Secure Composition of Robust and Optimising Compilers

MATTHIS KRUSE, CISPA Helmholtz Center for Information Security and Saarland University, Germany
MICHAEL BACKES, CISPA Helmholtz Center for Information Security, Germany
MARCO PATRIGNANI, University of Trento, Italy

To ensure that secure applications do not leak their secrets, they are required to uphold several security properties such as spatial and temporal memory safety as well as cryptographic constant time. Existing work shows how to enforce these properties individually, in an architecture-independent way, by using secure compiler passes that each focus on an individual property. Unfortunately, given two secure compiler passes that each preserve a possibly different security property, it is unclear what kind of security property is preserved by the composition of those secure compiler passes. This paper is the first to study what security properties are preserved across the composition of different secure compiler passes. Starting from a general theory of property composition for security-relevant properties (such as the aforementioned ones), this paper formalises a theory of composition of secure compilers. Then, it showcases this theory a secure multi-pass compiler that preserves the aforementioned security-relevant properties. Crucially, this paper derives the security of the multi-pass compiler from the composition of the security properties preserved by its individual passes, which include security-preserving as well as optimisation passes. From an engineering perspective, this is the desirable approach to building secure compilers.

This paper uses syntax highlighting accessible to both colourblind and black & white readers.

CCS Concepts: • **Security and privacy → Formal security models**.

Additional Key Words and Phrases: Memory-safety, Secure Compilation, Privacy

ACM Reference Format:
Matthis Kruse, Michael Backes, and Marco Patrignani. 2024. Secure Composition of Robust and Optimising Compilers. In . ACM, New York, NY, USA, 28 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Memory Safety (MS) is a security property obtained by composing Spatial Memory Safety (SMS), which ensures array accesses are all within bounds, and Temporal Memory Safety (TMS), which ensures pointers are only used when they are valid [Akritidis et al. 2009; Azevedo de Amorim et al. 2018; Jim et al. 2002; Michael et al. 2023; Nagarakatte et al. 2009, 2010; Necula et al. 2005]. Cryptographic Constant Time (CCT) is a security property that ensures sensitive data is not leaked via timing side-channels [Kocher 1996]. Together, SMS, TMS and Strict Cryptographic Constant Time (sCCT), an enforceable overapproximation of CCT, yield Memory Safety and Strict Cryptographic Constant Time (MS+sCCT), which is the gold standard of security properties for secure applications. Programs attaining MS+sCCT do not leak sensitive data either through erroneous memory accesses, nor through timing side-channels. As discussed in Example 1.1, these security properties can be enforced by compiler passes [Almeida et al. 2017; Bond et al. 2017], to ensure programmers need not be aware of the architectural details of where their code will run.

POPL '24, January 17-19, 2024, London, UK

ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXX.XXXXXXX

Example 1.1 (strncpy). Consider the C function strncpy that copies a null-terminated string src into dst up to a length of n. This function is subject to a subtle SMS vulnerability: The bounds check i < n should happen *before* the access to memory location x[i]: otherwise the memory location past the last element will be leaked to an attacker.

```
void strncpy(size_t n, char *dst, char *src) {
  for(size_t i = 0; src[i] != '\0' && i < n; ++i) {
    dst[i] = src[i];
  }
}
```

To prevent this vulnerability, one can use a compilation pass that enforces SMS, such as Softbounds [Nagarakatte et al. 2009] or BaggyBounds [Akritidis et al. 2009].

Because of timing attacks, fixing SMS is not enough to make strncpy secure. In fact, the loop can terminate early, as soon as the string-terminating character '\0' is encountered, thus making program execution time proportional to the length of the array pointed by src. Also in this case there exist compiler passes that can rewrite such programs into CCT ones [Cauligi et al. 2019].

Alas, code is not run in isolation, so a malicious attacker could supply code that intracts with strncpy and trigger a violation of either MS or CCT by calling strncpy with an argument for src that points to uninitialised memory. This would, in turn, triggering a series of reads from uninitialised memory, which is an immediate MS violation with devastating real-world consequences [Microsoft 2010a,b,c, 2015; VMWare 2023].

Robust compilers [Abate et al. 2019] are a form of secure compilers that preserve security properties even in the presence of arbitrary attackers interacting with compiled code. Thus, robust compilers can be used to prevent vulnerabilities resulting from uninitialised memory (as well as many other ones), e.g., by targeting capability-based languages such as CHERI [Woodruff et al. 2014], Arm Morello [Arm 2022], or MSWasm [Michael et al. 2023], where the compiler relies on capabilities to check that pointers are always initialised.

Unfortunately, given secure compiler passes that each preserve a possibly different security property, there is no way to tell what kind of security property will the composition of those secure compilers preserve. Worse, without a framework for composing secure compiler passes, it is not possible to enable separation of concerns, e.g., to have a secure compilation pass that ensures MS that is developed independently of another secure pass for CCT, that is developed independently of other passes, such as optimisations.

This paper introduces a framework for reasoning about the composition of secure and optimising compiler passes akin to those of Example 1.1 and it showcases the power of this framework by instantiating it on a multi-pass compilation chain. To this end, this paper first discusses how to compose security properties, such as TMS and SMS into MS, and then adding sCCT to the mix to obtain MS+sCCT. Then, this paper defines compiler composition and formalises that given two passes that securely preserve two (possibly distinct) properties, their composition securely preserves the composition of those properties. The paper then defines several secure compiler passes, where each is either preserving a different security property (TMS, SMS, sCCT) or performing a security-preserving optimisation, (e.g., applying Constant Folding (CF) or Dead Code Elimination (DCE)). Finally, this paper shows that composing these secure compiler passes into a multi-pass compilation chain results in the end-to-end preservation of MS+sCCT. Crucially, this paper derives the security of the multi-pass compiler from the composition of the security properties preserved by its individual passes. This result showcases how the framework allows the kind of formal security reasoning that compiler writers already want (and already do), obtaining precise, compositional security reasoning while providing minimal (and modular) proof effort.

In summary, this paper makes the following contributions:

- This paper formalises security properties (Section 3) that are of interest for real-world compiler writers, namely TMS, SMS and CCT (as identified by the plethora of work enforcing such properties individually [Akritidis et al. 2009; Almeida et al. 2017; Bond et al. 2017; Cauligi et al. 2019; Dhumbumroong and Piromsopa 2020; Jung et al. 2021; Kuepper et al. 2023; Nagarakatte et al. 2009, 2010; Nam et al. 2019; Shankaranarayana et al. 2023; Younan et al. 2010; Zhou et al. 2023]). Starting from ways to formalise those properties individually, this paper shows how to compose their formalisation. The resulting security property is MS+sCCT, i.e., the gold standard of security properties for secure programs [LeMay et al. 2021].
- This paper takes the secure compilation framework of [Abate et al. 2019] and extends it to reason about the security of all different known forms of compiler composition (Section 4). For this, this paper studies sequential compiler composition as well as compilers with multiple input languages or multiple output ones, as used in existing compilation chains. This paper proves that starting from two compilers that preserve two (possibly distinct) properties, their composition preserves the intersection of those properties. Finally, this paper proves that the order of composition of sequential compiler passes is irrelevant for the resulting security. This is crucial for reordering optimisation passes and thus generating secure and efficient code.
- This paper presents a case-study showcasing the conjunction of the previous contributions (Sections 5 and 6). To this end, it presents a compilation chain consisting of several passes that ultimately preserves MS+sCCT by means of composing the individual, secure passes concerning TMS, SMS, and sCCT, respectively. Furthermore, the chain includes two optimisation passes: One performs DCE and the other CF. The formalisation of this case study showcases the power of the presented framework: The divide-and-conquer approach to software engineering is a viable strategy even for the development of secure compilers.
- The key contributions of this paper are formalised in the Coq proof assistant and the paper indicates this with .

This paper starts by introducing relevant notions of security properties and secure compilation (Section 2), and discusses related work (Section 7) before concluding (Section 8).

Open Source & Technical Report. A technical report with the omitted formal details, lemmas and proofs, as well as the Coq formalisation are available as supplementary material.

2 BACKGROUND: SECURITY PROPERTIES AND SECURE COMPILERS

To introduce the security argument of this paper, this section first presents the concepts of (security) properties, of their satisfaction, and of their robust satisfaction (i.e., satisfaction in the presence of an active attacker; Section 2.1). Then, borrowing from existing work [Abate et al. 2021a, 2019], the section introduces secure compilers as compilers that preserve robust property satisfaction (Section 2.2).

2.1 Properties and (Robust) Satisfaction

This paper employs the security model where programs are written in a language whose semantics emits events a. Events include security-relevant actions (e.g., reading from and writing to memory, as detailed in Section 3) and the unobservable event ε. As programs execute, their emitted events are concatenated in traces $\overline{a}$, which serve as the description of the behaviour of a program.[1]

[1] Throughout the paper, sequences are indicated with an overbar (i.e., $\overline{a}$), empty sequences with $[\cdot]$, and concatenation of sequences $\overline{a_1}, \overline{a_2}$ as $\overline{a_1} \cdot \overline{a_2}$. Prepending elements to sequences uses the same notation: $a \cdot \overline{a}$.

Properties π are sets of traces of admissible program behaviours, ascribing what said property considers valid. The set of all properties can be partitioned into different *classes* ($\mathbb{C}$), i.e., safety, liveness, and neither safety nor liveness [Clarkson and Schneider 2008]. A class is simply a set of properties and for the class of safety properties, it is decidable whether a trace satisfies a safety property with just a finite trace prefix. As an example, consider a trace describing an interaction with a memory where the deallocation of an address l precedes a read at that address in memory: Dealloc $l \cdot$ Read l 1729 $\cdot \ldots$. This program behaviour is insecure with respect to a canonical notion of (temporal) memory safety dictating no use-after-frees of pointers [Azevedo de Amorim et al. 2018; Nagarakatte et al. 2010], because it reads from a memory location that was freed already. The previous finite trace prefix is enough to decide that the trace does not satisfy TMS and there is no way to append events to this prefix which would result in the trace being admissible. In the following, the execution of a whole program w that terminates in state r according to the language semantics and produces trace $\overline{a}$ is written as $w \overset{\overline{a}}{\Longrightarrow} r$. With this, property satisfaction is defined as follows: whole programs w satisfy a property π iff w yields a trace $\overline{a}$ such that $\overline{a}$ satisifies π (Definition 2.1).

Definition 2.1 (Property Satisfaction). $\vdash p : \pi \overset{\text{def}}{=} \text{if } w \overset{\overline{a}}{\Longrightarrow} r, \text{ then } \overline{a} \in \pi.$

Property satisfaction is defined on whole programs, i.e., programs without missing definitions. Thus, from a security perspective, this considers only a passive attacker model, where the attacker observes the execution and, e.g., retrieves secrets from that. To consider a stronger model similarly to what existing work does [Abate et al. 2021a, 2019; Backes et al. 2014; Bengtson et al. 2011; Fournet et al. 2007; Gordon and Jeffrey 2003; Maffeis et al. 2008; Michael et al. 2023; Sammler et al. 2019; Swasey et al. 2017], the concept of satisfaction can be extended with *robustness*. Robust satisfaction considers partial programs p, i.e., components with missing imports, which cannot run until said imports are fulfilled. To remedy this, *linking* takes two partial programs p_1, p_2 and produces a whole program w, i.e., link $(p_1; p_2) = w$. As typically done in works that consider the execution of partial programs [Abate et al. 2019; Ahmed and Blume 2011; Bowman and Ahmed 2015; Devriese et al. 2017a,b; El-Korashy et al. 2021; Patrignani and Garg 2021; Patterson and Ahmed 2017; Van Strydonck et al. 2019], this paper assumes that whole programs are the result of linking partial programs referred to as *context* (ctx) and *component* (comp). The context is an arbitrary program and thus has the role of an *attacker* that can interact with the component by means of whatever features the programming language has, and the component is what is security-relevant. With this, Definition 2.1 (Property Satisfaction) can be extended as follows: for components p to robustly satisfy a property π, take an attacker context C and link it with p, the resulting whole program must satisfy π.

Definition 2.2 (Robust Satisfaction). $\vdash_R p : \pi \overset{\text{def}}{=} \forall C, \text{if link } (C; p) = w, \text{ then } \vdash w : \pi.$

Example 2.3 (Double Free in Bluetooth Subsystem). Consider `CVE-2021-3564` [BlockSec 2021], one of many submissions for a double-free vulnerability. The vulnerability arises due to a race condition where the context-level function `hci_cmd_work` was not expected to behave maliciously, since it resides in the same source-code repository where the vulnerability occurs. Nevertheless, the component-level code of `hci_dev_do_open` is linked with `hci_cmd_work` and does not atomically check whether a pointer has been freed already: Therefore, `hci_dev_do_open` does not satisfy the no-double-frees property robustly, since there is an implementation for `hci_cmd_work` that leads to a violation of that property when linked with `hci_dev_do_open`.

2.2 Secure Compilers

A *compiler* ($\gamma^{\mathrm{L}}_{\mathrm{L}}$) translates syntactic descriptions of programs from a *source* (L) into a *target* (L) programming language. This translation is considered *correct* if it is semantics-preserving. That is, for a whole program w, the compiler should relate the L semantics of w with the semantics of T of the compiled counterpart of p in such a way that they are „compatible". Unfortunately, correct compilers may be insecure compilers [Abadi 1999a; Ahmed et al. 2018; Kennedy 2006; Patrignani et al. 2019] and programs translated by insecure compilers can violate security properties that the programmer assumes to hold. To define when a compiler is secure, this paper uses the robust compilation framework [Abate et al. 2019], which the following definition summarises.

For compilers $\gamma^{\mathrm{L}}_{\mathrm{L}}$ to robustly preserve a class of properties $\mathbb{C}$, if for any property π of that class $\mathbb{C}$ and programs p written in L where p robustly satisfies π, then the compilation of p, $\gamma^{\mathrm{L}}_{\mathrm{L}}(\mathrm{p})$, must robustly satisfy π.

Definition 2.4 (Robust Compilation). $\vdash \gamma^{\mathrm{L}}_{\mathrm{L}} : \mathbb{C} \overset{\text{def}}{=} \forall (\pi \in \mathbb{C})\ (\mathrm{p} \in \mathrm{L_{tms}})$, if $\vdash_R \mathrm{p} : \pi$, then $\vdash_R \gamma^{\mathrm{L}}_{\mathrm{L}}(\mathrm{p}) : \pi$.

Note that a class of properties $\mathbb{C}$ can represent just one property π by lifting [Clarkson and Schneider 2008] that property to sets of properties, i.e., use the powerset of π instead of π itself. Because of this, this paper writes $\vdash \gamma^{\mathrm{L}}_{\mathrm{L}} : \pi$, even though π is a property and not a class.

Example 2.5 (Types). Suppose L is a statically-typed language similar to C and T is dynamically typed, where both share the same syntax up to dynamic type checks. Consider the following L component and its compiled version below.

```
fn foo (char ∗ x, int n) := ifz valid_ptr(x, n, sizeof(char)) then x[0] else −1
fn foo (        x,     n) := ifz valid_ptr(x, n, sizeof(char)) then x[0] else −1
```

While the compiler emits code that may look correct, the generated code does not check that the provided argument is of the right type. Even though the pointer **x** is checked for validity, the context **foo((int∗)y, 1)** is able to provoke a read out of bounds. Suppose the component transferred control to the context and passed ownership of a **char** pointer **y** sized **1** cells, the context can now call the component again, casting this buffer to an **int∗** prior to that call. The pointer is valid for one **char**-sized memory cell, as expected, but the actual read operation now returns **sizeof(int)** many bytes instead of just **sizeof(char)** many. Thus, even if foo may have been robust with respect to the SMS, its compiled counterpart is not and therefore the compiler fails to attain Definition 2.4.

3 SECURITY PROPERTIES: FORMALISATION, ENFORCEMENT AND COMPOSITION

This section introduces a trace model and uses it to define the key properties of interest for this paper: TMS, SMS, MS, and sCCT (Section 3.1). These properties are of practical importance (as mentioned in Section 1) and also of interest in the case study (Sections 5 and 6) this paper presents later. Lastly, for each of the key properties, this section introduces corresponding monitors (Section 3.2) that check them.

3.1 Specification Trace Model

$$(\textit{Security Tag})\ \sigma ::= 🔒 \mid 🔓 \qquad (\textit{Control Tag})\ t ::= \mathrm{ctx} \mid \mathrm{comp} \qquad (\textit{Event})\ a ::= \varepsilon \mid \lightning \mid a_b; t; \sigma$$

$$(\textit{Pre-event})\ a_b ::= \mathrm{Alloc}\ l\ \mathrm{n} \mid \mathrm{Dealloc}\ l \mid \mathrm{Use}\ l\ \mathrm{n} \mid \mathrm{Branch}\ \mathrm{n} \mid \mathrm{Binop}\ \mathrm{n}$$

The specification trace model defines events as either the empty event (ε), a crash ($\lightning$), or as tuples consisting of a pre-event, a control-tag, and a security-tag. The purpose of the model is to define key security properties of interest, such as MS or a stricter variant of cryptographic constant time. To this end, security-tags indicate whether an event contains sensitive information (🔒) or not

(🔓), while control-tags state whether the context (ctx) or the component (comp) are responsible for emitting the event. The latter is necessary to be able to ignore actions done by a spurious context that, e.g., immediately deallocates a memory location twice, thus violating TMS [Nagarakatte et al. 2010]. Lastly, pre-events describe the actual kind of event that happened. One such kind is the allocation event (Alloc l n) that fires whenever a program claims n cells of memory and stores them at address l. Dually, deallocation (Dealloc l) announces that the object at location l is freed. These two events alone are enough to provide a partial description of TMS by requiring that, e.g., there is only one deallocation event that carries a location l. To be able to express SMS, there is also an event to describe reads from and writes to memory (Use l n). Finally, for cryptographic code, there is a general guideline that secrets must not be visible on a trace. Moreover, an instruction whose timing is data-dependent must not have a secret as an operand. Typical operations with data-dependent timing are branches and certain binary operations, such as division[2]. Both operations are also modelled in the specification trace model (Branch n and Binop n).

3.1.1 Temporal Memory Safety. TMS is a safety property that describes that an unallocated object must not be used in any way. Moreover, the property requires that all allocated objects must be deallocated at some point.

Definition 3.1 (TMS).

$$\text{tms} := \left\{ \overline{a} \;\middle|\; \begin{array}{rcl} \text{Alloc } l \text{ n;t;}\sigma & \leq_{\overline{a}} & \text{Dealloc } l\text{;t;}\sigma\text{'} \\ \text{Use } l \text{ n;t;}\sigma & \leq_{\overline{a}} & \text{Dealloc } l\text{;t;}\sigma\text{'} \\ \text{if Alloc } l \text{ n;t;}\sigma \text{ in } \overline{a} & \text{then} & \text{Dealloc } l\text{;t;}\sigma\text{' in } \overline{a} \\ \text{at most one Dealloc } l\text{;t;}\sigma & \text{in} & \overline{a} \\ \text{at most one Alloc } l \text{ n;t;}\sigma & \text{in} & \overline{a} \end{array} \right\}$$

Hereby, the notation $a_1 \leq_{\overline{a}} a_2$ means that if a_1 is in $\overline{a}$ and if a_2 is in $\overline{a}$, then a_1 appears before a_2.

3.1.2 Spatial Memory Safety. SMS prohibits out-of-bounds accesses:

Definition 3.2 (SMS).

$$\text{sms} := \left\{ \overline{a} \;\middle|\; \text{If Alloc } l \text{ n;t;}\sigma \leq_{\overline{a}} \text{Use } l \text{ m;t;}\sigma\text{'},\ \text{then } m < n \right\}$$

3.1.3 Memory Safety. Full MS (similar to earlier work [Jim et al. 2002; Michael et al. 2023; Nagarakatte et al. 2009, 2010; Necula et al. 2005]) is then described as the conjunction of Definitions 3.1 and 3.2. Note, however, that this definition says nothing about memory-safety issues introduced by side-channels, such as speculation.

Definition 3.3 (MS).

$$\text{ms} := \text{tms} \cap \text{sms}$$

3.1.4 Strict Cryptographic Constant Time. CCT is a hypersafety property [Barthe et al. 2018] and, thus, difficult to check with monitors. This is because, intuitively, hypersafety properties can relate multiple execution traces with eachother, but monitors work on a single execution. To sidestep this issue, this section defines the property sCCT, a stricter variant of CCT that enforces the policy that no secret appears on a trace (inspired by earlier work [Almeida et al. 2017]).

Definition 3.4 (sCCT).

$$\text{scct} := \left\{ \overline{a} \;\middle|\; \overline{a} = [\cdot] \quad \text{or} \quad \overline{a} = a_b; t; \text{🔓} \cdot \overline{a'} \wedge \overline{a'} \in \text{scct} \right\}$$

[2]This is highly architecture-dependent, but division is an operation that serves as a classic example for a data-dependent timing instruction, e.g., [Arm 2020, p. 755].

3.1.5 Memory Safe, Strict Cryptographic Constant Time. The combination of MS and sCCT is the intersection of these properties, MS+sCCT. Since MS and sCCT are just sets of traces that, intuitively, contain all program behaviors that follow a security policy, the intersection of them contains all program behaviors that follow both security policies, i.e., it entails all program behaviours that are both MS and sCCT.

Definition 3.5 (MS and sCCT).

$$\mathrm{msscct} := \mathrm{ms} \cap \mathrm{scct}$$

3.2 Monitors

Monitors enforce safety properties by accepting or rejecting traces, i.e., if it rejects a trace, the trace does not satisfy the property the monitor checks. Since reasoning on monitors is easier than directly on just traces, this section presents a monitor for each of the previously shown safety properties (Section 3.1). To lessen the burden when proving that a monitor accepts the trace of a program execution, each monitor uses a custom trace model that contains only the relevant information related to the property the monitor checks. To go from specification traces $\overline{a}$ to monitor-level traces $\overline{\boldsymbol{a}}$, each property π has an associated event agreement relation $a \cong_\pi \boldsymbol{a}$. Figure 1 shows how the event agreement is lifted to traces. The trace agreement is the same for all properties π up to the

$\boxed{\overline{a} \cong^*_\pi \overline{\boldsymbol{a}}}$ „Specification-level trace $\overline{a}$ agrees with monitor-level trace $\overline{\boldsymbol{a}}$ with respect to property π."

$$\text{(traceagree-empty)}\ \frac{}{[\cdot] \cong^*_\pi [\cdot]} \qquad \text{(traceagree-ign-L)}\ \frac{\overline{a} \cong^*_\pi \overline{\boldsymbol{a}}}{\varepsilon \cdot \overline{a} \cong^*_\pi \overline{\boldsymbol{a}}} \qquad \text{(traceagree-ign-R)}\ \frac{\overline{a} \cong^*_\pi \overline{\boldsymbol{a}}}{\overline{a} \cong^*_\pi \boldsymbol{\varepsilon} \cdot \overline{\boldsymbol{a}}} \qquad \text{(traceagree-cons)}\ \frac{a \cong_\pi \boldsymbol{a} \qquad \overline{a} \cong^*_\pi \overline{\boldsymbol{a}}}{a \cdot \overline{a} \cong^*_\pi \boldsymbol{a} \cdot \overline{\boldsymbol{a}}}$$

Fig. 1. Trace-Agreement relation that equates specification-level traces with monitor-level traces.

use of the event agreement in Rule traceagree-cons. With agreements, this section defines monitor satisfaction for traces and then it proves that monitor satisfaction implies property satisfaction. To this end, monitor satisfaction is defined as follows. A specification trace $\overline{a}$ monitor-satisfies property π iff there exists a (final) monitor state T and an abstract trace $\overline{\boldsymbol{a}}$ such that the specification trace $\overline{a}$ agrees with abstract trace $\overline{\boldsymbol{a}}$ and the initial monitor[3] can step to the (final) monitor state T with abstract trace $\overline{\boldsymbol{a}}$.

Definition 3.6 (Monitor Satisfaction). $\vdash_{mon} \overline{a} : \pi \stackrel{\text{def}}{=} \exists \overline{\boldsymbol{a}}\ T, \overline{a} \cong^*_\pi \overline{\boldsymbol{a}}$ and $\vdash \emptyset \xrightarrow{\overline{\boldsymbol{a}}}{}^* T$.

3.2.1 Monitor for TMS.

$$(\textit{Abstract Store})\ T_{TMS} ::= \{\text{allocated} : L \times t, \text{freed} : L \times t\} \quad \emptyset := \{\text{allocated} : \emptyset, \text{freed} : \emptyset\}$$

$$(\textit{Abstract Events})\ \boldsymbol{a} ::= \boldsymbol{\varepsilon} \mid \boldsymbol{Alloc}\ l\ t \mid \boldsymbol{Dealloc}\ l\ t \mid \boldsymbol{Use}\ l\ t \mid \lightning$$

$\boxed{\vdash T_{TMS} \xrightarrow{\boldsymbol{a}} T_{TMS}'}$ „Monitor T_{TMS} does one step to T_{TMS}' given event $\boldsymbol{a}$."

[3]In this paper, for all monitors, the initial monitor state is denoted as $\emptyset$.

(tms-use)
$$\frac{(l;t) \in T_{TMS}.\text{allocated} \qquad (l;t) \notin T_{TMS}.\text{freed}}{\vdash T_{TMS} \overset{\textbf{\textit{Use } } l\ t}{\rightsquigarrow} T_{TMS}}$$

(tms-alloc)
$$\frac{(l;t) \notin T_{TMS}.\text{allocated} \qquad (l;t) \notin T_{TMS}.\text{freed} \qquad T_{TMS}' = \{\text{allocated} : T_{TMS}.\text{allocated} \cup \{(l;t)\}, \text{freed} : T_{TMS}.\text{freed}\}}{\vdash T_{TMS} \overset{\textbf{\textit{Alloc} } l\ t}{\rightsquigarrow} T_{TMS}'}$$

(tms-dealloc)
$$\frac{(l;t) \in T_{TMS}.\text{allocated} \qquad (l;t) \notin T_{TMS}.\text{freed} \qquad T_{TMS}' = \{\text{allocated} : T_{TMS}.\text{allocated} \setminus \{(l;t)\}, \text{freed} : T_{TMS}.\text{freed} \cup \{(l;t)\}\}}{\vdash T_{TMS} \overset{\textbf{\textit{Dealloc} } l\ t}{\rightsquigarrow} T_{TMS}'}$$

For TMS, the state of the monitor is a record with two sets keeping track of allocated and deallocated locations. Rule tms-use simply requires that a location is (i) allocated and (ii) not freed. Rules tms-alloc and tms-dealloc both require a location to not be freed already and extend the monitor state accordingly. This restriction effectively disallows reallocation to reassign the same location to an object. However, the definition can easily be adapted by, e.g., attaching a natural number serving as a counter. Contrary to other monitors in this paper, the multi-step relation of the TMS monitor is non-standard:

$\boxed{\vdash T_{TMS} \overset{\overline{\boldsymbol{a}}}{\rightsquigarrow}{}^{*} T_{TMS}'}$ „Monitor T_{TMS} multi-steps to T_{TMS}' given trace $\overline{\boldsymbol{a}}$."

(tms-refl)
$$\frac{T_{TMS}.\text{allocated} = \emptyset}{\vdash T_{TMS} \overset{[\cdot]}{\rightsquigarrow}{}^{*} T_{TMS}}$$

(tms-ign-trans)
$$\frac{\vdash T_{TMS} \overset{\overline{\boldsymbol{a}}}{\rightsquigarrow}{}^{*} T_{TMS}'}{\vdash T_{TMS} \overset{\boldsymbol{\varepsilon}\cdot\overline{\boldsymbol{a}}}{\rightsquigarrow} T_{TMS}'}$$

(tms-trans)
$$\frac{\vdash T_{TMS} \overset{\boldsymbol{a}}{\rightsquigarrow} T_{TMS}' \quad \vdash T_{TMS}' \overset{\overline{\boldsymbol{a}}}{\rightsquigarrow}{}^{*} T_{TMS}''}{\vdash T_{TMS} \overset{\boldsymbol{a}\cdot\overline{\boldsymbol{a}}}{\rightsquigarrow} T_{TMS}''}$$

Rules tms-ign-trans and tms-trans are the same for all monitors, but Rule tms-refl has, in this case, an additional premise that no more locations should be allocated. This rejects the behavior of programs that forget to free memory.

$\boxed{a \cong_{\text{tms}} \boldsymbol{a}}$ „Abstract event $\boldsymbol{a}$ is equivalent to a with respect to TMS."

(tms-alloc-authentic)
$$\frac{}{\text{Alloc } l\ n; t; \sigma \cong_{\text{tms}} \textbf{\textit{Alloc}}\ l\ t}$$

(tms-branch-authentic)
$$\frac{}{\text{Branch } n \cong_{\text{tms}} \boldsymbol{\varepsilon}}$$

(tms-abort-authentic)
$$\frac{}{\lightning \cong_{\text{tms}} \boldsymbol{\lightning}}$$

The trace agreement is entirely straightforward, so only allocation, branch, and crash are shown.

Lemma 3.7 (Traces with Monitor Satisfaction are tms). *If* $\vdash_{mon} \overline{a} : \text{tms}$, *then* $\overline{a} \in \text{tms}$.

Example 3.8 (A program not satisfying TMS). Consider the following C++11 library that calls strncpy (Example 1.1) and prints the result to the standard output stream.

```
int greet() {  // allocates 12 chars containing a greeting message
  char* greetings = new char[12] { "Hello␣POPL!" }; // <- address l_x
  char* to = new char[12];                          // <- address l_y

  strncpy(12, to, greetings);
  delete to;

  printf("%cOPL\n", to[6]);
}
```

Up to the body of `printf`, the program execution yields the specification trace Alloc l_x 12; comp; 🔓 · Alloc l_y 12; comp; 🔓 · Use l_x 0; comp; 🔓 · Use l_x 0; comp; 🔓 · Use l_y 0; comp; 🔓 · Use l_x 1; comp; 🔓 · . . . · Use l_x 12; comp; 🔓 · Dealloc l_y; comp; 🔓 · Use l_y 6; comp; 🔓. Relating this trace to abstract monitor events yields $\overline{\boldsymbol{a}} = \boldsymbol{Alloc}\ l_x\ \text{comp} \cdot \boldsymbol{Alloc}\ l_y\ \text{comp} \cdot \boldsymbol{Use}\ l_x\ \text{comp} \cdot \boldsymbol{Use}\ l_x\ \text{comp} \cdot \boldsymbol{Use}\ l_y\ \text{comp} \cdot \ldots \cdot \boldsymbol{Use}\ l_x\ \text{comp} \cdot \boldsymbol{Dealloc}\ l_y\ \text{comp} \cdot \boldsymbol{Use}\ l_y\ \text{comp}$. Remembering the definition of `strncpy` (Example 1.1), observe that it does not deallocate its arguments. Even though the trace contains an out-of-bounds access right before returning from `strncpy`, this is no concern for TMS, since the location l_x is still allocated. However, having returned from `strncpy`, the `greet` function continues and deallocates l_y whose subsequent use in the `printf` call is a use-after-free bug.

The fix would be to **`delete`** `greetings` instead of `to` and add a **`delete`** `to` after the `printf` call, which leads to the abstract monitor trace $\overline{\boldsymbol{a}}' = \boldsymbol{Alloc}\ l_x\ \text{comp} \cdot \boldsymbol{Alloc}\ l_y\ \text{comp} \cdot \boldsymbol{Use}\ l_x\ \text{comp} \cdot \boldsymbol{Use}\ l_x\ \text{comp} \cdot \boldsymbol{Use}\ l_y\ \text{comp} \cdot \ldots \cdot \boldsymbol{Use}\ l_x\ \text{comp} \cdot \boldsymbol{Dealloc}\ l_x\ \text{comp} \cdot \boldsymbol{Use}\ l_y\ \text{comp}$. It follows that $\vdash_{mon} \overline{\boldsymbol{a}}' : \text{tms}$ and from Lemma 3.7 (Traces with Monitor Satisfaction are tms), it follows that the program satisfies Definition 3.1 (TMS), even though the program still violates SMS.

3.2.2 *Monitor for SMS.*

$$(\textit{Abstract Store})\ T_{SMS} ::= L \times t \times \mathbb{N} \qquad (\textit{Abstract Events})\ \boldsymbol{a} ::= \boldsymbol{\varepsilon} \mid \boldsymbol{Alloc}\ l\ t\ n \mid \boldsymbol{Use}\ l\ t\ n$$

$\boxed{\vdash T_{SMS} \overset{\boldsymbol{a}}{\rightsquigarrow} T_{SMS}'}$ „Monitor T_{SMS} does one step to T_{SMS}' given event $\boldsymbol{a}$."

$$\frac{(l; t; m) \in T_{SMS} \qquad n < m}{\vdash T_{SMS} \xrightarrow{\boldsymbol{Use}\ l\ t\ n} T_{SMS}}\ (\text{sms-use}) \qquad \frac{(l; t; m) \notin T_{SMS}}{\vdash T_{SMS} \xrightarrow{\boldsymbol{Alloc}\ l\ t\ n} T_{SMS} \cup \{(l; t; n)\}}\ (\text{sms-alloc})$$

The state of the monitor for SMS is a set containing tuples of locations, control-tags, and the allocation size. In comparison to the trace model of the TMS monitor, the trace model here is extended by sizing and positional information. Rule sms-use performs a bounds check and Rule sms-alloc adds bounds information to the state of the monitor. The trace agreement is entirely straightforward and similar to the one for TMS.

Lemma 3.9 (Traces with Monitor Satisfaction are sms). *If* $\vdash_{mon} \overline{a} : \text{sms}$, *then* $\overline{a} \in \text{sms}$.

Example 3.10 (Normal invocation of `strncpy`*).* Consider the insecure `strncpy` function from Example 1.1 with a context `strncpy(2, x, y)`, where `x` and `y` are pointers to valid regions of memory with allocated space for exactly two cells and do not contain the null-terminating character `'\0'`. For the sake of this example, the pointers have been allocated by the component and passed to the context. The loop of `strncpy` will copy exactly two cells and then check the loop condition for the last time. At that stage, the induction variable `i` is equal to 2 and, unfortunately, the order of checks is such that first the cell `x[i]` is read prior to bounds checking `i < n`. Because of this, there is an out-of-bounds memory access right before exiting the function. This is also visible on the trace, which can be sketched as . . . · Alloc l_x 2; comp; 🔓 · . . . · Alloc l_y 2; comp; 🔓 · . . . · Use l_x 0; comp; 🔓 · Use l_y 0; comp; 🔓 · Use l_x 1; comp; 🔓 · Use l_y 1; comp; 🔓 · Use l_x 2; comp; 🔓 · . . ., where l_x and l_y are the memory addresses associated to `x` and `y`, respectively. Omitting the events for all „. . ." for sake of brevity, the abstract monitor trace of this is $\boldsymbol{Alloc}\ l_x\ \text{comp}\ 2 \cdot \boldsymbol{Alloc}\ l_y\ \text{comp}\ 2 \cdot \boldsymbol{Use}\ l_x\ \text{comp}\ 0 \cdot \boldsymbol{Use}\ l_y\ \text{comp}\ 0 \cdot \boldsymbol{Use}\ l_x\ \text{comp}\ 1 \cdot \boldsymbol{Use}\ l_y\ \text{comp}\ 1 \cdot \boldsymbol{Use}\ l_x\ \text{comp}\ 2$.

After the allocation events, the state of the monitor is $\{(l_x; \text{comp}; 2), (l_y; \text{comp}; 2)\}$. All uses up to the last are accepted by the monitor, but the last event does not satisfy the premise $2 < 2$ in Rule sms-use. Therefore, the whole program (`strncpy` linked with this kind of context) is not SMS.

3.2.3 *Combining TMS and SMS Monitors to obtain MS.*

$\boxed{\vdash T_{MS} \overset{\boldsymbol{a}}{\rightsquigarrow} T_{MS}'}$ „Monitor T_{MS} does one step to T_{MS}' given event $\boldsymbol{a}$."

$$\frac{\vdash T_{TMS} \overset{\boldsymbol{a}_{\text{tms}}}{\rightsquigarrow} T_{TMS}' \qquad \vdash T_{SMS} \overset{\boldsymbol{a}_{\text{sms}}}{\rightsquigarrow} T_{SMS}'}{\vdash (T_{TMS}, T_{SMS}) \overset{(\boldsymbol{a}_{\text{tms}}, \boldsymbol{a}_{\text{sms}})}{\rightsquigarrow} (T_{TMS}', T_{SMS}')}\ \text{(ms-step)}$$

The combined monitor runs the one for TMS and the one for SMS in a lockstep. The trace agreement similarly just relates a specification event with an abstract TMS-event $\overline{\boldsymbol{a}}_{\text{tms}}$ and with an abstract SMS-event $\overline{\boldsymbol{a}}_{\text{sms}}$.

Lemma 3.11 (Traces with Monitor Satisfaction are ms). *If* $\vdash_{mon} \overline{a} : \text{ms}$*, then* $\overline{a} \in \text{ms}$.

3.2.4 *Monitor for sCCT.*

$$(\textit{Abstract Store})\ T_{sCCT} ::= \emptyset \qquad (\textit{Abstract Events})\ \boldsymbol{a} ::= \boldsymbol{\varepsilon} \mid \boldsymbol{\lightning} \mid \boldsymbol{Any}$$

$\boxed{\vdash T_{sCCT} \overset{\boldsymbol{a}}{\rightsquigarrow} T_{sCCT}'}$ „Monitor T_{sCCT} does one step to T_{sCCT}' given event $\boldsymbol{a}$."

$$\frac{}{\vdash T_{sCCT} \overset{\boldsymbol{\varepsilon}}{\rightsquigarrow} T_{sCCT}}\ \text{(scct-none)} \qquad \frac{}{\vdash T_{sCCT} \overset{\boldsymbol{\lightning}}{\rightsquigarrow} T_{sCCT}}\ \text{(scct-abort)}$$

The monitor state for the sCCT monitor is completely empty, since it does not need to keep track of information. As soon as any event is hit, the execution gets stuck, since any event is considered confidental from the perspective of this monitor.

$\boxed{a \cong_{\text{scct}} \boldsymbol{a}}$ „Abstract event $\boldsymbol{a}$ is equivalent to a with respect to CCT."

$$\frac{}{a_b; t; 🔓 \cong_{\text{scct}} \boldsymbol{\varepsilon}}\ \text{(scct-low-authentic)} \qquad \frac{}{a_b; t; 🔒 \cong_{\text{scct}} \boldsymbol{Any}}\ \text{(scct-high-authentic)}$$

$$\frac{}{\varepsilon \cong_{\text{scct}} \boldsymbol{\varepsilon}}\ \text{(scct-empty-authentic)} \qquad \frac{}{\lightning \cong_{\text{scct}} \boldsymbol{\lightning}}\ \text{(scct-abort-authentic)}$$

Accordingly, the event agreement simply disregards all events that involved public data (🔓) while mapping any other event that does involve private data (🔒) to the abstract ***Any*** event.

Lemma 3.12 (Monitor Traces are sCCT). *If* $\vdash_{mon} \overline{a} : \text{scct}$*, then* $\overline{a} \in \text{scct}$.

Example 3.13 (Data-independent timing mode). Consider the call `strncpy(1,x,y)` to the `strncpy` function (Example 1.1) with low (🔓) security for `x` and high security (🔒) for `y`. The trace of just the copying part inside of `strncpy` looks like Use l_x 0; comp; 🔓 · Use l_y 0; comp; 🔒. In terms of the abstract monitor trace, this is just ***Any***. Running this on the monitor would result in getting stuck, since there is no matching rule to step in the presence of ***Any*** event. By means of additional features to ensure cryptographic constant time even in the presence of memory reads and loads, such as a flag to enable a data independent timing mode, which is present in both Arm [Arm 2020, p. 543] and Intel [Intel 2023, p. 80] processors, the original trace now does Use l_y 0; comp; 🔓 instead of Use l_y 0; comp; 🔒. Because of this, the whole trace of the component equates to $\boldsymbol{\varepsilon}$ and the monitor can step without getting stuck.

3.2.5 *Combining MS and sCCT Monitors to obtain MS+sCCT.*

The combination of monitors for MS and sCCT yields one for MS+sCCT. The construction is entirely similar to the one for MS (Section 3.2.3).

Lemma 3.14 (Traces with Monitor Satisfaction are msscct). *If* $\vdash_{mon} \overline{a} : \text{msscct}$*, then* $\overline{a} \in \text{msscct}$.

4 COMPOSING SECURE COMPILERS

This section presents the key meta-theoretic results of this paper concerning sequential compiler composition (and of optimisation passes) (Section 4.1) and concerning other kinds of compiler composition (Section 4.2).

4.1 Secure Sequential Composition

The main result is that secure compilers in the robust compilation framework [Abate et al. 2019] compose *sequentially*. This is not intuitive in the sense that in the security domain, composition does not work without additional generalizations [Canetti et al. 2006; Fabian et al. 2022; McCullough 1988]. The sequential composition of compilers $\gamma_{\mathrm{L}}^{\mathrm{L}}$ and $\gamma_{\mathrm{L}}^{\mathrm{L}}$ is defined as follows: Given an L program p and compilers $\gamma_{\mathrm{L}}^{\mathrm{L}}, \gamma_{\mathrm{L}}^{\mathrm{L}}$, its compiled L counterpart is obtained by plugging p into $\gamma_{\mathrm{L}}^{\mathrm{L}} \circ \gamma_{\mathrm{L}}^{\mathrm{L}}$.

Definition 4.1. $\gamma_{\mathrm{L}}^{\mathrm{L}} \circ \gamma_{\mathrm{L}}^{\mathrm{L}} \overset{\text{def}}{=}$ Given p, yield $\gamma_{\mathrm{L}}^{\mathrm{L}}\left(\gamma_{\mathrm{L}}^{\mathrm{L}}(\mathrm{p})\right)$

Consider the compilation chain for TypeScript. First, TypeScript programs are translated to JavaScript which, e.g., V8 [Google 2008] eventually compiles in parts to *IgnitionBC*. The following theorem establishes what happens if all these compilation steps were robustly secure with respect to MS: The resulting *IgnitionBC* code would be MS regardless of the context the binary runs in.

Given $\gamma_{\mathrm{L}}^{\mathrm{L}}$ robustly preserves $\mathbb{C}_1$ and $\gamma_{\mathrm{L}}^{\mathrm{L}}$ robustly preserves $\mathbb{C}_2$, it follows that their sequential composition $\gamma_{\mathrm{L}}^{\mathrm{L}} \circ \gamma_{\mathrm{L}}^{\mathrm{L}}$ robustly preserves the intersection of classes $\mathbb{C}_1$ and $\mathbb{C}_2$.

Theorem 4.2 (Sequential Composition of Secure Compilers). *If* $\vdash \gamma_{\mathrm{L}}^{\mathrm{L}} : \mathbb{C}_1$ *and* $\vdash \gamma_{\mathrm{L}}^{\mathrm{L}} : \mathbb{C}_2$, *then* $\vdash \gamma_{\mathrm{L}}^{\mathrm{L}} \circ \gamma_{\mathrm{L}}^{\mathrm{L}} : \mathbb{C}_1 \cap \mathbb{C}_2$.

Since the composition of secure compilers is again a secure compiler, the theorem generalises to a whole chain of n secure compilers.

4.1.1 Securing Optimisations. Notably, real-world compilation chains also perform a series of (sequential) passes whose main purpose is not necessarily to translate from one language to another, but to, e.g., optimise the code or enforce a certain property. Both examples can be seen in practice, e.g. as in the work of [Akritidis et al. 2009; Manjikian and Abdelrahman 1997; Nagarakatte et al. 2009, 2010; Wegman and Zadeck 1991] and many more. Consider the following two LLVM optimisation passes: CF, which rewrites constant expressions to the constant they evaluate to, and DCE, which removes dead code by rewriting conditional branches. The order in which CF and DCE are performed influences the final result of the compilation (see Figure 2). This *phase ordering*

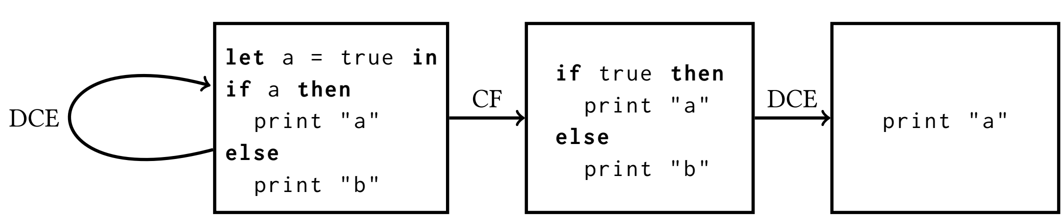

Fig. 2. Example program where the level of optimisations differ for one pass of applying CF and DCE in any order. Every edge is a compilation pass and the label on the edge states what the pass does, i.e., CF or DCE. The source code in the nodes is a glorified compiler intermediate representation and the code gets more optimised towards the right hand side of the figure.

problem is well–known in literature and a practical solution is to simply perform a fixpoint iteration of the optimisation pipeline [Click and Cooper 1995]. Compiler engineers typically try to find an order of optimisations that yields well-optimised programs for either code size [Cooper et al. 1999] or performance [Kulkarni et al. 2006]. Corollary 4.3 justifies that any such order of compilation passes is valid with respect to security. So, given two compilation passes $\gamma_1{}^{\mathsf{L}}_{\mathsf{L}}, \gamma_2{}^{\mathsf{L}}_{\mathsf{L}}$, both robustly preserving class $\mathbb{C}_1$ or $\mathbb{C}_2$, respectively, for any order of their composition the composed compiler robustly preserves the intersection of $\mathbb{C}_1$ and $\mathbb{C}_2$.

Corollary 4.3 (Sequential Composition of Secure Compilers). *If* $\vdash \gamma_1{}^{\mathsf{L}}_{\mathsf{L}} : \mathbb{C}_1$ *and* $\vdash \gamma_2{}^{\mathsf{L}}_{\mathsf{L}} : \mathbb{C}_2$, *then* $\vdash \gamma_1{}^{\mathsf{L}}_{\mathsf{L}} \circ \gamma_2{}^{\mathsf{L}}_{\mathsf{L}} : \mathbb{C}_1 \cap \mathbb{C}_2$ *and* $\vdash \gamma_2{}^{\mathsf{L}}_{\mathsf{L}} \circ \gamma_1{}^{\mathsf{L}}_{\mathsf{L}} : \mathbb{C}_2 \cap \mathbb{C}_1$.

4.2 Secure Upper and Lower Composition

Besides sequential composition, there are two other compositions, namely an *upper*, i.e., a compiler that takes multiple inputs and yields one output, and a *lower* composition, i.e., a compiler that takes one input and yields multiple outputs. Define the upper composition $\gamma^{\mathsf{L}+\mathsf{L}}_{\mathsf{L}}$ as follows: Given a program p, its compiled counterpart is obtained by plugging p into $\gamma^{\mathsf{L}}_{\mathsf{L}}$ if $\mathsf{p} \in \mathsf{L}$ or by plugging p into $\gamma^{\mathsf{L}}_{\mathsf{L}}$ if $\mathsf{p} \in \mathsf{L}$.

Definition 4.4 (Upper Composition). $\gamma^{\mathsf{L}+\mathsf{L}}_{\mathsf{L}} \stackrel{\text{def}}{=} \lambda \mathsf{p}. \begin{cases} \text{if } \mathsf{p} \in \mathsf{L}, \text{ then } \gamma^{\mathsf{L}}_{\mathsf{L}}(\mathsf{p}) \\ \text{if } \mathsf{p} \in \mathsf{L}, \text{ then } \gamma^{\mathsf{L}}_{\mathsf{L}}(\mathsf{p}) \end{cases}$

Examples of this are present in industry: Consider the Java Virtual Machine bytecode JVMBC, which is a popular target for programming language designers due to its high performance and relevance in industry. Compilers for several programming languages have it as their target language, some popular instances are Java and `Kotlin`. Technically speaking, they both compile to class files and `Kotlin` objects are considered to be the same as Java objects at that point. Both languages can be used at the same time in one project [Google [n. d.]]. A compiler that accepts both Java and `Kotlin` code translating to the same target language or intermediate representation performs a kind of *upper* composition. Now, the following theorem tells us what happens if these are secure: Given $\gamma^{\mathsf{L}}_{\mathsf{L}}$ robustly preserves $\mathbb{C}_1$ and $\gamma^{\mathsf{L}}_{\mathsf{L}}$ robustly preserves $\mathbb{C}_2$, it follows that their upper composition $\gamma^{\mathsf{L}+\mathsf{L}}_{\mathsf{L}}$ robustly preserves the intersection of classes $\mathbb{C}_1$ and $\mathbb{C}_2$.

Theorem 4.5 (Upper Composition of Secure Compilers). *If* $\vdash \gamma^{\mathsf{L}}_{\mathsf{L}} : \mathbb{C}_1$ *and* $\vdash \gamma^{\mathsf{L}}_{\mathsf{L}} : \mathbb{C}_2$, *then* $\vdash \gamma^{\mathsf{L}+\mathsf{L}}_{\mathsf{L}} : \mathbb{C}_1 \cap \mathbb{C}_2$.

Dually, the *lower* composition is concerned about compilers that accept the same source but yield different target languages. Define the lower composition $\gamma^{\mathsf{L}}_{\mathsf{L}+\mathsf{L}}$ as follows: Given a program p, its compiled counterpart is obtained by plugging p into $\gamma^{\mathsf{L}}_{\mathsf{L}}$ or by plugging p into $\gamma^{\mathsf{L}}_{\mathsf{L}}$, respectively, based on the internal decision.

Definition 4.6 (Lower Composition). $\gamma^{\mathsf{L}}_{\mathsf{L}+\mathsf{L}} \stackrel{\text{def}}{=} \lambda \mathsf{p}, L. \begin{cases} \text{if } L = \mathsf{L}, \text{ then } \gamma^{\mathsf{L}}_{\mathsf{L}}(\mathsf{p}) \\ \text{if } L = \mathsf{L}, \text{ then } \gamma^{\mathsf{L}}_{\mathsf{L}}(\mathsf{p}) \end{cases}$

Consider two compilers both accepting LLVMIR [Lattner and Adve 2004] and one of them emits x86_64, while the other emits `ARMv8`. It is intuitive that they are in some sense composed in the LLVM framework, but the decision of when to use one over the other is inherently *internal* to the formalisation effort of this kind of composition. For example, the user of this compiler provides an explicit flag that instructs to emit x86_64 or the framework itself detects the target platform via heuristics, such as supported instructions.

The following theorem demonstrates what happens if the involved compilers are secure: Given $\gamma^{\mathsf{L}}_{\mathsf{L}}$ robustly preserves $\mathbb{C}_1$ and $\gamma^{\mathsf{L}}_{\mathsf{L}}$ robustly preserves $\mathbb{C}_2$, it follows that their lower composition $\gamma^{\mathsf{L}}_{\mathsf{L+L}}$ robustly preserves the intersection of classes $\mathbb{C}_1$ and $\mathbb{C}_2$.

Theorem 4.7 (Lower Composition of Secure Compilers). *If* $\vdash \gamma^{\mathsf{L}}_{\mathsf{L}} : \mathbb{C}_1$ *and* $\vdash \gamma^{\mathsf{L}}_{\mathsf{L}} : \mathbb{C}_2$*, then* $\vdash \gamma^{\mathsf{L}}_{\mathsf{L+L}} : \mathbb{C}_1 \cap \mathbb{C}_2$.

5 CASE STUDY: LANGUAGE FORMALISATIONS

This section defines programming languages that the secure compilers defined in the next section will use. To this end, this section defines the languages $\mathsf{L}_{\mathsf{tms}}$, L, L_{ms}, and $\mathsf{L}_{\mathsf{scct}}$ which share many common elements (presented in Section 5.1). $\mathsf{L}_{\mathsf{tms}}$ is the only statically typed language and exhibits the property that all well-typed programs are TMS (Section 5.2). However, not all $\mathsf{L}_{\mathsf{tms}}$ programs are SMS. That is, there are well-typed $\mathsf{L}_{\mathsf{tms}}$ programs that perform an out-of-bounds access. Language L is untyped and does not provide any guarantees with regards to MS (Section 5.3). L_{ms} is exactly the same language as L, but this paper still distinguishes the two for sake of readability (Section 5.4). All three languages — so $\mathsf{L}_{\mathsf{tms}}$, L, and L_{ms} — assume CCT to hold.

Writing code attaining CCT should not be of the programmer's concerns [Cauligi et al. 2019]. Such consideration is also backed up by architecture providing a data (operand) independent timing mode, such as processors by Arm [Arm 2020, p. 543] and Intel [Intel 2023, p. 80]. In spirit of this, language $\mathsf{L}_{\mathsf{scct}}$ allows violating CCT by emitting events on, e.g., branching and division, that contain secrets (Section 5.5), but provides a way to read and write to a *model–specific register* that enables a "CCT-mode".

5.1 Shared Language Definitions

$$
\begin{aligned}
&(\mathit{Expressions})\ e ::= x \mid v \mid e_1{\oplus}e_2 \mid x[e] \mid \mathit{let}\ x{=}\mathit{new}\ e_1\ [e_2]\ \mathit{in}\ e_3 \mid \mathit{delete}\ x \mid x[e_1] \leftarrow e_2\\
&\qquad \mid \langle e_1;e_2\rangle \mid e.0 \mid e.1 \mid \mathit{let}\ x{=}e_1\ \mathit{in}\ e_2 \mid \mathit{return}\ e \mid \mathit{call}\ g\ e \mid \mathit{ifz}\ e\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid \mathit{abort}()\\
&(\mathit{Types})\ \tau ::= \mathbb{N}_t \mid \tau_1 \times \tau_2 \quad (\mathit{Functions})\ F ::= \mathit{fn}\ \mathit{foo}\ x := e \quad (\mathit{Libraries})\ \Xi ::= [\cdot] \mid F, \Xi\\
&(\mathit{Component\ Names})\ \xi ::= [\cdot] \mid \mathrm{foo}, \xi \quad (\mathit{Programs})\ \langle\, \Xi_{\mathrm{ctx}}; \Xi_{\mathrm{comp}} \rangle
\end{aligned}
$$

Above is the shared syntax of all the programming languages of this paper. Variables are referred to as $x, y, z, a, b, c, \ldots$ while functions may be referred to as multi-character words, such as foo, as well as short-forms like f, g, h. All languages share the type $\mathbb{N}_t$ representing natural numbers. Functions are constrained to take one argument and can only call other functions listed in libraries, which are just lists of functions. A program $\langle\, \Xi_{\mathrm{ctx}}; \Xi_{\mathrm{comp}} \rangle$ is indexed by two libraries that represent all context- and component-level functions, respectively. Lists of component-level names are referred to as ξ.

$$
\begin{aligned}
&(\mathit{Control\ Tags})\ t ::= \mathrm{ctx} \mid \mathrm{comp} \quad (\mathit{Communication\ Tags})\ c ::= ? \mid ! \mid \varnothing \quad (\mathit{Poison\ Tags})\ \rho ::= ☣ \mid \square\\
&(\mathit{Continuation\ Stacks})\ \overline{K} ::= [\cdot] \mid (K; g), \overline{K} \qquad (\mathit{Control\ Flow\ States})\ \Psi ::= (\Xi; \overline{K})\\
&(\mathit{Locations})\ l \in \mathbb{N} \qquad (\mathit{Substitutions})\ \gamma ::= [\cdot] \mid [v \text{ for } x], \gamma\\
&(\mathit{Stores})\ \Delta ::= [\cdot] \mid x \mapsto (l; t; \rho; n), \Delta \qquad (\mathit{Heaps})\ H ::= [\cdot] \mid v, H\\
&(\mathit{Memory\ States})\ \Phi ::= (H^{\mathrm{ctx}}; H^{\mathrm{comp}}; \Delta) \quad (\mathit{States})\ \Omega ::= (\Psi; t; \Phi) \quad (\mathit{Runtime\ Terms})\ r ::= \Omega \triangleright e
\end{aligned}
$$

States Ω are tuples[4] containing a control flow state Ψ, control tags t, and a memory state Φ. A control flow state Ψ entails a library, which provides definitions for functions calls, as well as a stack of continuations $\overline{K}$. Elements of the stack $\overline{K}$ are pairs of evaluation contexts and the name of a function associated to that evaluation context. Memory states Φ are tuples of two separate heaps H and a store Δ, which contains pointer metadata, such as the concrete memory location l, a control tag t indicating which heap the pointer points into, a poison tag ρ, as well as bounds information n. The bounds information is a mere proof artefact that has no semantic significance. The two separate heaps essentially model a sandbox, to prevent contexts from performing pointer arithmetic and reading from or writing to the data owned by a component. While this prevents the effects of out-of-bounds accesses across the context and component boundary, the goal of the design of the languages of this paper is to be able to express security violations. To this end, the poison tag ρ indicates whether a pointer has been freed ☣ or is still allocated □, so that pointers can be used even after their deallocation without the semantics getting stuck. Runtime terms are simply expressions e paired with the operational state Ω.

$$\begin{aligned}(\textit{Pre-Events})\ a_b &::= \text{Alloc}\ l\ v \mid \text{Dealloc}\ l \mid \text{Get}\ l\ v \mid \text{Set}\ l\ v\ v' \mid \cdots\\ (\textit{Events})\ a &::= \varepsilon \mid \lightning \mid (a_b; t; \sigma)\end{aligned}$$

All languages use the same trace model, where events are either the empty event ε, the program crash event $\lightning$, or a tuple consisting of a control tag t and a security tag σ. The former indicates whether the component compor the context ctxis to blame for emitting this event, the latter indicates the secrecy level of values of the emitted event, i.e., either 🔒 or 🔓. As for pre-events, the memory-related ones are allocation (Alloc $l\ v$), deallocation (Dealloc l), reading from (Get $l\ v$) and writing to memory (Set $l\ v\ v'$). The following is an excerpt of the operational semantics handling some of the memory operations.

$$\boxed{r \xrightarrow{a}_p r'} \quad \text{„}r \text{ does one primitive step to } r' \text{ emitting event } a\text{.”}$$

$$\frac{t = \Omega.t \qquad \Omega.\Delta(x) = (l; t; \rho; m) \qquad l + n \in \text{dom}\ \Omega.H^t(l+n)}{\Omega \triangleright x[n] \xrightarrow{(\text{Get}\ l\ n; t)}_p \Omega \triangleright H^t(l+n)}\ (e-\text{get}-\in) \qquad \frac{t = \Omega.t \qquad \Omega.\Delta(x) = (l; t; \rho; m) \qquad l + n \notin \text{dom}\ \Omega.H^t}{\Omega \triangleright x[n] \longleftarrow v \xrightarrow{(\text{Set}\ l\ n\ v; t)}_p \Omega \triangleright v}\ (e-\text{set}-\notin)$$

$$\frac{\Omega \vdash z\ \text{fresh} \qquad \Omega \vdash l\ \text{fresh} \qquad H_1^t = \Omega.H^t \ll n \qquad \Delta_1 = z \mapsto (l; \Omega.t; \square; n), \Omega.\Delta}{\Omega \triangleright \textit{new}\ x\ [n]e \xrightarrow{(\text{Alloc}\ l\ n; t)}_p \Omega\left[H^t \text{ for } H_1^t\right][\Delta \text{ for } \Delta_1] \triangleright e\,[z \text{ for } x]}\ (e-\text{new})$$

$$\frac{\Omega.\Delta(x) = (l; \Omega.t; \rho; n) \qquad \Delta_1 = \Omega.\Delta(x \mapsto (l; \Omega.t; ☣; n))}{\Omega \triangleright \textit{delete}\ x \xrightarrow{(\text{Dealloc}\ l; t)}_p \Omega\,[\Delta \text{ for } \Delta_1] \triangleright 0}\ (e-\text{dealloc}) \qquad \frac{}{\Omega \triangleright \textit{abort}() \xrightarrow{\lightning}_p \lightning}\ (e-\text{abort})$$

To demonstrate the use of the poison tag ρ as metadata for pointers instead of removing them from the store Δ, consider Rules $e-$get$-\in$, $e-$set$-\notin$ and $e-$dealloc . In Rule $e-$dealloc, the premise does not care at all about the actual state of the poison tag ρ and just overwrites it, marking the location as freed ☣. Besides that, the poison tag does not have any semantic meaning. For language L_{tms}, this tag is really just some semantic metadata that programmers have no access to. But, for the other languages, e.g., L, the poison tag is used to check pointer validity. Rule $e-$new allocates enough space on the respective heap, either H^{ctx} or H^{comp} depending on the execution

[4]Throughout the paper, the substitution notation is also used to update entries in states Ω.

context, i.e., the value of $\Omega.t$, and adds the appropriate metadata associated to the pointer in Δ. Reading from Rule $e-\text{get}-\in$ and writing to memory Rule $e-\text{set}-\notin$ have two cases: Either the heap is large enough or not and, depending on that, either the actual value stored at that location is read from or written to, or some garbage data is returned. However, note that the execution does not get stuck in such cases, it performs a step, and emits an appropriate event. Also note that whether a pointer is poisoned or not is not reflected on the trace.

$$(\textit{Pre-Events})\ a_b ::= \cdots \mid \text{Call}\ c\ g\ v \mid \text{Ret}\ c\ v \mid \text{Start} \mid \text{End}\ v$$

A key difference in comparison with the specification trace model (Section 3.1) is that, as standard in secure compilation work [Abate et al. 2019; El-Korashy et al. 2021; Patrignani and Garg 2021], the traces have a call and return event that signals context switches, which are referred to as *interaction events*. The reason for these interaction events is technical: They are a proof artifact for reconstructing a source context from a potentially malicious target context, where during that translation, the insertion of some wrapper code right before context switching may be necessary to make the proof succeed. Hereby, a Call ? foo v and Return ? v signal that program execution transitions from context- to component-level. Contrary, Call ! foo v and Return ? v signal that program execution transitions from component- to context-level. For calls without this context switch, the environmental semantics attaches the $\varnothing$ tag. In the following, $\neg\text{ctx} = \text{comp}$ and $\neg\text{comp} = \text{ctx}$.

$\boxed{r \xrightarrow{a}_{\text{ectx}} r}$ „Contextual step from runtime-term r to r' emitting event a."

$$\frac{\Omega.\overline{K} = (K;\text{foo}), \overline{K'} \qquad \xi \vdash \text{foo} : \Omega.t \dashv c}{\Omega \triangleright K'\ [\mathit{return}\ v] \xrightarrow{(\text{Ret}\ c\ v;\Omega.t)}_{\text{ectx}} \Omega\ [\neg\Omega.t\ \text{for}\ t] \left[\overline{K'}\ \text{for}\ \overline{K}\right] \triangleright K\ [v]}\ (e-\text{ret})$$

$$\frac{fn\ \text{foo}\ x := e \in \Omega.\Xi \qquad \xi \vdash \text{foo} : \Omega.t \dashv c}{\Omega \triangleright K\ [\mathit{call}\ \text{foo}\ v] \xrightarrow{(\text{Call}\ c\ \text{foo}\ v;\Omega.t)}_{\text{ectx}} \Omega\ [\neg\Omega.t\ \text{for}\ t] \left[(K;\text{foo}), \Omega.\overline{K}\ \text{for}\ \overline{K}\right] \triangleright e\ [v\ \text{for}\ x]}\ (e-\text{call}-\text{notsame})$$

The environmental semantics is mostly straightforward. In Rules $e-\text{ret}$ and $e-\text{call}-\text{notsame}$, the judgement $\Omega.\xi \vdash \text{foo} : \Omega.t$ checks whether foo is a component-level name by looking it up in the list of component-level names ξ and emits the appropriate transfer tag, i.e., either ! or ?. Additional rules that are left out ensure that, e.g., when calling the main function, the event Start is emitted, which is a design choice this paper does for convenience when reasoning about call-chains. Note that the End v event is not emitted if the program crashes.

The top-level execution $\langle\, \Xi_{\text{ctx}}; \Xi_{\text{comp}} \rangle \overset{\overline{a}}{\Rightarrow} r$ constructs an initial state Ω by linking Ξ_{ctx} and Ξ_{ctx} and then starts execution by calling the main function. The trace $\overline{a}$ emitted during that execution serves as abstraction of the behavior of the program enabling the use of Definitions 2.1 and 2.2.

5.2 L_{tms}: A Temporal but Not Spatial Memory Safe Language

L_{tms} uses the same syntax as presented earlier (Section 5) without extensions to the term level. But, L_{tms} is statically typed, where the type system is inspired by L^3 [Morrisett et al. 2005; Scherer et al. 2018]. The type system of L_{tms} exhibits the property that every well-typed L_{tms} program satisfies TMS (Theorem 5.1). The proof of this theorem relies on a projection $\text{Proj}^{\mathsf{L}_{\text{tms}}}(\delta, \mathsf{a}) = a$ from L_{tms} events to specification events a, because the properties defined earlier (Section 3.1) are defined in the specification trace model. Hereby, the map $\delta(\mathsf{l}) = l$ maps an $\mathsf{L_{tms}}$ location to a location l of the specification trace model.

$$\boxed{\mathrm{Proj}^{\mathsf{L_{tms}}}(\delta, \mathsf{a}) = a} \quad \text{„Project an } \mathsf{L_{tms}} \text{ event } \mathsf{a} \text{ to a specification event } a\text{.”}$$

$$\frac{\mathsf{a_b} \neq \lightning}{\mathrm{Proj}^{\mathsf{L_{tms}}}(\delta, (\mathsf{a_b}; \mathsf{ctx}; \sigma)) = \varepsilon}\ (\mathsf{L_{tms}}\text{-filter-context}) \qquad \frac{}{\mathrm{Proj}^{\mathsf{L_{tms}}}(\delta, \lightning) = \lightning}\ (\mathsf{L_{tms}}\text{-filter-abort}) \qquad \frac{}{\mathrm{Proj}^{\mathsf{L_{tms}}}(\delta, (\mathsf{Start}; \mathsf{comp})) = \varepsilon}\ (\mathsf{L_{tms}}\text{-filter-start})$$

$$\frac{\delta(\mathsf{l}) = l \qquad \mathsf{n} = n}{\mathrm{Proj}^{\mathsf{L_{tms}}}(\delta, (\mathsf{Alloc\ l\ n}; \mathsf{comp})) = (\mathrm{Alloc}\ l\ n; \mathrm{comp}; 🔓)}\ (\mathsf{L_{tms}}\text{-filter-alloc})$$

Most rules of the projection $\mathrm{Proj}^{\mathsf{L_{tms}}}(\delta, \mathsf{a})$ are left out since, for the most part, it does the expected, e.g., $\mathrm{Proj}^{\mathsf{L_{tms}}}(\delta, \mathsf{Dealloc\ l}; \mathsf{comp}; \sigma) = \mathrm{Dealloc}\ \delta(l); \mathrm{comp}; \sigma$. But, it also filters any action that a context does as well as the interaction events, since these are irrelevant for component-level TMS.

Theorem 5.1 ($\mathsf{L_{tms}}$-programs are TMS). *For any* Ξ_{comp}, $\vdash_R \Xi_{comp} : \mathrm{tms}$

5.3 L: A Memory-Unsafe Language

$$(\textit{Expressions})\ \mathbf{e} ::= \cdots \mid \mathbf{x\ is\ ☣} \mid \mathbf{e\ has}\ \tau$$

L extends the syntax presented earlier (Section 5.1) with dynamic typechecks **e has** τ and a way to inspect poison tags **x is ☣** in the metadata of pointers. For valid pointers (□) bound to variable **x**, the check **x is ☣** yields **1**. If the array bound to **x** was allocated, i.e., has been poisoned (☣), the check **x is ☣** evaluates to **0**.

$$\frac{}{\Omega \triangleright \mathbf{x\ has}\ \mathbb{N}_t \xrightarrow{\varepsilon}_p \Omega \triangleright \mathbf{1}}\ (e - \mathbf{x\ has}\ \mathbb{N}_t) \qquad \frac{}{\Omega \triangleright \mathbf{n\ has}\ \mathbb{N}_t \xrightarrow{\varepsilon}_p \Omega \triangleright \mathbf{0}}\ (e - \mathbf{n\ has}\ \mathbb{N}_t)$$

Dynamic typechecks **e has** τ match on **e** and evaluate to **0** if the term is of type τ and **1** otherwise.

The projection $\mathrm{Proj}^{\mathbf{L}}(\delta, \overline{\mathbf{a}})$ is equal to $\mathrm{Proj}^{\mathsf{L_{tms}}}(\delta, \overline{\mathsf{a}})$.

5.4 L_{ms}: Another Memory-Unsafe Language

To enhance readablity, this paper uses L_{ms}, despite it being exactly equal to **L** (Section 5.3). The projection $\mathrm{Proj}^{L_{ms}}(\delta, \overline{a})$ is also exactly equal to $\mathrm{Proj}^{\mathbf{L}}(\delta, \overline{\mathbf{a}})$.

5.5 $\mathsf{L_{scct}}$: A Memory-Unsafe Language with a Data Independent Timing Mode

$$(\textit{Expressions})\ \mathtt{e} ::= \mathtt{n}^{\sigma} \mid \cdots \mid \mathtt{let}\ \mathtt{x}^{\sigma}\mathtt{=e_1\ in\ e_2} \mid \cdots \mid \mathtt{wrdoit\ e} \mid \mathtt{rddoit\ x\ in\ e}$$

$$(\textit{States})\ \Omega ::= (\Psi; \mathtt{t}; \mathtt{n}; \Phi)$$

$\mathsf{L_{scct}}$ extends L_{ms} (Section 5.4) with a way to write to a model specific register that controls a data (operand) independent timing mode, a feature that is present in both Arm [Arm 2020, p. 543] and Intel [Intel 2023, p. 80] processors. To this end, states are extended with the value of the register, which is initially set to be not active. If the register is marked active, the intuition is that no secrets can appear on specification traces. If the register is marked inactive, secrets may appear on traces. For the other languages seen earlier, the mode is intuitively always-on, i.e., the mode of execution always uses data independent timing. The language also adds user-annotations to values and variables to know their secrecy σ, which is either high 🔒 or low 🔓. Security tags σ are on the usual secrecy lattice, where 🔒 $\leq \sigma$ and $\sigma \leq$ 🔓.

$$(\textit{Pre-Events})\ \mathtt{a_b} ::= \cdots \mid \widehat{\mathtt{Get}}\ \mathtt{l}\ \mathtt{v} \mid \widehat{\mathtt{Set}}\ \mathtt{l}\ \mathtt{v}\ \mathtt{v'} \mid \mathtt{Branch}\ \mathtt{n} \mid \mathtt{Binop}\ \mathtt{n}$$

To prevent secrets from leaking but still enable reasoning about memory safety, $\mathrm{L_{scct}}$ extends pre-events with $\widehat{\mathtt{Get}}\ \mathtt{l}\ \mathtt{v}$ and $\widehat{\mathtt{Set}}\ \mathtt{l}\ \mathtt{v}$. These indicate reads from and writes to memory without leaking secret information involved in the access. Moreover, the language extends pre-events with $\mathtt{Branch}\ \mathtt{n}$ and $\mathtt{Binop}\ \mathtt{n}$ that are emitted when evaluating a branch or certain binary expressions, such as division, respectively, whenever the data independent timing mode is inactive. The following rules demonstrate how this is handled semantically.

$$\frac{\mathtt{m} \neq 0 \quad \mathtt{n_3} = \mathtt{n_1} \oplus \mathtt{n_2} \quad \sigma'' \leq \sigma \quad \sigma'' \leq \sigma'}{\Psi;\mathtt{t};\mathtt{m};\Phi \triangleright \mathtt{n_1^{\sigma}} \oplus \mathtt{n_2^{\sigma'}} \xrightarrow{\varepsilon}_p \Psi;\mathtt{t};\mathtt{m};\Phi \triangleright \mathtt{n_3^{\sigma''}}}\ (e-\oplus-\text{noleak}) \qquad \frac{}{\Psi;\mathtt{t};\mathtt{m};\Phi \triangleright \mathtt{wrdoit}\ \mathtt{n}^{\sigma} \xrightarrow{\varepsilon}_p \Psi;\mathtt{t};\mathtt{n};\Phi \triangleright \mathtt{n}^{\sigma}}\ (e-\mathtt{wrdoit})$$

$$\frac{}{\Psi;\mathtt{t};0;\Phi \triangleright \mathtt{ifz}\ 0^{\sigma}\ \mathtt{then}\ \mathtt{e_1}\ \mathtt{else}\ \mathtt{e_2} \xrightarrow{\mathtt{Branch}\ 0;\mathtt{t};\sigma}_p \Psi;\mathtt{t};0;\Phi \triangleright \mathtt{e_2}}\ (e-\mathtt{ifz}-\mathtt{true}-\text{leak})$$

The evaluation steps are amended to propagate the security-tag annotations σ. When the data independent timing mode is active, pre-events $\mathtt{Branch}\ \mathtt{n}$ and $\mathtt{Binop}\ \mathtt{n}$ are emitted for conditionals and binary operations, respectively.

$\boxed{\mathrm{Proj}^{\mathrm{L_{scct}}}(\delta, \mathtt{a}) = a}$ „Project an $\mathrm{L_{scct}}$ event $\mathtt{a}$ to a specification event a."

$$\frac{\mathtt{a_b} \neq \lightning}{\mathrm{Proj}^{\mathrm{L_{scct}}}(\delta, (\mathtt{a_b};\mathtt{ctx};\sigma)) = \varepsilon}\ (\mathrm{L_{scct}}\text{-filter-context}) \qquad \frac{\delta(\mathtt{l}) = l \quad \mathtt{n} = n}{\mathrm{Proj}^{\mathrm{L_{scct}}}\left(\delta, (\widehat{\mathtt{Get}}\ \mathtt{l}\ \mathtt{n};\mathtt{comp};\sigma)\right) = (\mathrm{Get}\ l\ n;\mathrm{comp};🔓)}\ (\mathrm{L_{scct}}\text{-filter-}\widehat{\text{get}})$$

$$\frac{\delta(\mathtt{l}) = l \quad \mathtt{n} = n}{\mathrm{Proj}^{\mathrm{L_{scct}}}(\delta, (\mathtt{Get}\ \mathtt{l}\ \mathtt{n};\mathtt{comp};\sigma)) = (\mathrm{Get}\ l\ n;\mathrm{comp};\sigma)}\ (\mathrm{L_{scct}}\text{-filter-get})$$

The projection to the specification trace model is mostly straightforward and similar to the others, e.g., Section 6.1. However, for events containing the pre-events $\widehat{\mathtt{Get}}$ and $\widehat{\mathtt{Set}}$, the projection always translates the security-tag σ to 🔓, regardless of its actual value, as seen in Rule $\mathrm{L_{scct}}$-filter-$\widehat{\text{get}}$. The pre-events themselves still translate to just Get and Set, respectively. With this technical setup, the information whether a read or write happened on a secret value is not hidden by the semantics, e.g., by emitting ε, but when projecting to specification events. This allows flexibility: The trace can be checked to satisfy different properties, such as, in this case, TMS, SMS, sCCT, and their combined versions. Example 5.2 illustrates the differences of $\mathrm{L_{scct}}$ compared to the other languages.

Example 5.2 ($\mathrm{L_{scct}}$ with and without data independent timing). Consider again the context presented in Example 3.8, where everything is marked with a security tag of high 🔒. The following table shows parts of the execution trace, read from top to bottom, in the left column (*Active*) with and in the right column (*Inactive*) without data independent timing. The left side of the table ($\mathrm{L_{scct}}$), i.e., the two columns on the left, describes the execution trace of the program, while the right side of the table (*Spec*), i.e., the two columns on the right, describes the respective projections $\mathrm{Proj}^{\mathrm{L_{scct}}}(\delta, \overline{\mathtt{a}})$ to the specification trace model.

L_{scct}		*Spec*	
Active	*Inactive*	*Active*	*Inactive*
Alloc $\mathtt{l_x}$ 12; comp; 🔓	Alloc $\mathtt{l_x}$ 12; comp; 🔓	Alloc l_x 12; comp; 🔓	Alloc l_x 12; comp; 🔓
Alloc $\mathtt{l_y}$ 12; comp; 🔓	Alloc $\mathtt{l_y}$ 12; comp; 🔓	Alloc l_y 12; comp; 🔓	Alloc l_x 12; comp; 🔓
$\widehat{\text{Get}}$ $\mathtt{l_x}$ 0; comp; 🔒	Get $\mathtt{l_x}$ 0; comp; 🔒	Use l_x 0; comp; 🔓	Use l_x 0; comp; 🔒
ε	Branch 0; comp; 🔒	ε	Branch 0; comp; 🔒
$\widehat{\text{Get}}$ $\mathtt{l_x}$ 0; comp; 🔒	Get $\mathtt{l_x}$ 0; comp; 🔒	Use l_x 0; comp; 🔓	Use l_x 0; comp; 🔒
$\widehat{\text{Set}}$ $\mathtt{l_y}$ 0 'H'; comp; 🔒	Set $\mathtt{l_y}$ 0 'H'; comp; 🔒	Use l_y 0; comp; 🔓	Use l_y 0; comp; 🔒
$\widehat{\text{Get}}$ $\mathtt{l_x}$ 1; comp; 🔒	Get $\mathtt{l_x}$ 1; comp; 🔒	Use l_x 1; comp; 🔓	Use l_x 1; comp; 🔒
ε	Branch 0; comp; 🔒	ε	Branch 0; comp; 🔒
⋮	⋮	⋮	⋮
$\widehat{\text{Get}}$ $\mathtt{l_x}$ 12; comp; 🔒	Get $\mathtt{l_x}$ 12; comp; 🔒	Use l_x 12; comp; 🔓	Use l_x 12; comp; 🔒
ε	Branch 1; comp; 🔒	ε	Branch 1; comp; 🔒
Dealloc $\mathtt{l_y}$; comp; 🔓	Dealloc $\mathtt{l_y}$; comp; 🔒	Dealloc l_y; comp; 🔓	Dealloc l_y; comp; 🔓
$\widehat{\text{Get}}$ $\mathtt{l_y}$ 6; comp; 🔒	Get $\mathtt{l_y}$ 6; comp; 🔒	Use l_y 6; comp; 🔓	Use l_y 6; comp; 🔒

When the data independent timing mode is off, the execution yields events in similar fashion to before (Sections 5.2 to 5.4). But, if it is turned on, then the branching event does not fire anymore and both reading and writing to memory gets ultimately translated to a specification trace with no exposed secrets.

6 CASE STUDY: COMPOSING SECURE COMPILER PASSES AND OPTIMISATIONS

This section defines several secure compilers, each of which robustly preserves a different property of interest as depicted in Figure 3. The section demonstrates the power of the framework (Sections 3

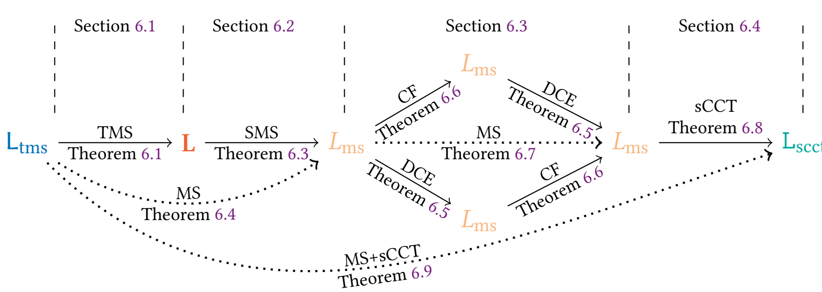

Fig. 3. Visualisation of the optimising compilation pipeline that attains a combination of MS and CCT. Vertices in the graph are the programming languages from earlier sections (Section 5). All edges are secure compilers, but dotted edges use the presented framework (Section 4) and strikethrough edges classic proof techniques. The dashed lines partition the graph into the sections where the respective theorems are presented.

and 4) by composing these compilers for a secure and optimising compilation chain that robustly preserves MS+sCCT. The first step in this chain is the compiler from L_{tms} to L that robustly preserves just TMS (Theorem 6.1). From here, an instrumentation from L to L_{ms} ensures that no out-of-bounds accesses can happen and, thus, programs at this point attain SMS (Theorem 6.3). Since these properties compose into MS, composing these passes yields a compiler that robustly preserves MS (Theorem 6.4). At this stage, the section presents two optimising translations, namely CF and DCE, each of which robustly preserves MS (Theorems 6.5 and 6.6). These translations can be freely ordered in the compilation chain without compromising memory safety (Theorem 6.7). The last step of the chain ensures that code stays sCCT (Theorem 6.8) when lowered from L_{ms} to L_{scct}. The final result is that the whole compilation chain robustly preserves MS+sCCT (Theorem 6.9).

6.1 Robust Temporal Memory Safety Preservation

This subsection defines a secure compiler from L_{tms} to $\mathbf{L}$. To this end, the compiler needs to ensure that when execution switches from context to component, the type signatures are respected. It can do so by inserting dynamic typechecks prior to entering the body of a function belonging to the component.

$$\gamma_{\mathbf{L}}^{L_{tms}}(x) = \mathbf{x}$$
$$\gamma_{\mathbf{L}}^{L_{tms}}(n) = \mathbf{n}$$
$$\gamma_{\mathbf{L}}^{L_{tms}}(e_1 \oplus e_2) = \left[\gamma_{\mathbf{L}}^{L_{tms}}(e_1)\right] \oplus \left[\gamma_{\mathbf{L}}^{L_{tms}}(e_2)\right]$$
$$\gamma_{\mathbf{L}}^{L_{tms}}(x[e]) = \mathbf{x}\left[\left[\gamma_{\mathbf{L}}^{L_{tms}}(e)\right]\right]$$
$$\gamma_{\mathbf{L}}^{L_{tms}}(\mathsf{delete}\ x) = \mathbf{delete}\ \left[\gamma_{\mathbf{L}}^{L_{tms}}(x)\right]$$
$$\gamma_{\mathbf{L}}^{L_{tms}}(\mathsf{fn}\ g\ x : \mathbb{N}_t \to \tau_e := e) = \mathbf{fn\ g\ x := ifz\ x\ has}\ \mathbb{N}_t\ \mathbf{then}\ \left[\gamma_{\mathbf{L}}^{L_{tms}}(e)\right]\ \mathbf{else\ abort()}$$

Since $\mathbf{L}$ has no static typechecks, it could happen that a bogus context Ξ_{ctx} invokes a callable object accepting a $\mathbb{N}_t$ with $\langle 17;29 \rangle$. By inserting the check, the compiler ensures that execution does not proceed in such cases. The compiler does not insert other checks and proceeds as the identity function (which in this paper amounts to a simple re-colouring of L_{tms} to $\mathbf{L}$ expressions).

Compiling the `strncpy` function from Section 1 with $\gamma_{\mathbf{L}}^{L_{tms}}$, the compiler would in this case ensure that the arguments that are evaluated in the compiled `strncpy` are valid.

Theorem 6.1 (Compiler $\gamma_{\mathbf{L}}^{L_{tms}}$ is secure with respect to TMS). $\vdash \gamma_{\mathbf{L}}^{L_{tms}} : \mathrm{tms}$

6.1.1 Proving Robust Safety Property Preservation. We illustrate the proof of Theorem 6.1 since the other secure compilation proofs of this paper follow the same approach. Unfolding the theorem statement yields the following assumptions: for any $\pi \in \lceil \mathrm{tms} \rceil$[5], $\bar{a}$, $\mathbf{r}$, and component Ξ_{comp}, we have that $\vdash_R \Xi_{comp} : \pi$ and $\langle\, \Xi_{ctx}; \gamma_{\mathbf{L}}^{L_{tms}}(\Xi_{comp}) \rangle \overset{\bar{\mathbf{a}}}{\Longrightarrow} \mathbf{r}$, where Ξ_{ctx} is arbitrary. The proof obligation is $\mathrm{Proj}^{\mathbf{L}}(\delta, \bar{\mathbf{a}}) \in \pi$, i.e., the specification trace associated to $\bar{\mathbf{a}}$ satisfies the property π. A way to show this is to relate trace $\bar{\mathbf{a}}$ to some L_{tms} trace $\bar{a}$ (which already satisfies the property as per the assumptions). The assumptions already contain a target execution associated to this trace, so the task is to find an associated L_{tms} execution that yields $\bar{\mathbf{a}}$. The trace $\bar{\mathbf{a}}$ is split into different parts, as commonly done in secure compilation works [Abate et al. 2018; El-Korashy et al. 2021], where each part contains the events that either the context or the component does, but not both. Because of this, all such trace segments are „well-bracketed" in the sense that they start with either Start, Call ! **foo v**, or Ret ! **v** and end with either End **v**, Ret ? **v**, or Call ? **foo v**. In the following, the former is referred to as a context segment, since these executions happen in Ξ_{ctx}, and the latter is referred to as a component segment, since these executions happen in $\gamma_{\mathbf{L}}^{L_{tms}}(\Xi_{comp})$. Figure 4 visualises this division for a program execution with one call from context to component and how

[5] $\lceil\cdot\rceil$ lifts the property to a hyperproperty by applying the powerset operation [Clarkson and Schneider 2008].

the target execution is related to a source execution. In the figure, the green dashed lines encompass the component segments while the orange boxes contain the actual context switches from context to component or vice versa. From a technical perspective, as typically done in compilation proofs, the proof requires some setup to maintain a relation between Ω and Ω . Two cross-language relations make this precise: (i) $\multimap_\delta$ relates states that are involved in a context segment, allowing the target execution to perform internal calls, and (ii) $\approx_\delta$ relates states that are involved in a component segment, where both states need to agree exactly, i.e., the memory and the control flow states are required to contain the same information. The relations are indexed with δ, which is an injective mapping from $\mathsf{L}_{\mathsf{tms}}$ locations l to L locations l. Note that the relations $\multimap_\delta$ and $\approx_\delta$ swap when context switching.

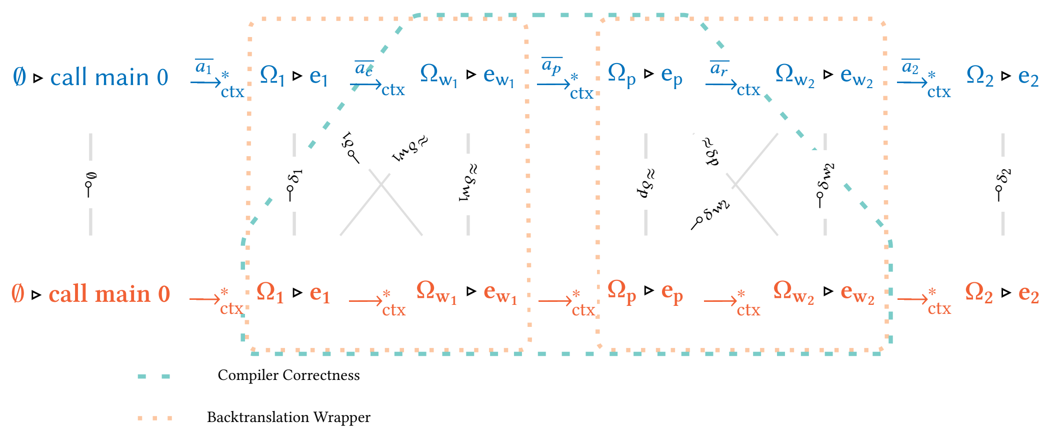

Fig. 4. Proof diagram for Theorem 6.1 depicting the general structure of robust preservation proofs. Nodes in the graph represent runtime states. Vertical lines indicate cross language relations, while horizontal ones are execution steps. The green dashed trapezoid encompasses the component segment, while the orange dotted rectangles entail the context switches. L traces are omitted for readability. $\mathsf{L}_{\mathsf{tms}}$ trace segments $\overline{\mathsf{a_c}}$ and $\overline{\mathsf{a_r}}$ describe the events that happen at the boundaries, i.e., during a context switch. $\overline{\mathsf{a_p}}$ is the behavior of the component and the traces $\overline{\mathsf{a_1}}$ and $\overline{\mathsf{a_2}}$ describe the context.

So far, the paper explained how to relate an $\mathsf{L}_{\mathsf{tms}}$ execution with a L execution. The next question is therefore how to build the corresponding $\mathsf{L}_{\mathsf{tms}}$ execution. This is done using a standard secure compilation proof technique called trace-based backtranslation [Abate et al. 2019; El-Korashy et al. 2021; Patrignani and Garg 2021], which can be used to build a context Ξ_{ctx} that behaves similar to Ξ_{ctx}. For context segments of the trace $\overline{\mathsf{a}}$ it is also necessary to show that the execution behaves similarly, i.e., the context obtained from the backtranslation generates trace $\overline{\mathsf{a}}$. For component segments of the trace, the relatedness of states and traces follows from a compiler correctness argument. These two arguments yield the source execution $\langle\, \Xi_{\mathsf{ctx}}; \Xi_{\mathsf{comp}} \rangle \overset{\overline{\mathsf{a}}}{\Longrightarrow} \mathsf{r}$.

The proof now works as follows. Given that $\mathsf{Proj}^{\mathsf{L}}\,(\delta, \overline{\mathsf{a}}) = \mathsf{Proj}^{\mathsf{L}_{\mathsf{tms}}}\,(\delta, \overline{\mathsf{a}})$, the proof goal changes from $\mathsf{Proj}^{\mathsf{L}}\,(\delta, \overline{\mathsf{a}}) \in \pi$ to $\mathsf{Proj}^{\mathsf{L}_{\mathsf{tms}}}\,(\delta, \overline{\mathsf{a}}) \in \pi$. This follows by specializing the robust satisfaction assumption $\vdash_R \Xi_{\mathsf{comp}} : \pi$ to use the context Ξ_{ctx}, which is obtained from the backtranslation, and to use the source execution $\langle\, \Xi_{\mathsf{ctx}}; \Xi_{\mathsf{comp}} \rangle \overset{\overline{\mathsf{a}}}{\Longrightarrow} \mathsf{r}$.

6.2 Robust (Spatial) Memory Safety Preservation

$$\begin{aligned}
\gamma^{\mathbf{L}}_{L_{ms}}(\mathbf{new\ x\ [e_1]e_2}) &= \mathit{let}\ x_{SIZE}{=}\gamma^{\mathbf{L}}_{L_{ms}}(\mathbf{e_1})\ \mathit{in\ new}\ x\ [x_{SIZE}]\gamma^{\mathbf{L}}_{L_{ms}}(\mathbf{e_2})\\
\gamma^{\mathbf{L}}_{L_{ms}}(\mathbf{x[e]}) &= \mathit{let}\ x_n{=}\gamma^{\mathbf{L}}_{L_{ms}}(\mathbf{e})\ \mathit{in\ ifz}\ 0 \leq x_n < x_{SIZE}\ \mathit{then}\ x[x_n]\ \mathit{else\ abort}()\\
\gamma^{\mathbf{L}}_{L_{ms}}(\mathbf{x[e_1]} \leftarrow \mathbf{e_2}) &= \mathit{let}\ x_n{=}\gamma^{\mathbf{L}}_{L_{ms}}(\mathbf{e_1})\ \mathit{in\ ifz}\ 0 \leq x_n < x_{SIZE}\ \mathit{then}\ x[x_n] \leftarrow \gamma^{\mathbf{L}}_{L_{ms}}(\mathbf{e_2})\ \mathit{else\ abort}()
\end{aligned}$$

The compiler $\gamma^{\mathbf{L}}_{L_{ms}}$ only inserts bounds-checks whenever reading from or writing to memory in order to enforce SMS. For passing pointers, it has to pass them with their size information as well. To this end, the compiler introduces another, fresh identifier x_{SIZE} for each allocation that binds x to keep track of the allocation size.

Example 6.2 (Instrumented `strncpy`*).* Consider again `strncpy`, but instrumented for SMS:

```
void strncpy(size_t n, size_t dst_size, char *dst,
                       size_t src_size, char *src) {
  for(size_t i = 0; i < src_size && src[i] != '\0' && i < n; ++i) {
    if(i < src_size && i < dst_size) {
      dst[i] = src[i];
    }
  }
}
```

When calling this in similar fashion to Example 3.10, the event Use l_x 2; comp; 🔓 would not be emitted during execution, since the bounds check prevents the condition `src[i] != '\0'` from executing.

Theorem 6.3 (Compiler $\gamma^{\mathbf{L}}_{L_{ms}}$ is secure with respect to SMS). $\vdash \gamma^{\mathbf{L}}_{L_{ms}}$: sms

Theorem 6.4 states that the composition of $\gamma^{\mathsf{L_{tms}}}_{\mathbf{L}}$ and $\gamma^{\mathbf{L}}_{L_{ms}}$ is secure with respect to MS and follows from Theorems 6.1 and 6.3 using Theorem 4.2.

Theorem 6.4 (Compiler $\gamma^{\mathsf{L_{tms}}}_{\mathbf{L}} \circ \gamma^{\mathbf{L}}_{L_{ms}}$ is secure with respect to MS). $\vdash \gamma^{\mathsf{L_{tms}}}_{\mathbf{L}} \circ \gamma^{\mathbf{L}}_{L_{ms}}$: ms

Proof. From Theorem 6.1 (Compiler $\gamma^{\mathsf{L_{tms}}}_{\mathbf{L}}$ is secure with respect to TMS) it follows that for any $\mathsf{L_{tms}}$ program p, it compiles to an $\mathbf{L}$ program $\mathbf{p}$ that robustly satisfies TMS. Note that p robustly satisfies TMS by the properties of the typesystem of $\mathsf{L_{tms}}$. Then, Theorem 6.3 (Compiler $\gamma^{\mathbf{L}}_{L_{ms}}$ is secure with respect to SMS) demonstrates that, assuming $\mathbf{p}$ robustly satisfies SMS, the program $\mathbf{p}$ compiles to an L_{ms} program p that also robustly satisfies SMS. From Theorem 6.4 (Compiler $\gamma^{\mathsf{L_{tms}}}_{\mathbf{L}} \circ \gamma^{\mathbf{L}}_{L_{ms}}$ is secure with respect to MS) it follows that p compiles to p that robustly satisfies MS, since MS is the intersection of TMS and SMS. □

6.3 Optimising Compilers

$$\gamma_{DCE}{}^{L_{ms}}_{L_{ms}}(\mathit{ifz}\ \mathit{true}\ \mathit{then}\ e_1\ \mathit{else}\ e_2) = \gamma_{DCE}{}^{L_{ms}}_{L_{ms}}(e_1)$$
$$\gamma_{DCE}{}^{L_{ms}}_{L_{ms}}(\mathit{ifz}\ \mathit{false}\ \mathit{then}\ e_1\ \mathit{else}\ e_2) = \gamma_{DCE}{}^{L_{ms}}_{L_{ms}}(e_2)$$
$$\gamma_{DCE}{}^{L_{ms}}_{L_{ms}}(e_1 \oplus e_2) = \gamma_{DCE}{}^{L_{ms}}_{L_{ms}}(e_1) \oplus \gamma_{DCE}{}^{L_{ms}}_{L_{ms}}(e_2)$$

$$\gamma_{CF}{}^{L_{ms}}_{L_{ms}}(e) = \mathtt{mix}(e, [\cdot])$$

$$\begin{aligned}
\mathtt{mix}(x, \overline{\gamma}) &= n && \text{if } [n \text{ for } x] \in \overline{\gamma} \\
\mathtt{mix}(x, \overline{\gamma}) &= x && \text{if } [n \text{ for } x] \notin \overline{\gamma} \\
\mathtt{mix}(n \oplus m, \overline{\gamma}) &= k && \text{if } n \oplus m = k \\
\mathtt{mix}(\mathit{let}\ x{=}n\ \mathit{in}\ e, \overline{\gamma}) &= \mathtt{mix}(e, [x \text{ for } n], \overline{\gamma}) \\
\mathtt{mix}(x[e], \overline{\gamma}) &= x[\mathtt{mix}(e, \overline{\gamma})] \\
\mathtt{mix}(\mathit{let}\ x{=}e_1\ \mathit{in}\ e_2, \overline{\gamma}) &= \mathit{let}\ x{=}\mathtt{mix}(e_1, \overline{\gamma})\ \mathit{in}\ \mathtt{mix}(e_2, \overline{\gamma}) \\
\mathtt{mix}(\mathit{ifz}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3, \overline{\gamma}) &= \mathit{ifz}\ \mathtt{mix}(e_1, \overline{\gamma})\ \mathit{then}\ \mathtt{mix}(e_2, \overline{\gamma})\ \mathit{else}\ \mathtt{mix}(e_3, \overline{\gamma})
\end{aligned}$$

The two optimising compiler passes from L_{ms} to L_{ms} perform DCE and CF, respectively. The DCE pass applies a naive rewrite rule on conditionals. For CF, the pass uses an auxiliary function `mix` that does the actual work. It rewrites constant binary operations, e.g., $17 - 1$ to 16, and replaces variables that are assigned to constants with their constant, e.g., $\mathit{let}\ x{=}7\ \mathit{in}\ x$ to 7. Both passes are secure with respect to MS. The proof for either is relatively simple, because both DCE and CF do not change the way memory accesses happen. Moreover, since the input and output languages to these compilers are the same, attacker contexts do not have more power in the target language than in the source.

Theorem 6.5 (Compiler $\gamma_{DCE}{}^{L_{ms}}_{L_{ms}}$ is secure with respect to MS). $\vdash \gamma_{DCE}{}^{L_{ms}}_{L_{ms}} : \mathrm{ms}$

Theorem 6.6 (Compiler $\gamma_{CF}{}^{L_{ms}}_{L_{ms}}$ is secure with respect to MS). $\vdash \gamma_{CF}{}^{L_{ms}}_{L_{ms}} : \mathrm{ms}$

With both Theorems 6.5 and 6.6 it follows from Corollary 4.3 that the two passes can be interchanged arbitrarily:

Theorem 6.7 (Compilers $\gamma_{CF}{}^{L_{ms}}_{L_{ms}} \circ \gamma_{DCE}{}^{L_{ms}}_{L_{ms}}$ and $\gamma_{CF}{}^{L_{ms}}_{L_{ms}} \circ \gamma_{DCE}{}^{L_{ms}}_{L_{ms}}$ are secure with respect to MS). $\vdash \gamma_{CF}{}^{L_{ms}}_{L_{ms}} \circ \gamma_{DCE}{}^{L_{ms}}_{L_{ms}} : \mathrm{ms}$ *and* $\vdash \gamma_{DCE}{}^{L_{ms}}_{L_{ms}} \circ \gamma_{CF}{}^{L_{ms}}_{L_{ms}} : \mathrm{ms}$.

6.4 Robust Strict Cryptographic Constant Time Preservation

$$\gamma^{L_{ms}}_{L_{scct}}(\mathit{fn}\ g\ x := e) = \mathtt{fn\ g\ x := wrdoit\ 1;}\gamma^{L_{ms}}_{L_{scct}}(e)$$
$$\gamma^{L_{ms}}_{L_{scct}}(\mathit{call}\ g\ e) = \mathtt{call\ g}\ \gamma^{L_{ms}}_{L_{scct}}(e)\mathtt{;wrdoit1}$$
$$\gamma^{L_{ms}}_{L_{scct}}(e_1 \oplus e_2) = \gamma^{L_{ms}}_{L_{scct}}\ e_1 \oplus \gamma^{L_{ms}}_{L_{scct}}\ e_2$$

Given the fact that L_{scct} provides a CCT-mode that can be turned on or off, the compiler inserts wrapper code for function bodies to ensure that execution in the component always happen in this CCT-mode. The context can overwrite the flag and exit the mode, but upon invoking a function that is part of the component, the flag would be set again. Because of this, the compiler is secure with respect to sCCT, similarly proven as in Section 6.1.

Theorem 6.8 (Compiler $\gamma_{\mathrm{L_{scct}}}^{L_{\mathrm{ms}}}$ is secure with respect to sCCT). $\vdash \gamma_{\mathrm{L_{scct}}}^{L_{\mathrm{ms}}} : \mathrm{scct}$

6.5 Robust Preservation of Intersection of Memory Safety and Strict Cryptographic Constant Time

Let $\gamma_{\mathrm{L_{scct}}}^{\mathrm{L_{tms}}}$ be the compiler that is the composition of $\gamma_{\mathrm{L}}^{\mathrm{L_{tms}}}$, $\gamma_{L_{\mathrm{ms}}}^{\mathrm{L}}$, $\gamma CF_{L_{\mathrm{ms}}}^{L_{\mathrm{ms}}}$, $\gamma DCE_{L_{\mathrm{ms}}}^{L_{\mathrm{ms}}}$, and $\gamma_{\mathrm{L_{scct}}}^{L_{\mathrm{ms}}}$, then the following theorem holds.

Theorem 6.9 (Compiler $\gamma_{\mathrm{L_{scct}}}^{\mathrm{L_{tms}}}$ is secure with respect to sCCT). $\vdash \gamma_{\mathrm{L_{scct}}}^{\mathrm{L_{tms}}} : \mathrm{ms} \cap \mathrm{scct}$

Proof. From Theorem 6.4 (Compiler $\gamma_{\mathrm{L}}^{\mathrm{L_{tms}}} \circ \gamma_{L_{\mathrm{ms}}}^{\mathrm{L}}$ is secure with respect to MS), we have that any $\mathrm{L_{tms}}$ program p compiles into a L_{ms} program p that robustly satisfies MS. Then, from Theorem 6.7 (Compilers $\gamma CF_{L_{\mathrm{ms}}}^{L_{\mathrm{ms}}} \circ \gamma DCE_{L_{\mathrm{ms}}}^{L_{\mathrm{ms}}}$ and $\gamma CF_{L_{\mathrm{ms}}}^{L_{\mathrm{ms}}} \circ \gamma DCE_{L_{\mathrm{ms}}}^{L_{\mathrm{ms}}}$ are secure with respect to MS) we have that p gets optimised to a program p' that is also MS, where the order of optimisations does not matter for p' to be MS. Assuming p' robustly satisfies sCCT, by Theorem 6.8 (Compiler $\gamma_{\mathrm{L_{scct}}}^{L_{\mathrm{ms}}}$ is secure with respect to sCCT) it compiles to an $\mathrm{L_{scct}}$ program p that robustly satisfies sCCT as well. Finally, from Theorem 4.2 (Sequential Composition of Secure Compilers) it follows that, given p robustly satisfies sCCT and MS, p also robustly satisfies sCCT and MS. □

7 RELATED WORK

This section discusses work on robust compilation (Section 7.1) and on other secure compilation criteria (Section 7.2). Since the case study of Sections 5 and 6 implements measures for preserving MS and CCT, this section then presents relevant related work as well (Sections 7.3 and 7.4).

7.1 Secure Compilation as Robust Preservation

The robust preservation of properties as a compiler-level criterion has been analyzed extensively [Abate et al. 2021a, 2019; Patrignani et al. 2019; Patrignani and Garg 2021] and thus we build on that framework. No existing work is concerned with composing robustly safe compilers. These works consider languages with different trace models and our technical setup can be adapted to that as long as security properties and their monitors are still defined on the same trace model. The work relating robust preservation with universal composability [Patrignani et al. 2022] is closest to what this paper presents. The authors demonstrate a similar compositionalty theorem to what is presented here (Section 4) but use it in the context of protocols. They do not demonstrate the scalability of the approach. Moreover, they are missing the upper and lower compositions.

7.2 Other Secure Compilation Criteria

While this paper focuses on the robust preservation framework [Abate et al. 2019], other secure compilation criteria exist. The survey on formal approaches to secure compilation [Patrignani et al. 2019] discusses a broad spectrum already, while this section presents a very high-level overview. Fully abstract compilation [Abadi 1999b] states that a compiler should preserve and reflect observational equivalence between source and target programs. It was shown [Abate et al. 2021b] that fully abstract compilers robustly preserve program properties that are either trivial or meaningless. As a mitigation for this, the authors presented a categorical approach based on maps of distributive laws [Watanabe 2002], which they call many maps of distributive laws. Maps of distributive laws have been investigated before as a possible secure compilation criterion [Tsampas et al. 2020]. Other approaches are extensions of the compiler correctness criterion as discussed in other work [Patterson and Ahmed 2019] or the introduction of opaque observations [Vu et al.

2021] to reconcile compiler optimisations with security. Note that this work also presents secure compilers that are optimising, but contrary to the other [Vu et al. 2021], provides a formal account of these in the robust preservation framework.

7.3 Memory Safety Mechanisms

Different mechanisms for enforcing memory safety exist that also consider the secure compilation domain, i.e., have an active attacker model. For example, the „pointers as capabilities" principle represents pointers as machine-level capabilities [El-Korashy et al. 2021], which behave in a similar fashion to capabilities by means of linear typing [Morrisett et al. 2005]. The approach of this paper also uses linear typing, but differs from L^3 [Morrisett et al. 2005] in the way that functions are not first-class. Moreover, this paper considers an active attacker, while the work on L^3 only discusses whole programs and, thus, has no active attacker model. The instrumentation to ensure memory safety that this paper presents is inspired by Softbounds [Nagarakatte et al. 2009]. That work inserts bounds-checks in front of pointer-dereferences and, for this to work, inserts meta-data information on pointer creation. Softbounds also works in a more advanced setting with structured fields accesses and also introduces a table-lookup for pointers that are stored in memory. This paper only considers arrays of primitive data, i.e., there are no pointers to pointers or structures. Several other approaches to memory-safety exist in literature, specifically as compiler instrumentations [Akritidis et al. 2009; Dhumbumroong and Piromsopa 2020; Jung et al. 2021; Nam et al. 2019; Shankaranarayana et al. 2023; Younan et al. 2010; Zhou et al. 2023], hardware-extensions [Chen et al. 2023; Kim et al. 2023; Kwon et al. 2013; Saileshwar et al. 2022], or programming language extensions [Benoit and Jacobs 2019; Elliott et al. 2018, 2015; Jim et al. 2002; Li et al. 2022; Weis et al. 2019; West and Wong 2005]. What differentiates this work from them is that this work uses known, compiler-based approaches to ensure memory-safety as a means to investigate secure compiler compositions. This paper does not provide efficient memory-safety, but serves as a theoretical foundation for the secure compilation domain.

To extend the languages in this paper with a less restricted form of pointer arithmetic, the region coloring memory safety monitor presented in earlier work [Michael et al. 2023] can be used. The work presenting this monitor provides an approach for the robust preservation of memory safety compiling from C to WASM. However, they do not discuss composition of secure compilers but rather investigate an instance of a secure compiler.

7.4 Cryptographic Constant Time Mechanisms

The approach to preserving cryptographic constant time in this paper is high-level, where a programming language exposes a way to switch the semantics to a data (operand) independent timing mode. Since identifiers in $\mathrm{L}_{\mathrm{scct}}$ are annotated with a secrecy tag, this approach is similar to others with information flow control. For example, Vale [Bond et al. 2017] uses Dafny to ensure constant-time assembly code, while Jasmin [Almeida et al. 2017] makes use of the Coq proof assistant to reject non-constant-time programs. CT-Wasm [Watt et al. 2019] enforces constant-timeness by means of a type system. Different to the approach of this paper, these approaches necessitate that the programmer writes CCT code. An approach to allow programmers to write more high-level code is CryptOpt [Kuepper et al. 2023], which generates efficient target-code by means of a randomised search. This paper abstracts over concrete mitigation strategies and simply assumes that there is a flag to switch to a cryptographic-constant time execution mode. This can be realised by employing the FaCT [Cauligi et al. 2019] compiler, which translates common non-constant time code patterns to be constant-time, and the data (object) independent timing execution mode of modern processors.

8 CONCLUSION

This paper tackled the problem of understanding what kind of security properties does a secure compiler preserve, when said compiler is the combination of compiler passes that preserve possibly different security properties. For this, this paper first formalised security properties of interest and their composition. Then, it proved that composing secure compilers that preserve certain properties results in a secure compiler that preserves the composition of these properties. Finally, this paper defines a multi-pass compiler and proves that it preserves MS+sCCT. Crucially, this paper derives the security of the multi-pass compiler from the composition of the security properties preserved by its individual passes, which include security-preserving as well as optimisation passes.

REFERENCES

Martín Abadi. 1999a. *Protection in Programming-Language Translations.* Springer Berlin Heidelberg, Berlin, Heidelberg, 19–34. https://doi.org/10.1007/3-540-48749-2_2

Martín Abadi. 1999b. *Protection in Programming-Language Translations.* Springer Berlin Heidelberg, Berlin, Heidelberg, 19–34. https://doi.org/10.1007/3-540-48749-2_2

Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1351–1368. https://doi.org/10.1145/3243734.3243745

Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2021a. An Extended Account of Trace-Relating Compiler Correctness and Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 4, Article 14 (nov 2021), 48 pages. https://doi.org/10.1145/3460860

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 *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 256–25615. https://doi.org/10.1109/CSF.2019.00025

Carmine Abate, Matteo Busi, and Stelios Tsampas. 2021b. Fully Abstract and Robust Compilation. In *Programming Languages and Systems*, Hakjoo Oh (Ed.). Springer International Publishing, Cham, 83–101.

Amal Ahmed and Matthias Blume. 2011. An Equivalence-Preserving CPS Translation via Multi-Language Semantics. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 431–444. https://doi.org/10.1145/2034773.2034830

Amal Ahmed, Deepak Garg, Catalin Hritcu, and Frank Piessens. 2018. Secure Compilation (Dagstuhl Seminar 18201). *Dagstuhl Reports* 8, 5 (2018), 1–30. https://doi.org/10.4230/DagRep.8.5.1

Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against out-of-Bounds Errors. In *Proceedings of the 18th Conference on USENIX Security Symposium* (Montreal, Canada) *(SSYM'09)*. USENIX Association, USA, 51–66.

José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1807–1823. https://doi.org/10.1145/3133956.3134078

Arm. 2020. *Arm®A-profile Architecture Registers.* Accessed: 2023-06-09.

Arm. 2022. *Morello for A-profile Architecture.* Accessed: 2023-06-10.

Arthur Azevedo de Amorim, Cătălin Hriţcu, and Benjamin C. Pierce. 2018. The Meaning of Memory Safety. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 79–105.

Michael Backes, Cătălin Hriţcu, and Matteo Maffei. 2014. Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations. *Journal of Computer Security* 22, 2 (2014), 301–353. https://doi.org/10.3233/jcs-130493

Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. 328–343. https://doi.org/10.1109/CSF.2018.00031

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.* 33, 2, Article 8 (feb 2011), 45 pages. https://doi.org/10.1145/1890028.1890031

Tuur Benoit and Bart Jacobs. 2019. Uniqueness Types for Efficient and Verifiable Aliasing-Free Embedded Systems Programming. In *International Conference on Integrated Formal Methods.*

BlockSec. 2021. CVE-2021-3564. Available from MITRE, CVE-ID CVE-2021-2564.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-2564

Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 2017. Vale: Verifying High-Performance Cryptographic Assembly Code. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 917–934. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/bond

William J. Bowman and Amal Ahmed. 2015. Noninterference for Free. *SIGPLAN Not.* 50, 9 (aug 2015), 101–113. https://doi.org/10.1145/2858949.2784733

Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. 2006. Universally Composable Security with Global Setup. Cryptology ePrint Archive, Paper 2006/432. https://eprint.iacr.org/2006/432 https://eprint.iacr.org/2006/432.

Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: A DSL for Timing-Sensitive Computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 174–189. https://doi.org/10.1145/3314221.3314605

Dongwei Chen, Dong Tong, Chun Yang, Jiangfang Yi, and Xu Cheng. 2023. FlexPointer: Fast Address Translation Based on Range TLB and Tagged Pointers. *ACM Trans. Archit. Code Optim.* 20, 2, Article 30 (mar 2023), 24 pages. https://doi.org/10.1145/3579854

Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*. 51–65. https://doi.org/10.1109/CSF.2008.7

Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (mar 1995), 181–196. https://doi.org/10.1145/201059.201061

Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. 1999. Optimizing for Reduced Code Space Using Genetic Algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems* (Atlanta, Georgia, USA) *(LCTES '99)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/314403.314414

Dominique Devriese, Marco Patrignani, and Frank Piessens. 2017a. Parametricity versus the Universal Type. *Proc. ACM Program. Lang.* 2, POPL, Article 38 (dec 2017), 23 pages. https://doi.org/10.1145/3158126

Dominique Devriese, Marco Patrignani, Frank Piessens, and Steven Keuchel. 2017b. Modular, Fully-abstract Compilation by Approximate Back-translation. *Logical Methods in Computer Science* Volume 13, Issue 4 (Oct. 2017). https://doi.org/10.23638/LMCS-13(4:2)2017

Smith Dhumbumroong and Krerk Piromsopa. 2020. BoundWarden: Thread-enforced spatial memory safety through compile-time transformations. *Science of Computer Programming* 198 (2020), 102519. https://doi.org/10.1016/j.scico.2020.102519

Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2021. CapablePtrs: Securely Compiling Partial Programs Using the Pointers-as-Capabilities Principle. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. 1–16. https://doi.org/10.1109/CSF51468.2021.00036

Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. 2018. Checked C: Making C Safe by Extension. In *2018 IEEE Cybersecurity Development (SecDev)*. 53–60. https://doi.org/10.1109/SecDev.2018.00015

Trevor Elliott, Lee Pike, Simon Winwood, Patrick C. Hickey, James Bielman, Jamey Sharp, Eric L. Seidel, and John Launchbury. 2015. Guilt free ivory. *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (2015).

Xaver Fabian, Marco Patrignani, and Marco Guarnieri. 2022. Automatic Detection of Speculative Execution Combinations. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS 2022)*. ACM.

Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2007. A Type Discipline for Authorization Policies. *ACM Trans. Program. Lang. Syst.* 29, 5 (aug 2007), 25–es. https://doi.org/10.1145/1275497.1275500

Google. [n. d.]. Android Studio Webpage. https://developer.android.com/. Accessed: 2023-05-30.

Google. 2008. V8 Javascript Engine. https://v8.dev/blog/10-years. Accessed: 2023-05-30.

Andrew D. Gordon and Alan Jeffrey. 2003. Authenticity by Typing for Security Protocols. *J. Comput. Secur.* 11, 4 (jul 2003), 451–519.

Intel. 2023. *Intel®64 and IA-32 Architectures Software Developer Manual.* Acccessed: 2023-06-09.

Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference, General Track*.

Tina Jung, Fabian Ritter, and Sebastian Hack. 2021. PICO: A Presburger In-Bounds Check Optimization for Compiler-Based Memory Safety Instrumentations. *ACM Trans. Archit. Code Optim.* 18, 4, Article 45 (jul 2021), 27 pages. https://doi.org/10.1145/3460434

Andrew Kennedy. 2006. Securing the .NET programming model. *Theoretical Computer Science* 364, 3 (2006), 311–317. https://doi.org/10.1016/j.tcs.2006.08.014 Applied Semantics.

Sungkeun Kim, Farabi Mahmud, Jiayi Huang, Pritam Majumder, Chia che Tsai, Abdullah Muzahid, and Eun Jung Kim. 2023. WHISTLE: CPU Abstractions for Hardware and Software Memory Safety Invariants. *IEEE Trans. Comput.* 72 (2023), 811–825.

Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO '96*, Neal Koblitz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 104–113.

Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom. 2023. CryptOpt: Verified Compilation with Randomized Program Search for Cryptographic Primitives. *Proc. ACM Program. Lang.* 7, PLDI, Article 158 (jun 2023), 25 pages. https://doi.org/10.1145/3591272

P.A. Kulkarni, D.B. Whalley, G.S. Tyson, and J.W. Davidson. 2006. Exhaustive optimization phase order space exploration. In *International Symposium on Code Generation and Optimization (CGO'06)*. 13 pp.–318. https://doi.org/10.1109/CGO.2006.15

Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. 2013. Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-Based Security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 721–732. https://doi.org/10.1145/2508859.2516713

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. 2021. Cryptographic Capability Computing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 253–267. https://doi.org/10.1145/3466752.3480076

Liyi Li, Yiyun Liu, Deena Postol, Leonidas Lampropoulos, David Van Horn, and Michael Hicks. 2022. A Formal Model of Checked C. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*. 49–63. https://doi.org/10.1109/CSF54842.2022.9919657

Sergio Maffeis, Martín Abadi, Cédric Fournet, and Andy Gordon. 2008. Code-Carrying Authorization. In *13th European Symposium on Research in Computer Security, MÃ¡laga, Spain, October 6-8, 2008. Proceedings* (13th european symposium on research in computer security, málaga, spain, october 6-8, 2008. proceedings ed.), Vol. 5283. Springer Berlin Heidelberg, 563–579. https://www.microsoft.com/en-us/research/publication/code-carrying-authorization/

N. Manjikian and T.S. Abdelrahman. 1997. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems* 8, 2 (1997), 193–209. https://doi.org/10.1109/71.577265

D. McCullough. 1988. Noninterference and the Composability of Security Properties. In *2012 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, USA, 177. https://doi.org/10.1109/SECPRI.1988.8110

Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. 2023. MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code. *Proc. ACM Program. Lang.* 7, POPL, Article 15 (jan 2023), 30 pages. https://doi.org/10.1145/3571208

Microsoft. 2010a. CVE-2010-2557. Available from MITRE, CVE-ID CVE-2010-2557.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2557

Microsoft. 2010b. CVE-2011-0035. Available from MITRE, CVE-ID CVE-2011-0035.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0035

Microsoft. 2010c. CVE-2011-0036. Available from MITRE, CVE-ID CVE-2011-0036.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0036

Microsoft. 2015. CVE-2015-1770. Available from MITRE, CVE-ID CVE-2015-1770.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1770

Greg Morrisett, Amal Ahmed, and Matthew Fluet. 2005. L3: A Linear Language with Locations. In *Typed Lambda Calculi and Applications*, Paweł Urzyczyn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 293–307.

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for c. *SIGPLAN Not.* 44, 6 (jun 2009), 245–258. https://doi.org/10.1145/1543135.1542504

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. *SIGPLAN Not.* 45, 8 (jun 2010), 31–40. https://doi.org/10.1145/1837855.1806657

Myoung Jin Nam, Periklis Akritidis, and David J Greaves. 2019. FRAMER: A Tagged-Pointer Capability System with Memory Safety Applications. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 612–626. https://doi.org/10.1145/3359789.3359799

George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (may 2005), 477–526. https://doi.org/10.1145/1065887.1065892

Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation: A Survey of Fully Abstract Compilation and Related Work. *ACM Comput. Surv.* 51, 6, Article 125 (feb 2019), 36 pages. https://doi.org/10.1145/3280984

Marco Patrignani and Deepak Garg. 2021. Robustly Safe Compilation, an Efficient Form of Secure Compilation. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 1 (feb 2021), 41 pages. https://doi.org/10.1145/3436809

Marco Patrignani, Robert Künnemann, and Riad S. Wahby. 2022. Universal Composability is Robust Compilation. arXiv:1910.08634 [cs.PL]

Daniel Patterson and Amal Ahmed. 2019. The next 700 Compiler Correctness Theorems (Functional Pearl). *Proc. ACM Program. Lang.* 3, ICFP, Article 85 (jul 2019), 29 pages. https://doi.org/10.1145/3341689

Daniel Patterson and Amal J. Ahmed. 2017. Linking Types for Multi-Language Software: Have Your Cake and Eat It Too. *ArXiv* abs/1711.04559 (2017).

Gururaj Saileshwar, Rick Boivie, Tong Chen, Benjamin Segal, and Alper Buyuktosunoglu. 2022. HeapCheck: Low-Cost Hardware Support for Memory Safety. *ACM Trans. Archit. Code Optim.* 19, 1, Article 10 (jan 2022), 24 pages. https://doi.org/10.1145/3495152

Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2019. The High-Level Benefits of Low-Level Sandboxing. *Proc. ACM Program. Lang.* 4, POPL, Article 32 (dec 2019), 32 pages. https://doi.org/10.1145/3371100

Gabriel Scherer, Max New, Nick Rioux, and Amal Ahmed. 2018. FabULous Interoperability for ML and a Linear Language. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS) (FabOpen image in new windowous Interoperability for ML and a Linear Language, Vol. LNCS - Lecture Notes in Computer Science)*, Christel Baier and Ugo Dal Lago (Eds.). Springer, Thessaloniki, Greece. https://doi.org/10.1007/978-3-319-89366-2_8

Amogha Udupa Shankaranarayana, Gopal Raveendra Soori, Michael Ferdman, and Dongyoon Lee. 2023. TAILCHECK: A Lightweight Heap Overflow Detection Mechanism with Page Protection and Tagged Pointers.

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 89 (oct 2017), 26 pages. https://doi.org/10.1145/3133913

Stelios Tsampas, Andreas Nuyts, Dominique Devriese, and Frank Piessens. 2020. A categorical approach to secure compilation. *ArXiv* abs/2004.03557 (2020).

Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. 2019. Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code. *Proc. ACM Program. Lang.* 3, ICFP, Article 84 (jul 2019), 29 pages. https://doi.org/10.1145/3341688

VMWare. 2023. CVE-2023-20892. Available from MITRE, CVE-ID CVE-2023-20892.. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-20892

Son Tuan Vu, Albert Cohen, Arnaud De Grandmaison, Christophe Guillon, and Karine Heydemann. 2021. Reconciling optimization with secure compilation. *Proceedings of the ACM on Programming Languages* 5 (2021), 1 – 30.

Hiroshi Watanabe. 2002. Well-behaved Translations between Structural Operational Semantics. In *International Workshop on Coalgebraic Methods in Computer Science.*

Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem. *Proc. ACM Program. Lang.* 3, POPL, Article 77 (jan 2019), 29 pages. https://doi.org/10.1145/3290390

Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (apr 1991), 181–210. https://doi.org/10.1145/103135.103136

Torben Weis, Marian Waltereit, and Maximilian Uphoff. 2019. Fyr: a memory-safe and thread-safe systems programming language. *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (2019).

Richard West and Gary T. Wong. 2005. Cuckoo: a Language for Implementing Memory- and Thread-safe System Services. In *International Conference on Programming Languages and Compilers.*

Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Minneapolis, Minnesota, USA) *(ISCA '14)*. IEEE Press, 457–468.

Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. 2010. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (Beijing, China) *(ASIACCS '10)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/1755688.1755707

Jie Zhou, John Criswell, and Michael Hicks. 2023. Fat Pointers for Temporal Memory Safety of C. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 86 (apr 2023), 32 pages. https://doi.org/10.1145/3586038