA-32, in kcycles for ARMv7-M)										
-O1	x86-64	1.848	4.307	9.237	15.39	2.619	4.971	3.768	4.28	1.374
	ARMv7-M	3961	13979	14	0.3	0.2	1.7	448	12396	263963
	IA-32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	8.7	5.642
-O2	x86-64	1.848	4.307	8.606	4.236	2.384	4.024	3.895	3.286	1.134
	ARMv7-M	3961	13978	13	0.06	0.17	1.6	406	11036	217161
	IA-32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	7.634	5.44
-O3	x86-64	1.848	4.307	8.598	4.237	2.11	4.485	3.831	3.244	1.128
	ARMv7-M	3961	13978	13	0.06	0.15	1.6	406	10486	217508
	IA-32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	7.598	4.84
-Os	x86-64	1.848	4.307	9.636	6.239	2.617	4.9	3.25	4	1.322
	ARMv7-M	3961	13978	13	0.14	0.2	1.7	458	12067	280746
	IA-32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	8.556	4.788
-Oz	x86-64	1.848	4.307	10.273	17.057	3.769	5.763	3.836	4.298	1.412
	ARMv7-M	3961	13978	17	0.4	0.19	1.8	459	12922	291721
	IA-32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	9.106	5.93
Binary size (in KB)										
-O1	x86-64	489	494	24	11	14	44	62	495	128
	ARMv7-M	504	505	66	90	64	100	123	505	192
	IA-32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	444	104
-O2	x86-64	618	619	25	13	15	45	74	619	212
	ARMv7-M	636	637	68	91	65	103	132	641	278
	IA-32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	544	163
-O3	x86-64	661	666	25	13	17	47	83	663	225
	ARMv7-M	674	675	68	91	66	104	135	680	287
	IA-32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	578	167
-Os	x86-64	485	490	24	11	14	45	57	492	133
	ARMv7-M	511	512	67	90	64	102	124	515	198
	IA-32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	454	112
-Oz	x86-64	474	475	23	11	14	44	59	476	122
	ARMv7-M	486	487	66	90	64	102	123	489	194
	IA-32	n/a	n/a	n/a	n/a	n/a	n/a	n/a	436	101

benefits with our approach at these optimization levels. On the contrary, for `era-rsa-enc` and `era-rsa-dec`, the function implementing the protection only contains the erasure of the sensitive buffer, we thus observe almost no performance difference with Vu et al. Similarly for `mask-aes`, data required for mask computations is stored in memory (global variables) and there is almost no difference between the two versions. As for other applications, we note a clear improvement for both targets, 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 and inevitably degrade performance.

For `ct-rsa`, we compare our approach against the constant-time selection implementation of mbedTLS [Paul Bakker, ARM 2019], which is basically the same as Listing 8, but with the computation of the bitmask (line 3) outlined into a separate function, marked as non-inlinable in an attempt to prevent the compiler from optimizing it away. For `ct-mgmr`, we compare our approach against the specially-crafted implementation of OpenSSL [The OpenSSL Project 2003]. Now, general-purpose compilers offer no guarantees of preserving constant-timeness: future versions of the same compiler may spot the trick turn it back to a (probably faster) time-sensitive implementation [Simon et al. 2018]. Our approach allows constant-time selection functions to be safely inlined; yet these only take a small fraction of the execution time and we do not notice a clear difference with other constant-time implementations.

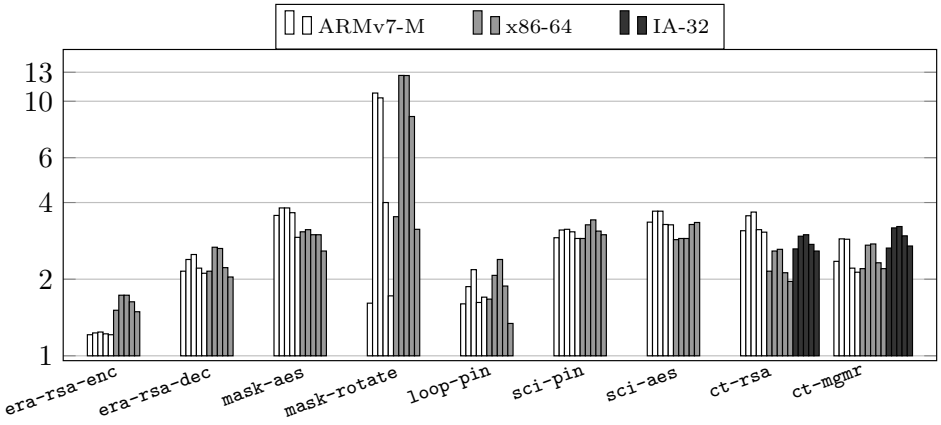

Fig. 1. Speedup over unoptimized programs—ordered by optimization level -O1,2,3,s,z.

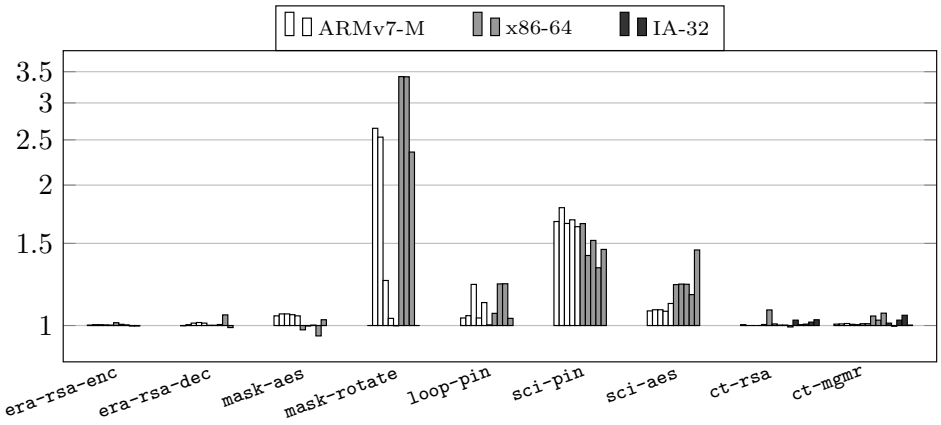

Fig. 2. Speedup over Vu et al. [2020] (for `era-rsa-enc`, `era-rsa-dec`, `mask-aes`, `mask-rotate`, `loop-pin`, `sci-pin`, `sci-aes`) and programming tricks [Simon et al. 2018] (for `ct-rsa`, `ct-mgmr`)—ordered by optimization level -O1,2,3,s,z.

8.3 Compilation Time

Figure 3 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.

In general, the overhead is under 10%, except for `sci-aes` and `sci-pin` where it ranges from 13% up to 70%. As discussed in Section 3.2.1, the SCI protection represents a very important fraction of the code due to the interleaved counter incrementations. As a consequence, code size increases significantly with fully-preserved countermeasures, which justifies the compilation time overhead.

8.4 Alternative Implementations

Compilers provide an *inline assembly* syntax to embed target-specific assembly code in a function. This is used by operating system programmers and for low-level optimizations, and also for sensitive

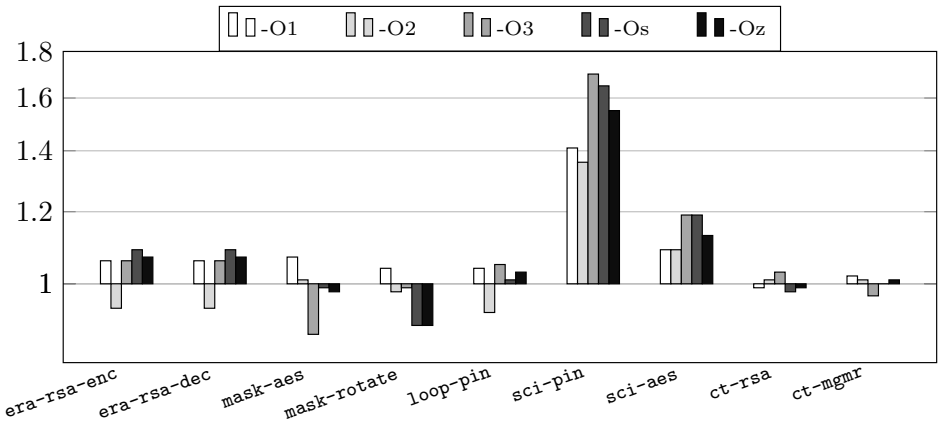

Fig. 3. Compilation time overhead over the original (insecure) benchmark.

applications to avoid interference from the compiler [Rigger et al. 2018]. GCC-compatible compilers implement an inline assembly syntax with explicit inputs, outputs, side-effects and I/O effects [Stallman and DeveloperCommunity 2009], standing as a contract between opaque assembly code and the compiler. We may thus leverage this feature to implement opaque observation. More specifically, the opaque observation of a C variable `v` is made up of an empty inline assembly region, with the variable set as input and output of the region, so that `v` is opaque to the compiler after evaluation of the inline assembly expression, just as if it was defined by `__blt_obs_var`.

While this approach does make observations opaque, it complicates the implementation of observations, and more specifically 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 metadata.

Now, the natural question is to compare the performance of an approach based on inline-assembly with our compiler-native implementation of opaque observations. To this end, we consider a subset of the applications presented in Section 3, made of `era-rsa-enc`, `era-rsa-dec`, `mask-aes`, `mask-rotate`, `loop-pin`, `ct-rsa` and `ct-mgmr`. 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 4 presents the speedup of our compiler-native implementation w.r.t. inline-assembly, at different optimization levels.

In the majority of cases, both implementations generate the same assembly code. For `ct-rsa` and `ct-mgmr`, the slight performance difference is due to discrepancies in register allocation. There is a significant performance difference 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 version happens not to be unrolled, while the compiler-native version is. Interestingly, this is only the case for x86-64: the same loop is always unrolled 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 of its side-effects—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 similar secure development scenarios (monitoring is slightly more difficult with inline assembly). Yet this may

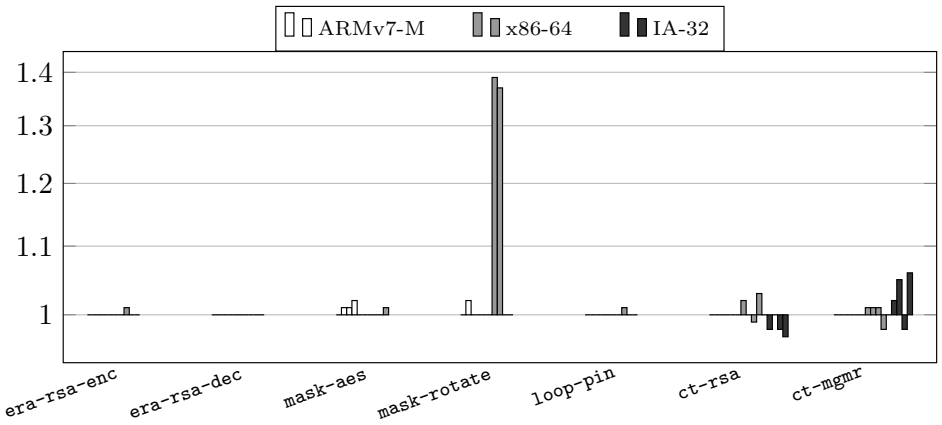

Fig. 4. Speedup over inline assembly—ordered by optimization level -O1,2,3,s,z.

not always be the case in the future: compilers are not forbidden to analyze inline assembly and take optimization decisions violating the opacity hypothesis.

9 CONCLUSION

We addressed a fundamental, open issue in security engineering: preserving security protections through optimizing compilation. We formalized the notion of observation and its preservation through program transformations. We instantiated this definition and preservation mechanisms from source code down to machine code. The approach relies on fundamental principles of compiler correctness: (1) the preservation of I/O effects and (2) the interaction of data dependences with constructs that are opaque to static analyses. We proved the correctness of the approach, and validated it within the LLVM framework with virtually no change to existing compilation passes.

There are potential applications beyond secure compilation. Some uses of the `volatile` keyword could be replaced with opaque observations, enabling finer grained optimizations on kernel code. Avenues for further research also include software engineering scenarios such as testing of production code without instrumentation with runtime checks, and more robust debugging of optimized code.

REFERENCES

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

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

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

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

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

aiT 2003. aiT. https://www.absint.com/ait/index.htm. Accessed 19 May 2018.

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

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.

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

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

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

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.* Frama-C. https://frama-c.com/download/acsl.pdf

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.

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

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

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

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

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

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct. 1991), 451–490. https://doi.org/10.1145/115372.115320

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

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.

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

Paul Bakker, ARM. 2019. mbedTLS. tls.mbed.org

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

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

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

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.

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

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*.

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

Daan Sprenkels. 2020. *Side-channel resistant values.* LLVM Foundation. http://lists.llvm.org/pipermail/llvm-dev/2019-September/135079.html

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

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

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

Marc Witteman. 2018. *Secure Application Programming in the Presence of Side Channel Attacks.* Technical Report.

Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. 2017. Dead Store Elimination (Still) Considered Harmful. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 1025–1040.

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

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