Formal Approaches to Secure Compilation

A survey of fully abstract compilation and related work

MARCO PATRIGNANI*, Stanford University, USA and CISPA, Germany
AMAL AHMED, Northeastern University, USA
DAVE CLARKE, Uppsala University, Sweden

Secure compilation is a discipline aimed at developing compilers that preserve the security properties of the source programs they take as input in the target programs they produce as output. This discipline is broad in scope, targeting languages with a variety of features (including objects, higher-order functions, dynamic memory allocation, call/cc, concurrency) and employing a range of different techniques to ensure that source-level security is preserved at the target level. This paper provides a survey of the existing literature on formal approaches to secure compilation with a focus on those that prove fully abstract compilation, which has been the criterion adopted by much of the literature thus far. This paper then describes the formal techniques employed to prove secure compilation in existing work, introducing relevant terminology, and discussing the merits and limitations of each work. Finally, this paper discusses open challenges and possible directions for future work in secure compilation.

ACM Reference Format:
Marco Patrignani, Amal Ahmed, and Dave Clarke. 2018. Formal Approaches to Secure Compilation: A survey of fully abstract compilation and related work. *ACM Comput. Surv.* 1, 1, Article 1 (January 2018), 46 pages. https://doi.org/10.1145/3280984

This paper uses colours to make it easier for readers to follow the formal details.
Please view or print this paper in colour.

1 INTRODUCTION

Compilers are programs that transform code written in one language, called the *source* language, into code written in another language, called the *target* language. Many source languages are high level, so they provide powerful abstractions for the program to use, such as types and module systems. When source programs interoperate with each other, e.g., by calling each other's functions, they have to adhere to the available source-level abstractions.[1] Target languages, on the other hand, are typically low level and either have *different* abstractions or are devoid of any high-level abstraction. Thus, the abstractions provided by target languages rarely coincide with those provided by source languages.

*This work was conducted when the author was at KU Leuven, at MPI-SWS and at CISPA.
[1]Throughout this survey, when we refer to "programs," we typically mean some program fragments, components, or modules, *not whole programs.*

Authors' addresses: Marco Patrignani, Computer Science, Stanford University, Stanford, USA , CISPA, Saarbrücken, Germany, mp@cs.stanford.edu; Amal Ahmed, Northeastern University, Boston, MA, USA, amal@ccs.neu.edu; Dave Clarke, Uppsala University, Uppsala, Sweden, .

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 0 Association for Computing Machinery.
0360-0300/2018/1-ART1 $15.00
https://doi.org/10.1145/3280984

As an example, consider the Java code of Listing 1, which is translated into the C code of Listing 2—for the sake of brevity, both code snippets have been simplified to a minimum.

```
1 package Bank;
2
3 public class Account{
4   private int balance = 0;
5
6   public void deposit( int amount ) {
7     this.balance += amount;
8   }
9 }
```

Listing 1. Example of Java source code.

```
1 typedef struct account_t {
2   int balance = 0;
3   void ( *deposit ) ( struct Account*, int ) = deposit_f;
4 } Account;
5
6 void deposit_f( Account* a, int amount ) {
7   a->balance += amount;
8   return;
9 }
```

Listing 2. C code obtained from compiling the Java code of Listing 1.

When the Java code in Listing 1 interacts with other Java code, the latter cannot access the contents of `balance` since it is a private field. However, when the Java code is compiled into the C code in Listing 2 and then interacts with arbitrary C code, the latter can access the contents of `balance` by doing simple pointer arithmetic. Given a pointer to a C `Account` **`struct`**, an attacker can add the size (in words) of an **`int`** to it and read the contents of `balance`, effectively violating a confidentiality property that the source program had.

This violation occurs because there is a discrepancy between what abstractions the source language offers and what abstraction the target language has. This discrepancy is both inevitable and dangerous. The inevitability stems from the fact that source languages provide powerful abstractions whose goal is allowing a programmer to write better code. The danger stems from the fact that source-level abstractions can be used to enforce security properties, but target languages that do not preserve such security properties are vulnerable to attacks. Unfortunately, most target languages cannot preserve the abstractions of their source-level counterparts [1, 67].

In order to withstand the danger posed by exploitable target languages, secure compilation techniques can be adopted. Secure compilation is a discipline that studies compilers (or, more generally, compilation schemes) that preserve the security properties of source languages in their compiled, target-level counterparts. Secure compilation is concerned with the security of *partial programs* (also called *components* in this paper) since attackers are modelled as the environment such programs interact with. Partial programs are programs that do not implement all the functionality they require to operate. Instead, they are linked together with an environment (often also called a *context*) that provides the missing functionality in order to create a runnable whole program. An open environment is used to model possible attackers to the component, which is not possible when whole programs are considered.

Secure compilers use a variety of techniques to protect compiled components from attacker contexts. Once devised, such compilers must be proven to be secure, i.e., they must be proven to conform to some criterion that implies secure compilation. As we discuss later in this survey, a

variety of formal statements can capture when a compiler is secure. One such formal criterion that has been widely adopted for secure compilation is *compiler full abstraction* [1].

Informally, a compiler is fully abstract when it translates equivalent source-level components into equivalent target-level ones. Formally, a fully-abstract compiler *preserves* and *reflects* observational equivalence (usually contextual equivalence) between source and target programs. Reflection of observational equivalence means that the compiler outputs target-level components that behave as their source-level counterparts, this is generally a consequence of the compiler being correct. Preservation of observational equivalence implies that the source-level abstractions in the generated target-level output are not violated by a target-level client. For example, consider two instances $\mathbf{P_1}$ and $\mathbf{P_2}$ of the same component which contains two different values for some variable $\mathbf{x}$. Denote their compiled counterparts by $[\![\mathbf{P_1}]\!]$ and $[\![\mathbf{P_2}]\!]$. If the compilation scheme is fully abstract, and if $\mathbf{P_1}$ and $\mathbf{P_2}$ are equivalent, then $[\![\mathbf{P_1}]\!]$ and $[\![\mathbf{P_2}]\!]$ must also be. So, if the content of $[\![\mathbf{x}]\!]$ is *confidential* (for example, the value stored in $\mathbf{x}$ could be **`private`** and never communicated), no program interacting with $[\![\mathbf{P_1}]\!]$ or $[\![\mathbf{P_2}]\!]$ can observe it. Notice that a fully abstract compiler does not eliminate source-level security flaws. A fully abstract compiler is, in a sense, conservative, as it introduces no more vulnerabilities at the target-level than the ones already exploitable at the source-level.

One of the goals of this survey is to describe fully abstract compilation as well as other formal criteria for secure compilation. A second goal is to present work on compilers that are proven to conform to these formal criteria. Furthermore, since the work in this area uses different techniques to attain proofs of secure compilation, a final goal of this paper is to explain these proof techniques.

The rest of the paper is organised as follows. To understand threat models for secure compilation, Section 2 first discusses the kind of attacks that secure compilers need to defend against. To understand the security properties these attacks violate, and to connect their preservation with secure compilation criteria, Section 3 then describes how to formally express security properties using observational equivalence. To state which security guarantees a compiler provides, Section 4 continues by presenting different compiler properties and the kind of security guarantees they yield. Finally, Section 5 presents the survey of research on secure compilation, describing the published papers on this subject. Section 6 concludes this paper by presenting open challenges and future research directions.

Appendix A of this paper expands on the topic of Section 3, describing other widely adopted program equivalences. Additionally to show that a compiler upholds a property, a formal proof is required, so Appendix B also discusses the proof techniques used in secure compilation work.

This paper makes the following contributions:

- it presents the formal techniques adopted to reason about and formalise a secure compiler as well as to prove it secure,
- it surveys existing work on secure compilation, detailing methodology, contributions and limitations of each work;
- it highlights open problems and future research directions.

What This Survey is Not About

It is important to specify what the reader will *not* find in this paper. We have deliberately chosen to focus on papers that approach secure compilation from a *formal* angle. That said, there are many other papers that cover compiler implementations tailored to addressing a specific attack or a specific exploit [69, 120], but which do not take a formal approach. One example is the work on Code Pointer Integrity (CPI), which aims to compile C/C++ code relying on target-level support for isolated memory to prevent out-of-bound access of sensitive pointers [69]. Such work, which does not provide a formal statement of correctness, is not covered in this survey.

Another category of work that is not covered in this survey focuses on target-level enforcement of specific security policies (but without establishing any connection to source-level properties). For instance, Inlined Reference Monitors (IRM) are aimed at hardening target code with runtime checks that enforce a specific security policy [31, 44–46]. Another well known example is Control-Flow Integrity (CFI), which involves a rewriting of target code to enforce that no jumps can be made outside of the locations specified in the target control-flow graph [3, 4]. CFI can be used to implement efficient IRM, Software Fault Isolation (SFI) [132] and other properties [4]. Note that CFI, IRM, and other analogous methods can be incorporated into a secure compiler to ensure that it enforces the particular security properties these methods guarantee. Many of these papers are also often mainly concerned with implementing the specific countermeasure they describe instead of providing a correctness criterion that the countermeasure enforces. Hence, such papers are not discussed in this survey.

By excluding these papers from the survey we do *not* want to diminish their research contributions. On the contrary, we believe that starting from the few pointers provided in this section, the interested reader can find out more about these categories of research and use those techniques when building secure compilers. However, the goal of the survey is to cover a different broad family of research, one that focuses on the foundational principles and reasoning tools for secure compilation.

A note on colours. When dealing with compilers it is important to clearly distinguish between their source and target languages. We use a **blue, bold** font for source elements and a pink, sans-serif one for target elements, but use *black* for notions common to both languages to avoid repeating them in two colours.

2 SOURCE-LEVEL ABSTRACTIONS AND TARGET-LEVEL ATTACKS

This section explains the security relevance of secure compilation and the threat models considered when developing a secure compiler.

The secure compilation literature contains several examples of source-level security properties that can be violated by target-level attackers. These violations define part of the threat model that work on secure compilation needs to address since they define possible attacks on compiled code. Attackers are often modelled as target-level programs as this captures their ability to operate at that level, injecting or linking arbitrary target code, which means they can do anything that target-level code can. This power often stems from an exploitable security vulnerability that the system is assumed have, though the details of the vulnerability are often unspecified.

The capabilities of an attacker vary depending on the target language considered. For example, when using untyped assembly code, the attacker can access and alter the contents of the whole address space (she is generally assumed to operate at Kernel level to capture the most dangerous threat). When using simply-typed λ-calculus terms, the attacker must supply well-typed code that can only interact with existing code via functions calls. What the attacker cannot do in any case is to employ techniques that are not available in the target language. So, if the target language is strongly typed, an attacker cannot violate the well-typedness. Additionally, side channels (e.g., timing) are often out of the scope of compiler security since they are often not expressible in the target language. We will come back to protection against side channels in Section 6.

The remaining details of the threat model, i.e., the intended security properties and the definition of the system under attack, vary in each paper on secure compilation so we omit them here.

To illustrate what attackers can do with compiled code, this section presents a series of examples of the most relevant threats that a secure compiler needs to mitigate. These examples are taken from the surveyed work; their syntax is massaged for uniformity of presentation.

Example 2.1 (Confidentiality of values [9]). Consider the Java-like code below, where function `setSecret` sets the field `secret` to 1 and then returns 0.

```
private secret : Int = 0;

public setSecret() : Int {
  secret = 1;
  return 0;
}
```

The field `secret` is used to store confidential data (it is **`private`**) and it is inaccessible from other source-level code, so a target-level attacker should not be able to retrieve its value. If this code gets compiled to a language where memory locations are identified by natural numbers (e.g., an untyped assembly language or $\lambda\mu$-hashref [61]), then the address where `secret` is stored can be read by attackers. By dereferencing the number associated with the location of `secret`, attackers can violate the intended confidentiality property of the code.

Example 2.2 (Integrity of values [9]). Analogous to Example 2.1, function `proxy` below sets the variable `secret` to 1, and then calls the function `callback`, that was passed in as a parameter.

```
public proxy( callback : Unit → Unit ) : Int {
  var secret = 1;
  callback();
  return 0;
}
```

The variable `secret` is inaccessible to the code in the `callback` function at the source level. However, if this code is compiled to a target language that can manipulate the call stack, it can access the `secret` variable and change its value. Similarly, malicious target-level code can manipulate the return address stored on the stack, altering the expected flow of computation.

Example 2.3 (Finite memory size [61]). When dealing with memory, its size can also affect the behaviour of a component, however the memory size is often not a concern of source languages. Consider a source language with a dynamic memory allocation operation **`new`**. Function `kernel` below allocates `n` new `Objects`, calls a function `callback` and executes *security-relevant code* before returning 0.

```
public kernel( n : Int, callback : Unit →Unit ) : Int {
  for (i = 0 to n){
    new Object();
  }
  callback();
  // security-relevant code
  return 0;
}
```

At the source level, the *security-relevant code* will always be executed. However, if this code is compiled to a language that limits its memory to contain only `n` `Objects`, code execution can be disrupted during the `callback`. If the `callback` allocates another object, the *security-relevant code* will not be executed.

Example 2.4 (Deterministic memory allocation [61]). Dynamic memory allocation is a feature that leads to complications, as illustrated in the code below. In this example, the code allocates two `Objects`, and then returns the first one.

```
public newObjects( ) : Object {
  var x = new Object();
  var y = new Object();
```

```
4   return x;
5 }
```

While at source level `Object y` is inaccessible, this is not true in certain target languages. A target-level attacker that knows the memory allocation order can predict where an object will be allocated and influence its memory contents. She can tamper with `y` by guessing or by calculating its address.

Example 2.5 (Well-typedness of programs [99]). When the source language is strongly typed and the target one is not (e.g., untyped assembly as the target language), complications can arise, as illustrated in the code below. In this example, the code provides an implementation of a `Pair` class (with the expected `first` and `second` fields) and a method to access the first element `getFirst()`. The code also provides an implementation of a `Secret` class whose field `secret` is inaccessible.

```
1  class Pair {
2   private first, second : Obj = null;
3   public getFirst(): Obj {
4    return this.first;
5   }
6  }
7  class Secret {
8   private secret : Int = 0;
9  }
10 object o : Secret
```

The value of `secret` cannot be leaked at the source level, but when compiled to untyped assemby code, the compiled counterparts of this code can leak its `secret` value. An attacker can perform a call to method `getFirst()` with current object `o`; this will return the `secret` field, since fields are accessed by offset in assembly.

Example 2.6 (Well-bracketed control flow [102]). Consider untyped assembly as the target language and the following three components being compiled.

```
1  class Main {
2   public static main(): Obj { Proxy.proxy(); }
3  }
4
5  class Proxy {
6   public static proxy(): Obj { Mod.callFun(); }
7
8   public static fun():Obj { ... }
9  }
10
11 class static Mod {
12  public callFun():Obj { Proxy.fun(); }
13 }
```

The sequence of calls is the following: `Main`→`Proxy`→`Mod`→`Proxy`.

In all assembly languages, calls and returns are jumps to addresses in memory. So, the second time `Proxy` is called, it can return to `Main`, as it learnt the address to return there when `Main` called it. However, that instruction bypasses the rest of the call stack and particularly `Mod` in a way that is not possible in the code above, where control flow follows a well-bracketed sequence of calls/returns. If `Main` and `Mod` had a shared invariant, it can thus be violated in this way.

Example 2.7 (Information flow [22]). Another form of information leakage is information flow. Information flow is generally concerned with a public output (called *low* security) being affected by secure input (called *high* security). The code of Listing 3 presents an example of direct information flow: a *high*-security value, indicated with the subscript h is stored in a *low*-security memory cell, indicated with the subscript l.

```
1 public storeValue( value : Int_h ) : Int_l {
2   var location : Loc_l = value;
3   return 0;
4 }
```

Listing 3. Example code with direct information flow.

A more subtle form of information flow comes in the form of *indirect* information flow, as presented in Listing 4.

```
1 public isZero( value : Int_h ) : Int_l {
2   if ( value == 0 ) {
3     return 1
4   }
5   return 0
6 }
```

Listing 4. Example code with indirect information flow.

In this case assume the attacker does not have direct access to the memory, yet she can detect whether `value` is 0 or not by observing the output of the function. This is called indirect information flow because the presence of a high-security value in the guard of a branching construct influences the output of the branch.

A target language that does not prevent information flow cannot withstand these leaks.

Example 2.8 (Continuation manipulation for declassification leak). Consider a target language with call with current continuation, also known as `call/cc`. This is a Scheme construct that lifts the continuation of a function (i.e., what happens after the body of the said function is executed) as a first-class language citizen and thus lets a programmer manipulate it.

A declassification policy is a common security policy that states that something, e.g., a key, is secure only until a certain event or period in time, and afterwards it is publicly accessible.

Consider the following source program. It contains a key used to perform some secure communication over the network (`network_send( encrypt( secret, k ) )`. That program engages in three interactions, counted using a private variable `x`, before releasing the no-longer-secret key.

```
1   public key : Key = null;
2   private k : Key = secret;
3   private x : Int = 0;
4
5   public three-times-protocol( ) : Object {
6     continue();
7     network_send( encrypt( secret, k ) );
8     x++;
9     continue();
10    network_send( encrypt( secret, k ) );
11    x++;
12    continue();
13    network_send( encrypt( secret, k ) );
14    x++;
15    // the key is declassified
16  }
17  private continue( ) : Object {
18    if( x > 2 ){
19      key = k;
20    }
21    callback();
22  }
```

Listing 5. Example code with a declassification violation.

A target-level attacker implementing `callback()` can capture the continuation

```
c =x++;continue();network_send( encrypt( secret, k ) );x++;continue();//...
```

that is passed after the first invocation of `continue()`. She can use it just to increment `x` up to three and throw away the continuation of the continuation, i.e.,

```
continue()network_send( encrypt( secret, k ) );;x++;//...
```

Then, she can replay the initial continuation *c* and call `continue()` for the second time. However, the code will operate as this is the third time by looking at `x`, and the attacker can thus receive the key `k` over the public variable `key` before time. This is only possible in the target code, where continuations can be manipulated, and not in the source program.

Example 2.9 (Network-based threats [7]). Networked components are subject to a large number of attacks which are generally focussed on the information being communicated across the network. The Dolev-Yao attacker model is often used to model network-based attacks [41]. A Dolev-Yao attacker can eavesdrop, intercept, and synthesise any message being communicated on a network. Intuitively, it is seen as an arbitrary process E running alongside of other processes A and B, which are also executing in parallel. E is given access to the communication medium shared between A and B, the same way a computer can sniff all WiFi packets in an open network. By inspecting all communicated messages, E can trivially see any unencrypted confidential data being communicated between A and B. Moreover, E can impersonate either A or B and it can replay previously sent messages, which could cause faulty behaviour in any other process. Additionally, E can mount forward secrecy attacks by buffering communicated messages and then reconstructing their meaning later on, when it receives some missing piece of information with which it can understand the buffered messages. Sometimes, only one party of the communication is trusted, so instead of having A communicating with B, A may be communicating with E. In this case, A does not only need to protect the communication, it also needs to monitor E, which could stray from the communication protocol and enact fault-inducing behaviour.

The literature on network-based attacks is vast; a more precise analysis is presented in Section 5.2.1 and in the related work of the surveyed work.

3 EXPRESSING SECURITY PROPERTIES AS PROGRAM EQUIVALENCES

In order to withstand the attacks described so far, as well as other security breaches, it is important to formally specify the security properties of components. In many of the work surveyed in this paper, program equivalence techniques are often used to express these concepts.[2] Contextual equivalence (Section 3.1) is the most coarse-grained program equivalence; its relevance for security has been discussed in many papers [9, 14, 61, 99, 101] and the formal statement of fully abstract compilation relies on it. This section defines contextual equivalence and shows how it can be used to formally capture the security concerns of the example code presented in Section 2. Other equivalences that can be used in place of contextual equivalence are presented in Appendix A.

As discussed in Section 1, secure compilation is only relevant when compiling components. Thus, the program equivalence techniques presented in this paper describe the behaviour of components or partial programs. There are other techniques for reasoning about equivalence of whole programs, which we do not present here as they are out of scope for secure compilation.

[2] Other techniques such as properties [116] and hyperproperties [32] exist, but we do not focus on them as the surveyed work does not rely on them.

3.1 Contextual Equivalence

Contextual equivalence (also known as observational equivalence) provides a notion of observation of the behaviour of a component (generally divergence) and states when two components exhibit the same observable behaviour. The notion of contextual equivalence (Definition 3.3 below) relies on the definition of context and of divergence (Definition 3.1 and 3.2). This section also discusses the pros and cons of contextual equivalence (Section 3.1.1).

Definition 3.1 (Context). A context $\mathbb{C}$ is a program with a hole (denoted by $[\cdot]$), which can be filled by a component P, generating a new whole program: $\mathbb{C}[P]$.

Contexts can be used to model the external code that can interact with a specific piece of software (in this case, the hole-filling component P). Based on the language of P, contexts can assume a variety of forms. For example, if P is the λ-calculus expression $\lambda x.(xx)$, a context is another λ-calculus expression with a hole, such as $(\lambda y.(yy))\ [\cdot]$ or $[\cdot]\ (\lambda y.(yy))$. In this case, if P is plugged in the hole of either context, the resulting whole program is $(\lambda y.(yy))\ (\lambda x.(xx))$, which is commonly known as Ω, the diverging term. Analogously, when P is the Java code of Listing 1 contexts are other Java programs which refer to (and use) the classes P defines, such as the Java code in Listing 6.

```
1 package main;
2 import Bank.Account;
3
4 public class Main{
5   public static void main( String [] args ){
6     Account acc = new Account();
7   }
8 }
```

Listing 6. Example of a Java context interacting with the code of Listing 1.

From a semantics perspective, plugging a component in a context makes the program whole, so its behaviour can be observed via its operational semantics. A different way to close the term would be via system-level semantics [55]. In this kind of approach, the context (called the opponent) is not constrained by the syntax of a language as is the case with contextual equivalence, so it can model a powerful attacker to the code. Since this approach has not been used in secure compilation work we do not discuss it further.

Definition 3.2 (Divergence). A component P diverges if it performs an unbound number of reduction steps. We denote that P diverges as $P\Uparrow$.

Definition 3.3 (Contextual equivalence [110]). Two components P_1 and P_2 are contextually equivalent if they are interchangeable in any context without affecting the observable behaviour of the program: $P_1 \simeq_{ctx} P_2 \triangleq \forall\mathbb{C}.\ \mathbb{C}[P_1]\Uparrow \iff \mathbb{C}[P_2]\Uparrow$.

For strictly-terminating languages, the requirement that both programs diverge or both terminate must be replaced with the requirement that both terminate yielding equal values of ground type (e.g., integers or bools).

Using contextual equivalence, only what can be observed by the context is of any relevance; this changes from language to language, as different languages have different functionality. Contexts can model malicious attackers that interoperate with the secure software (the hole-filling component P), possibly mounting attacks such as those described in Section 2.

Contextual equivalence can be used to model security properties of source code, as described by Example 3.4 below, for various examples from Section 2.

Example 3.4 (Security properties via contextual equivalence). The code snippet of Example 2.1 described confidentiality properties. That code is presented alongside a new snippet in Figure 1

(relevant differences in side-by-side snippets are coloured in red). If the two snippets are contextually equivalent, then the value of `secret` is confidential to the code. In fact, the two snippets assign different values to `secret`, so if they are contextually equivalent, then `secret` must not be discernible by external code.

```
1 private secret : Int = 0;
2
3 public setSecret( ) : Int {
4   secret = 0;
5   return 0;
6 }
```

```
1 private secret : Int = 0;
2
3 public setSecret( ) : Int {
4   secret = 1;
5   return 0;
6 }
```

Fig. 1. These code snippets express confidentiality properties.

The code snippet of Example 2.2 described integrity properties. Figure 2 presents that code alongside other code that checks whether the variable `secret` has been modified during the `callback`. Since `secret` is allocated on the stack, returning from function `proxy` will erase it; if it were allocated globally, its integrity would need to be checked at each lookup of its value. If these code snippets are contextually equivalent, then `secret` cannot be modified during the `callback`.

```
1 public proxy( callback : Unit → Unit )
      : Int {
2   var secret = 0;
3   callback();
4   if ( secret == 0 ) {
5     return 0;
6   }
7   return 1;
8 }
```

```
1 public proxy( callback : Unit → Unit )
      : Int {
2   var secret = 0;
3   callback();
4
5   return 0;
6
7
8 }
```

Fig. 2. These code snippets express integrity properties.

The code snippet of Example 2.3 described memory size properties. Figure 3 presents that code alongside other code that does not allocate `n` new `Objects`. In Example 2.3, external code could disrupt the execution flow by overflowing the memory. If these code snippets are contextually equivalent, then the memory size does not affect the computation.

```
1 public kernel( n : Int, callback : Unit
      → Unit ) : Int {
2   for (Int i = 0; i < n; i++){
3     new Object();
4   }
5   callback();
6   // security-relevant code
7   return 0;
8 }
```

```
1 public kernel( n : Int, callback : Unit
      → Unit ) : Int {
2
3
4
5   callback();
6   // security-relevant code
7   return 0;
8 }
```

Fig. 3. These code snippets express unbounded memory size properties.

The code snippet of Example 2.4 describes memory allocation properties. Figure 4 presents that code alongside other code that returns the second allocated object instead of the first one. If these code snippets are contextually equivalent, then the memory allocation order is invisible to an attacker, as she is unable to distinguish between objects based on their allocation order.

```
1 public newObjects( ) : Object {
2   var x = new Object();
3   var y = new Object();
4   return x;
5 }
```

```
1 public newObjects( ) : Object {
2   var x = new Object();
3   var y = new Object();
4   return y;
5 }
```

Fig. 4. These code snippets express memory allocation properties.

3.1.1 Discussion. Contextual equivalence shows its limitations both in the attacks it can express and complexity it introduces in proofs. Timing attacks or, more generally, side-channels attacks cannot be expressed with contextual equivalence. Thus, these attacks are generally disregarded by secure compilation techniques; they can be countered using orthogonal protection mechanisms which are beyond the scope of this survey. This interesting future work is discussed in Section 6.

Moreover, although other definitions of contextual equivalence exist (for example, Curien [34] uses reduction to the same value instead of divergence), no alternative formulation drops the universal quantification on contexts. Reasoning (and proving properties) about contexts is notoriously complex [18, 52, 61, 99]. To compensate for this difficulty, different forms of equivalence can be used, e.g., trace semantics [62, 64, 100, 102], weak bisimulation [113], applicative bisimilarity [12] and logical relations [16], but only if they are proved to be *as precise* as contextual equivalence. These equivalences are described in Appendix A. Given an equivalence $\approx$, if $\approx$ is correct and complete w.r.t. $\simeq_{ctx}$, then it can be used in place of $\simeq_{ctx}$.Correctness in this case captures that $\approx$ does not distinguish between programs that are equivalent for $\simeq_{ctx}$, completeness on the other hand captures that all programs that are equivalent for $\simeq_{ctx}$ are still equivalent for $\approx$ [34, 85, 110]. [3] In fact, many proofs of compiler full abstraction rely on a program equivalence that is as precise as contextual equivalence to be carried out.

Having presented how to formalise security properties as program equivalence, we next discuss how to define compiler properties that preserve such security properties.

4 A SPECTRUM OF COMPILER PROPERTIES

This section presents the three main kinds of compiler properties that have been frequently studied in the literature. The three properties are: (1) that compilation is type-preserving (Section 4.1), which establishes that a compiler from a typed source language to a typed target transforms well-typed source terms into well-typed target terms; (2) that compilation is semantics-preserving (informally known as compiler correctness, Section 4.2), which establishes that a compiler generates a target program with the same observable behaviour as the source program; and (3) that compilation is equivalence-preserving (Section 4.3), which establishes that the compiler translates programs that are equivalent in the source language—e.g., contextually equivalent or indistinguishable to an observer at a given security level—into programs that are similarly equivalent in the target language. Each of these properties is pertinent to secure compilation. Equivalence-preservation is the most obviously relevant as it is a requirement for fully abstract compilation. Semantics-preservation is important because it would be a dubious achievement to build a secure compiler that is not correct. Finally, type-preservation has been employed as a means of enforcing full abstraction. It has also been used in the absence of a proof of full abstraction to offer confidence that—under some assumptions which we discuss later—certain security properties hold at the target level.

Before we dive into details, we define what we mean by a compiler in the rest of the paper.

Definition 4.1 (Compiler). A compiler $[\![\cdot]\!]^{S}_{T}$ is a function from source S to target T components.

[3] In this case, $\approx$ is said to be fully-abstract, but we drop this terminology to avoid confusion with fully-abstract compilation.

Definition 4.2 (Modular compiler). A compiler is modular if linking a set of source components and then compiling the resulting program is equivalent to individually compiling each of those source components and then linking the resulting target components—i.e., if $[\![C_1 + \cdots + C_n]\!]^S_T = [\![C_1]\!]^S_T + \cdots + [\![C_n]\!]^S_T$. where we write + to denote linking.

4.1 Type-Preserving Compilation

A compiler that is type preserving translates well-typed source programs into well-typed target programs. The seminal work by Tarditi et al. [127] and Morrisett et al. [90] showed how to design typed intermediate languages (TILs) and typed assembly languages (TALs) so that source-language types can be translated into target-language types such that target-level type checking ensures type and memory safety. Let Γ denote a source-level typing environment and let the judgment $\Gamma \vdash P$ mean that P is well-typed in Γ. (An alternative formulation could be $\Gamma \vdash P : \tau$, which also specifies that P has type τ). Formally, type-preserving compilation can be expressed as $\Gamma \vdash P \Rightarrow [\![\Gamma]\!]^S_T \vdash [\![P]\!]^S_T$. This statement would, of course, have to be tuned to the specific typing judgements of the source and target languages under consideration.[4]

Morrisett et al. [90] showed how to design a typed assembly language (TAL) that extends an idealized RISC-like assembly with a type system that be can used to ascribe types to basic blocks, to data stored in register files, and to closures and tuples on the heap. They show how to compile a variant of System F with integers, products, and term-level recursion to TAL. The compilation pipeline consists of five type-preserving passes: conversion to continuation-passing style (CPS), which makes the control-flow explicit; closure conversion, which transforms functions into a closure that pairs closed functions with their environment, using an existential type to hide the type of the environment from clients of the closure; hoisting of closures to top-level; allocation of closures and tuples on the heap; and code generation to TAL.

Since the late 1990s, there have been numerous results on type-preserving compilation, showing how to extend the original idea to compilers for object-oriented languages [30, 75] or security-typed languages [22, 23, 84, 139].

An important point to note is that type-preserving compilers do not *necessarily* guarantee preservation of source-level security properties. For instance, a type-preserving compiler could compile an object-oriented language with private fields into a un(i)typed language. The translation is clearly type preserving but not security preserving since target-level clients of the object (attackers) can read the contents of private fields, leading to a security violation (Example 2.5). To ensure data stored in private fields is kept hidden from target-level clients, a type-preserving compiler can use a more clever type translation. For instance, League et al. [75] present a type translation that makes use of target-level type abstraction—specifically, existential types, which can be used to guarantee information hiding—to ensure that data stored in private fields of source objects remains hidden from target-level clients of the object. Thus, determining whether type-preserving compilers preserve security properties requires inspecting the type translation.

League *et al.* [75] and Chen *et al.* [30] present type-preserving compilers that also preserve security guarantees. These are secure compilers, so we discuss them in Section 5.1.

4.1.1 Security-Type-Preserving Compilation. Security-type-preserving compilation is a form of type-preserving compilation with security relevance. Security-typed languages provide a type system for information-flow control that allows programmers to label data with a security levels drawn from a security lattice. In such languages, type checking ensures that high-security data never influences low-security outputs, a property formally known as *noninterference*. Starting with

[4] Technically speaking, the target language does not have to be statically typed: any compiler that translates to a dynamically typed (un(i)typed) target language is type preserving.

the seminal work of Volpano *et al.* [131], security type systems have been developed in a variety of forms (e.g., [130, 140**?**]) and proved to satisfy noninterference. Formally, noninterference states that if a program P with an environment Γ that tracks the types and security labels for each free variable in P (cfr Example 2.7), running P with identical low-security values but unrelated high-security values yields the same low-security outputs. Noninterference of P is denoted with $\mathsf{NI}_\Gamma(P)$. For well-typedness to imply noninterference, the following must hold: $\Gamma \vdash P \Rightarrow \mathsf{NI}(P)$.

We say a compiler is security-type preserving when it translates well-typed source components into well-typed target components *and* well-typedness implies noninterference at both the source and the target level [22, 23]. Thus, security-type-preserving compilation must satisfy the following properties in order to imply secure compilation.

(1) The security properties of interest in the source language are enforced by means of a type system for information-flow control.
(2) The target language must be equipped with a security type system as well.
(3) Security types should be preserved: a target variable must be assigned the same security level as its source-level counterpart.
(4) The compiler must be correct.

The first property has been described above.

The second property requires security typed target languages. Several researchers have shown how low-level languages, such as assembly, can be equipped with a security type system as described in the previous section. An important requirement that this induces is that attackers written in these languages must also be well typed, and types are used to ensure that compiled code may only be linked with well behaved attackers that cannot compromise security.

The third property requires that the compiler preserves the security labels of variables. For instance, if a compiler translates a high-security source variable into a low-security target variable, then the contents of that variable would become visible to low-security observers at the target level, allowing security violations. An implication of this requirement is that both languages must have the same security lattice, but it would also suffice for there to be an order-preserving one-to-one embedding of the source lattice in the target lattice.

More formally, if we denote a target-level context as Δ and a target-level program as M. a security-type-preserving compiler must satisfy the following requirements:

(1) $\Gamma \vdash \mathbf{P} \Rightarrow [\![\Gamma]\!]^{\mathsf{S}}_{\mathsf{T}} \vdash [\![\mathbf{P}]\!]^{\mathsf{S}}_{\mathsf{T}}$,
(2) $\Gamma \vdash \mathbf{P} \Rightarrow \mathsf{NI}_\Gamma(\mathbf{P})$,
(3) $\Delta \vdash \mathsf{M} \Rightarrow \mathsf{NI}_\Delta(\mathsf{M})$.

On the positive side, proving a compiler to be security type-preserving does not require any reasoning about contextual equivalence. On the negative side, not all target languages enjoy a type system (e.g., untyped assembly), and those that do, do not necessarily enjoy one that is powerful enough to imply noninterference.

As discussed above, a type-preserving compiler is not necessarily a secure one. Some source language enforce security properties by means other than a noninterference type system, e.g., by using private fields. The strength of security-type-preserving compilation is demonstrating that noninterference is preserved. The additional result stating that well-typed source and target programs are noninterfering, when combined with manually inspecting the type translation to ensure that it preserves security labels of data, makes such a compilation scheme secure. In other words, the fact that security is preserved is taken care of by the target-level noninterference theorem, whose applicability is granted by the type-preservation result of the compiler.

4.2 Semantics-Preserving Compilation

There is a significant body of work on proving compiler correctness—that is, establishing that compilation preserves the semantics or behaviour of source programs. In this paper, we use the term semantics-preserving compilation and correct compilation interchangeably. Most compiler-verification work over the last decade has focused on *whole-program* compiler correctness, which assumes that compiled code will not be linked with any other code. However, for the purpose of this survey, we are primarily interested in establishing correct compilation of *components*, since secure compilation is most interesting in the *presence of target-level linking* since: (1) such linking scenarios arise in nearly every realistic software system, and (2) secure compilation aims to ensure that compiled components are protected from the target-level attacker code that these compiled components might be linked with. Several recent compiler-verification efforts have tackled *compositional* compiler correctness, which formally takes target-level linking (composition) into account. However, as we discuss below, all of these efforts specify compositional compiler correctness—the theorem to be proved—in different ways, leading to different benefits and drawbacks. There is currently no consensus on how to formally specify correct compilation of components.

4.2.1 Whole-Program Compiler Correctness. A compiler is correct if it produces target-level programs that behave the same as their source-level counterparts. We state this property as follows.

Definition 4.3 (Compiler correctness). A whole-program compiler is correct if all its source observables (e.g., reads and prints, indicated with Behav(·)) are the same as its target observables:

$$\mathbf{P} \text{ is whole and } \mathsf{Behav}(\mathbf{P}) = \mathsf{Behav}(\llbracket \mathbf{P} \rrbracket^{\mathsf{S}}_{\mathsf{T}})$$

For the sake of simplicity this definition assumes that the source language is deterministic and that source and target observables come from the same alphabet.[5] For a non-deterministic source language, we wish to show that the compiled target version *refines* the source—that is, replace = with $\supseteq$ since we wish to show that *every* target behavior is a *possible* source behavior. Also, if we use different alphabets for observables in source and target, we must replace the equivalence = or refinement $\supseteq$ with a cross-language relation between observables.

Compilers are complex software artefacts that frequently contain bugs due to which the compiler might fail to preserve the semantics (behaviour) of source programs. One way to ensure that a compiler is correct is to verify it [76, 77]. A verified compiler is one that has been proved to be semantics-preserving with the aid of a proof assistant such as Coq, which allows bug-free extraction of the compiler implementation, erasing proofs in the verified artefact.

The CompCert project is the most well-known effort to provide a verified compiler; CompCert is a Coq-verified multi-pass compiler from a considerable subset of C to PowerPC, ARM and x86 assembly [76, 77]. Other work has followed and extended the CompCert approach to verify compilers for multi-threaded languages [78], just-in-time compilation [91], C with relaxed memory concurrency [117], high-level functional languages [68], and much more.

4.2.2 Compositional Compiler Correctness. While the results above apply only to compilation of whole programs, there is also recent work on various forms of *compositional compiler correctness* that guarantees correct compilation of *components*. Unlike whole-program compiler correctness, compositional compiler correctness must account for the possibility of linking compiled code with other target-level code (or attackers). Different results in this area impose different restrictions on what code they may be linked with—intuitively, these restrictions correspond to assumptions about the attacker model.

[5] The interested reader is referred to Leroy [77] for a discussion on lifting the determinism hypothesis.

At a high level, all compositional compiler correctness results must find some way to formally specify when a source component $\mathbf{P}$ should be considered equivalent to a target component P'—which we write as $\mathbf{P} \approx \mathsf{P}'$—and then prove that $\mathbf{P} \approx [\![\mathbf{P}]\!]^{\mathsf{S}}_{\mathsf{T}}$. Recent work formalizes source-target equivalence in remarkably different ways.

One way to specify source-target equivalence is to use a *cross-language logical relation* that relates source and target programs. Benton and Hur [24] were the first to put this technique to use to verify a nontrivial compiler from a call-by-value λ-calculus to an SECD machine, and later for System F with recursion to the same target language [25]. Hur and Dreyer extended the approach to verify a compiler from ML to an idealized assembly [60]. Both results show that if a source component $\mathbf{P}$ compiles to a target component $[\![\mathbf{P}]\!]^{\mathsf{S}}_{\mathsf{T}}$ then $\mathbf{P}$ is logically related to $[\![\mathbf{P}]\!]^{\mathsf{S}}_{\mathsf{T}}$ by the cross-language relation. This approach does not scale to modular verification of *multi-pass* compilers. The latter would require proving transitivity of cross-language logical relations—i.e., that a relation between source and intermediate-language components and another between intermediate and target-language components compose to imply source-target relatedness—which is open problem for realistic languages. Neis et al. [93] specify source-target equivalence using a *parametric inter-language simulation* (PILS), a variation on cross-language logical relations that can be proved transitive. They show how to verify a multi-pass compiler from an ML-like source to an idealized assembly. Both the cross-language logical relation and the PILS approaches only supports linking with target code shown to be related to some code in the *same* source language—that is, they assume that target-level attackers are no more powerful than source attackers.

Perconti and Ahmed [106] specify source-target equivalence by first defining a multi-language [80] that embeds the source and target language of the compiler and then using multi-language contextual equivalence to specify source-target equivalence. They prove for a type-preserving compiler that if a source component $\mathbf{P}$ compiles to target component $[\![\mathbf{P}]\!]^{\mathsf{S}}_{\mathsf{T}}$, then $\mathbf{P}$ should be contextually equivalent (in the multi-language) to $\mathcal{ST}([\![\mathbf{P}]\!]^{\mathsf{S}}_{\mathsf{T}})$ (where the boundary $\mathcal{ST}$ is used to sensibly embed a target component in a source context). They give a two-pass verified compiler for System F with existential and recursive types (language F) that performs closure conversion (to an intermediate language C) and heap allocation (to a language A), and proved the correctness of each pass using contextual equivalence for the multi-language FCA. Stewart et al. [121] specify source-target equivalence using a *structured simulation* that captures rely-guarantee invariants needed across all passes of the CompCert C compiler. They give an *interaction semantics*, which provides an abstract specification of interoperability between source and target components, which allows linking with any target component that respects CompCert's memory model (which is uniform across all intermediate languages of the CompCert compiler). It is unclear how to extend this approach to compilers whose source and target languages use different memory models (e.g., ML and assembly). Both the multi-language and the interaction-semantics approach can accommodate linking with target-level attackers that are more powerful than any source-level attacker.

Finally, Kang et al. [66] prove correctness of a modified version of CompCert, dubbed SepCompCert, which support separate compilation. That is, it restricts linking to only those components produced by the same compiler therefore assuming a more limited attacker than all other results.

Each of these projects use a different formal statement for compositional compiler correctness. While there is work underway on a unified statement of compositional compiler correctness, details of this subject are beyond the scope of the current paper, so we refer the reader to the work of Patterson and Ahmed [105]. Nonetheless, all of the above formal statements of compositional compiler correctness should imply the following property—a restricted form of *equivalence reflection*—as a corollary, when both programs are deterministic or have the same kind of nondeterminism (e.g., they both support internal choice like process calculi).

Property 1 (Corollary of correct compositional compilation).

$$\forall \mathbf{P_1}, \mathbf{P_2} \in \mathbf{S}.\, \mathbf{P_1} \simeq_{ctx} \mathbf{P_2} \Longleftarrow [\![\mathbf{P_1}]\!]^{\mathsf{S}}_{\mathsf{T}} \overset{w\mathsf{ST}}{\simeq}_{ctx} [\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}}$$

Intuitively, this holds from compiler correctness and the fact that the compiler can be applied both to a program and to a context.

The above property uses a form of equivalence at the target level ($\overset{w\mathsf{ST}}{\simeq}_{ctx}$) that is called "well-behaved" contextual equivalence, which is defined as follows.

Definition 4.4 (Well-behaved contextual equivalence).

$$\mathsf{P_1} \overset{w\mathsf{ST}}{\simeq}_{ctx} \mathsf{P_2} \triangleq \forall \mathbb{C}_\mathsf{T} \in \mathsf{T}.(\exists \mathbb{C}_\mathbf{S} \in \mathbf{S}.\mathbb{C}_\mathbf{S} \approx \mathbb{C}_\mathsf{T}) \Rightarrow (\mathbb{C}_\mathsf{T}[\mathsf{P_1}]\Uparrow \iff \mathbb{C}_\mathsf{T}[\mathsf{P_2}]\Uparrow)$$

Well-behaved contextual equivalence is analogous to contextual equivalence except that, instead of arbitrary target-level contexts, it considers only target-level contexts $\mathbb{C}_\mathsf{T}$ that *behave like* some source-level context $\mathbb{C}_\mathbf{S}$ (written $\mathbb{C}_\mathbf{S} \approx \mathbb{C}_\mathsf{T}$ using the aforementioned source-target equivalence) [96]. Well-behaved contexts replicate the expressiveness of source-level contexts at the target level, so well-behaved target contexts, in essence, model other correctly compiled source-level code.

4.3 Equivalence-Preserving Compilation

A growing body of research is devoted to studying compilation that preserves certain forms of source-level equivalence. This has mostly taken the form of fully abstract compilation, which we discuss next in Section 4.3.1. There are also compilers that preserve other forms of source-level equivalence, which we discuss in Section 4.3.2 along with potential avenues for future work along these lines.

4.3.1 Fully Abstract Compilation. A compiler is fully abstract if it has that property that two source-level components are indistinguishable at the source target if and only if their compiled versions are indistinguishable at the target-level.

Definition 4.5 (Fully abstract compilation [1]). A compiler is fully abstract if it preserves and reflects contextual equivalence: $\forall \mathbf{P_1}, \mathbf{P_2} \in \mathbf{S}.\ \mathbf{P_1} \simeq_{ctx} \mathbf{P_2} \iff [\![\mathbf{P_1}]\!]^{\mathsf{S}}_{\mathsf{T}} \simeq_{ctx} [\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}}$.

Reflection of contextual equivalence usually follows as a consequence of compiler correctness, but preservation of contextual equivalence implies that no security flaws (as expressible via equivalences) are introduced by the compilation scheme. The security relevance of equivalence-preserving compilation comes from the fact that, as discussed in Section 3, many security properties can be expressed in terms of (source-level) contextual equivalence. Equivalence-preserving compilation considers all target-level contexts when establishing that the compiled components are indistinguishable, so it captures the power of an attacker operating at the level of the target-language. As mentioned in Section 1, if the source language is already insecure (i.e., it fails to provide abstractions that ensure security), then it can be "securely compiled" to a target language by preserving contextual equivalence, but the output of the compilation will still be insecure. Finally, note that (compositional) compiler correctness does not imply fully abstract compilation, nor vice versa; for details, we refer the reader to Section 1 of New et al. [94].

Source-Level Reasoning. An additional benefit of fully abstract compilers comes in the form of *source-level reasoning*. Source-level reasoning means that in order to understand how a component behaves, the programmer need only think about how it interacts with other code at the source level; there is no need to think about interactions with code from some other (possibly lower-level) language. From a security point of view, this property ensures that security properties of implementations follow from reviewing the source code and its source-level semantics [21, 109]. Source-level reasoning simplifies the task of a programmer, who need not be concerned with the

behaviour of target-level code (attackers) and can focus only potential source-level behaviors when reasoning about safety and security properties of their code.

Compiler full abstraction can also be formulated using stochastic assumptions, for example that certain values can be guessed only with negligible odds, by relying on probabilistic contextual equivalence (Definition A.5). This type of full abstraction result is called *probabilistic full abstraction*.

A fully abstract translation from one language to another is not always possible [97]. When it is achievable, the proof of full abstraction of a compiler is generally split into two theorems based on the preservation (⇒) and reflection (⇐) directions. So, Definition 4.5 may be split into two sub-statements:

$$\textit{Preservation} = \forall \mathbf{P}_1, \mathbf{P}_2 \in \mathbf{S}.\ \mathbf{P}_1 \simeq_{ctx} \mathbf{P}_2 \Rightarrow [\![\mathbf{P}_1]\!]^{\mathsf{S}}_{\mathsf{T}} \simeq_{ctx} [\![\mathbf{P}_2]\!]^{\mathsf{S}}_{\mathsf{T}}$$

$$\textit{Reflection} = \forall \mathbf{P}_1, \mathbf{P}_2 \in \mathbf{S}.\ [\![\mathbf{P}_1]\!]^{\mathsf{S}}_{\mathsf{T}} \simeq_{ctx} [\![\mathbf{P}_2]\!]^{\mathsf{S}}_{\mathsf{T}} \Rightarrow \mathbf{P}_1 \simeq_{ctx} \mathbf{P}_2$$

Both statements have a universal quantification over all possible contexts, due to the expansion of the definition of contextual equivalence (Definition 3.3). This makes these proofs particularly complicated, as languages such as assembly have contexts that do not offer a clearly inductive (or co-inductive) structure, and so are of little help for the proof. Some work adopt other equivalences, as seen in Section 3, that are as precise as contextual equivalence in order to simplify these proofs. For example contextual equivalence at the target level can be replaced with trace equivalence ($\overset{\mathsf{T}}{=}$, presented in Definition A.11) [61, 64, 99, 102] or with logical relations (Appendix A.4) [17, 18, 26, 129], changing the statement of preservation as follows:

$$\textit{Preservation with traces} = \forall \mathbf{P}_1, \mathbf{P}_2 \in \mathbf{S}.\ \mathbf{P}_1 \simeq_{ctx} \mathbf{P}_2 \Rightarrow [\![\mathbf{P}_1]\!]^{\mathsf{S}}_{\mathsf{T}} \overset{\mathsf{T}}{=} [\![\mathbf{P}_2]\!]^{\mathsf{S}}_{\mathsf{T}}$$

This statement does not involve a conclusion involving a universal quantification over all possible target-level contexts, so it is simpler to prove than the original *Preservation* statement. Appendix B discusses in detail existing proof techniques for proving this part of compiler full-abstraction, which is where the security relevance of the compiler, as well as all complications arise.

A correct compiler provides a good starting point for a fully abstract one, since if a compiler is correct we have Property 1 that

$$\textit{Reflection with well-behaved contexts} = \forall \mathbf{P}_1, \mathbf{P}_2 \in \mathbf{S}.\ [\![\mathbf{P}_1]\!]^{\mathsf{S}}_{\mathsf{T}} \overset{\mathsf{wST}}{\simeq_{ctx}} [\![\mathbf{P}_2]\!]^{\mathsf{S}}_{\mathsf{T}} \Rightarrow \mathbf{P}_1 \simeq_{ctx} \mathbf{P}_2$$

The reflection statement for fully abstract compilers has a stronger assumption, since it considers all contexts, not just well-behaved ones (which are a subset of all contexts). This gives us the following assumption that can be used to prove the *Reflection* statement from *Reflection with well-behaved contexts.*

$$\forall \mathbf{P}_1, \mathbf{P}_2 \in \mathbf{S}.\ [\![\mathbf{P}_1]\!]^{\mathsf{S}}_{\mathsf{T}} \simeq_{ctx} [\![\mathbf{P}_2]\!]^{\mathsf{S}}_{\mathsf{T}} \Rightarrow [\![\mathbf{P}_1]\!]^{\mathsf{S}}_{\mathsf{T}} \overset{\mathsf{wST}}{\simeq_{ctx}} [\![\mathbf{P}_2]\!]^{\mathsf{S}}_{\mathsf{T}}$$

The connection between fully abstract compilation and security is strong [58]; though it has also been studied without security as a main concern [111, 112]. In this other context, fully abstract translations were used to compare the expressiveness of different λ-calculi; if such a translation between two calculi exists, then they are equi-expressive.

A Note on Language-based Reflection. Some modern compilation schemes are fully abstract but they also have little security relevance. This is because the source languages involved have *reflection* i.e., the ability to examine and modify the structure and behaviour of an object at runtime. As highlighted in the past [88, 134], reflection rules out any sensible notion of contextual equivalence and with reflection, contextual equivalence is reduced to alpha-equivalence. Thus, a fully abstract compiler would be one that translates components with the same syntactic structure into components with the same syntactic structure, for example without any possibility to define confidential

data. Since reflection nullifies any useful abstraction based on the behaviour of compiled programs, reflection makes it almost impossible to define security properties. Thus, although a fully abstract compilation scheme may be definable for a language with reflection, it would not be secure, as the language would not be able to express any useful security property.

Modular Fully Abstract Compilation. For modular compilers from typed source to untyped target languages, full abstraction alone has shortcomings (explained in Section 5.2.3), as highlighted by Patrignani et al. [102] and by Juglaret et al. [64]. For such compilers to be secure, a property that has been proposed is modular full-abstraction (Definition 4.6). Denote a list of components $C^1, \cdots, C^n$ as C.

Definition 4.6 (Modular full-abstraction). A compiler $[\![\cdot]\!]^{\mathsf{S}}_{\mathsf{T}}$ has modular full-abstraction if

$$\forall C^1_{\mathsf{S}}, C^2_{\mathsf{S}}, C^3_{\mathsf{S}}, C^4_{\mathsf{S}}.(\forall C_{\mathsf{T}}.\ [\![C^2_{\mathsf{S}}]\!]^{\mathsf{S}}_{\mathsf{T}} \simeq_{\mathrm{ctx}} C_{\mathsf{T}}) \Rightarrow (\forall C'_{\mathsf{T}}.\ [\![C^4_{\mathsf{S}}]\!]^{\mathsf{S}}_{\mathsf{T}} \simeq_{\mathrm{ctx}} C'_{\mathsf{T}}) \Rightarrow \\ (C^1_{\mathsf{S}} + C^2_{\mathsf{S}} \simeq_{\mathrm{ctx}} C^3_{\mathsf{S}} + C^4_{\mathsf{S}} \iff [\![C^1_{\mathsf{S}}]\!]^{\mathsf{S}}_{\mathsf{T}} + C_{\mathsf{T}} \simeq_{\mathrm{ctx}} [\![C^3_{\mathsf{S}}]\!]^{\mathsf{S}}_{\mathsf{T}} + C'_{\mathsf{T}})$$

This property can be derived just by full abstraction (Section 4.3.1) and compiler modularity (Definition 4.2). The definition of modular full abstraction is equivalent to the following one (Definition 4.7), but it was presented that way to be as general as possible.

Definition 4.7 (Modular full-abstraction #2). A compiler $[\![\cdot]\!]^{\mathsf{S}}_{\mathsf{T}}$ has modular full-abstraction if: $\forall C^1_{\mathsf{S}}, C^2_{\mathsf{S}}, C^3_{\mathsf{S}}, C^4_{\mathsf{S}}.\ C^1_{\mathsf{S}} + C^2_{\mathsf{S}} \simeq_{\mathrm{ctx}} C^3_{\mathsf{S}} + C^4_{\mathsf{S}} \iff [\![C^1_{\mathsf{S}}]\!]^{\mathsf{S}}_{\mathsf{T}} + [\![C^2_{\mathsf{S}}]\!]^{\mathsf{S}}_{\mathsf{T}} \simeq_{\mathrm{ctx}} [\![C^3_{\mathsf{S}}]\!]^{\mathsf{S}}_{\mathsf{T}} + [\![C^4_{\mathsf{S}}]\!]^{\mathsf{S}}_{\mathsf{T}}$.

Modular full-abstraction is only relevant in a setting where the source is typed and the target is untyped. If the target is also typed, types enforce that a compiler is modular.

Secure compartmentalizing Compilation. A flavour of secure compilation that is reminiscent of the "preservation" part of full-abstraction is secure compartmentalising compilation (SCC) [64]. SCC is applicable for source languages with undefined behaviour while target languages do not and for compartmentalised programs, i.e., programs that can be split in a fixed series of components each implementing a certain interface. Define a component to be fully defined if its undefined behaviour cannot affect other components.

Definition 4.8 (Secure compartmentalizing compilation). A compiler is SCC if:

- For any compartmentalized program and for all ways of partitioning this program into a set of uncompromised components $\mathcal{C}_{\mathsf{S}}$ and their interfaces **CI**, and a set of compromised components $\mathcal{B}_{\mathsf{S}}$ and their interfaces **BI**, so that $\mathcal{C}_{\mathsf{S}}$ is fully defined with respect to **BI**, and
- for all ways of replacing the uncompromised components with components $\mathcal{D}_{\mathsf{S}}$ that satisfy the same interfaces **CI** and are fully defined with respect to **BI**,
- if $[\![\mathcal{C}_{\mathsf{S}}]\!]^{\mathsf{S}}_{\mathsf{T}} + [\![\mathcal{B}_{\mathsf{S}}]\!]^{\mathsf{S}}_{\mathsf{T}} \not\simeq_{\mathrm{ctx}} [\![\mathcal{D}_{\mathsf{S}}]\!]^{\mathsf{S}}_{\mathsf{T}} + [\![\mathcal{B}_{\mathsf{S}}]\!]^{\mathsf{S}}_{\mathsf{T}}$,
- then there exist components $\mathcal{A}_{\mathsf{S}}$ satisfying interfaces **BI** and fully defined with respect to **CI** such that $\mathcal{C}_{\mathsf{S}} + \mathcal{A}_{\mathsf{S}} \not\simeq_{\mathrm{ctx}} \mathcal{D}_{\mathsf{S}} + \mathcal{A}_{\mathsf{S}}$.

Intuitively, the source is allowed to have undefined behaviour because the assumption about full definedness ensures that undefined behaviour in a component does not affect code and data of other components.

The direction of this implication is exactly the contrapositive statement of preservation of compiler full abstraction. In fact, SCC can be implied by a form of full abstraction that is modular and that constrains the context into fixed partitionings of its components, while arbitrary contexts are loose in this regard.

4.3.2 *Other (Relational) Notions.*

Fully abstract compilers preserve contextual equivalence, but compilers can alternatively preserve other source-level relational properties that are relevant for security. Next, we discuss such secure compilation results.

Noninterference-Preserving Compilation. Bowman and Ahmed [26] present a translation from the dependency core calculus (DCC) [2]—which can be used to encode secure information flow—into a language with higher-order polymorphism (but no security types or lattices). They show that the translation is correct (preserves semantics) and, more notably, that it preserves noninterference, i.e., the property that a low-level observer (attacker) cannot distinguish high-level (protected) computations. Note that noninterference is a relational property: it is expressed using a logical relation $\mathbf{e_1} \approx_{\zeta} \mathbf{e_2} : \tau$ that captures observer-sensitive equivalence, namely that two terms of type τ look "equivalent" to an observer with level ζ. To express noninterference in the target, Bowman and Ahmed formalize a notion of observer-sensitive equivalence that makes essential use of both first-order and higher-order parametric polymorphism. In particular, they show how to encode DCC's security lattice and protection judgment using a *protection ADT* encoded in the target. Note that since observer-sensitive equivalence is not the same as contextual equivalence for DCC, Bowman and Ahmed's result is not a full abstraction result. Nonetheless, the proof relies on a backtranslation like full abstraction results.

Trace-Preserving Compilation. Patrignani and Garg [103] have proposed an alternative secure-compilation statement based on traces called Trace-Preserving Compilation (TPC). Trace-preserving compilation was studied for reactive languages, where traces fully describe the behaviour of a component and therefore trace equivalence and contextual equivalence coincide. Thus, TPC is stated in terms of traces and not in terms of preservation of equivalences.t Intuitively, a trace-preserving compiler generates code that (i) preserve the behaviour of their source-level counterparts when the low-level environment provides valid inputs and (ii) correctly identify and recover from *invalid* inputs. Invalid inputs are those that have no source-level counterpart (e.g., if booleans are encoded as the integers 0 and 1 by a compiler, then 2 would an invalid boolean input in the target). Condition (i) implies what is often called correct compilation—that the target preserves source behaviour when all interacting components have been compiled using the same (or an equivalent) compiler. Condition (ii) ensures that compiled code detects target-level attacks and responds to them in a secure way. Formally, we indicate the set of traces describing the behaviour of a program P as $\mathsf{TR}(P)$. Traces $\overline{\lambda}$ are lists of action λ and we differentiate input and output actions (i.e., received and generated by P) by decorating them with ? and ! respectively.

To formally define TPC we need a cross-language relation among actions $\approx \subseteq \lambda \times \lambda$ that is total, invertible and injective. This relation identifies what are valid and invalid actions and it can be used to state when whole traces are related by applying it pointwise to the elements of two traces $\overline{\lambda}$ and $\overline{\lambda}$. Additionally, $\surd$ is a target-only action that has no source-level counterpart and that is opaque, i.e., it does not depend on any hidden internal state. Finally, define $\mathsf{obs}(\overline{\lambda})$ to be the set of all even-length prefixes of trace $\overline{\lambda}$, i.e., those that end with an observable generated by compiled component.

Definition 4.9 (TPC). $[\![\cdot]\!]$ is TPC if $\forall\mathbf{P}$

$$\mathsf{TR}([\![\mathbf{P}]\!]) = \{\overline{\lambda} \mid \mathsf{obs}(\overline{\lambda}) = \bigcup_{n\in\mathbb{N}} \mathsf{int}_{\mathsf{n}}(\mathbf{P})\}$$

$$\mathsf{int}_{\emptyset}(\mathbf{P}) = \{\overline{\lambda} \mid \exists\overline{\lambda} \in \mathsf{TR}(\mathbf{P}).\, \overline{\lambda} \approx \overline{\lambda}\}$$

$$\mathsf{int}_{\mathsf{n+1}}(\mathbf{P}) = \{\overline{\lambda} \mid \overline{\lambda} \equiv \overline{\lambda}_1\lambda?\surd\overline{\lambda}_2 \text{ and } \overline{\lambda}_1\overline{\lambda}_2 \in \mathsf{int}_{\mathsf{n}}(\mathbf{P}) \text{ and } \nexists\surd \in \overline{\lambda}_1$$
$$\text{and no source trace relates to } \overline{\lambda}_1\lambda?\}$$

The case for $\mathsf{int}_{\emptyset}(\cdot)$ yields all valid traces, which satisfy condition (i) above. The other case satisfies condition (ii) as it considers traces that receive an invalid action ($\lambda?$) and respond to it with $\surd$, which we know does not leak information. The $n + 1$ case adds only one such $\surd$at any possible position in the trace, so long as it is in response to an invalid action.

Since it is defined in terms of traces, TPC has been related with hyperproperties preservation, proving that any trace-preserving compiler preserves a relevant subclass of hyperproperties: hypersafety [32]. Intuitively, hypersafety captures systems where "something bad does not happen" and it includes termination insensitive non interference as well as observational determinism. Patrignani and Garg [103] define how to preserve the meaning of hypersafety across languages and as an example it shows how the meaning of noninterference is preserved, so a TPC compiler preserves also noninterference correctly.

Example 4.10 (TPC preserves noninterference). We now give a brief example of how TPC preserves noninterference (and analogously any hypersafety), which also motivates what makes a trace-preserving compiler secure.

Noninterference is a hyperproperty because it is concerned with multiple runs of the same program, two in this case, that differ on the secret inputs but are the same on the public outputs. A TPC compiler preserves noninterference because it produces code whose target traces include those that relate to the source ones, so we know that those do not violate noninterference. Additionally, the other traces include $\surd$, and we know that $\surd$ is opaque (which in the noninterference setting means public) so it is ok to respond with it to extra target inputs, as doing so does not leak any information.

TPC has also been proven to be strictly stronger than general full abstraction under the assumption that trace equivalence and contextual equivalence coincide. However, if a compiled component replies uniformly and securely (e.g., by halting the computation) to all invalid inputs, the two notions coincide. This is particularly relevant since all existing fully abstract compilers actually behave this way, so they attain TPC and so it is clear that they also preserve hyperproperties.

Future Definitions. In future work, it might be useful to consider compilation schemes that preserve more limited notions of equivalence, instead of contextual equivalence. In particular, this might help with the significant limitation of dynamically enforced full abstraction results, namely the large performance overheads of compiled code. For instance, one could preserve only integrity properties, so as to prevent code from being tampered with, but not checking for violations of confidentiality properties. This would demand insertion of fewer dynamic checks in compiled code and could potentially improve its efficiency. We expect these and other such notions will be the subject of future work on dynamically enforced secure compilation [53].

This section has presented the different formal statements of a secure compiler. Discussing the techniques used to prove that these statements is done in Appendix B; this paper now surveys existing secure compilation work that adopt them.

5 ACHIEVING SECURE COMPILATION

This section surveys the existing research on secure compilation. We broadly partition existing work in two main branches depending on how they achieve secure compilation: statically (Section 5.1) or dynamically (Section 5.2). Each of these sections presents a list of the work that fit the related approach. For each work we give a brief description of which source and languages it uses as well as which interesting features these languages have. Additionally we give a high-level overview of the compiler and of how it achieves security. Finally, for each work we state what formal compiler property is proven and by virtue of which proof technique.

5.1 Static Secure Compilation

Only one way of achieving secure compilation statically exists for now, which is by relying on a type system for the target language (Section 5.1).

To prove the steps of the type-preserving compiler of Morrisett et al. [90] correct (cfr Section 4.1), Ahmed and Blume [17] proved that typed closure conversion from (and to) System F is fully abstract. This translation exploits additional typed wrappers for source terms in the target language. Typed closure conversion turns each function into a closure: a pair consisting of a function pointer and an environment that provides bindings from free variables to values. The conversion is type-directed and generates typed pairs, which are given the type of their closure environment. Typed wrappers are terms that translate source values $\mathbf{v}$ of type τ to target values of type $[\![\tau]\!]$ based on the syntactic structure of $\mathbf{v}$ and of type τ. As a proof technique the authors adopt a cross-language step-indexed logical relation [16] to prove the translation fully abstract. Moreover, the proofs exploit several key properties of typed wrappers: wrapper termination (i.e., wrapper functions are total), cancellation (i.e., a translation from τ to $[\![\tau]\!]$ and one from $[\![\tau]\!]$ to τ cancel each other) and parametricity (enabling the usage of wrappers for abstract types). In subsequent work, Ahmed and Blume proved that a typed CPS translation from the simply-typed λ-calculus to System F is also fully abstract [18]. Moreover, they invalidated one of the type-preserving steps of the type-preserving compiler to TAL of Morrisett *et al.* [90], namely that CPS conversion to TAL is type preserving. Instead of using global "answer types" (i.e., the type of the continuation), the typed CPS translation of Ahmed and Blume gives each continuation its own individually abstract answer type. Consequently, a well-typed function in typed CPS form can only use its continuation, and this prevents "bad" target terms from being well-typed. To prove the translation fully abstract, the authors combine source and target language in a Matthews and Findler-style multilanguage system [80] so that both languages have access to each other's values. Preservation and reflection of contextual equivalence is proven by using a multilanguage step-indexed logical relations.

Barthe et al. [22] devised a secure compilation scheme from a WHILE language to a typed, stack-based assembly language. Both languages have information flow type systems, which is the mechanism exploited by the compilation scheme to be secure. Information-flow type systems enforce non-interference, as seen in Section 4.1.1. The secure compilation scheme produces target code that has the non-interference properties of the source code, thus making the translation security types-preserving. Since the security properties of the source language stem only from the type system, the compilation is secure. Subsequently, the authors extended their secure compilation results to a concurrent setting, extending both source and target languages with thread creation [23]. The compilation scheme exploits the typing information to label its output code as being either high or low security. Then, this information is fed into a secure scheduler, which uses it to ensure that the interleaving of observable events may not depend on sensitive data. Together, the compiler and the scheduler prevent internal timing leaks to an attacker with access to low security variables.

Tse and Zdancewic attempted a security-preserving translation from the dependency core calculus (DCC) to System F [129]. DCC extends Moggi's computational λ-calculus [89] with a notion of program dependency that captures security, partial evaluation, program slicing, and call-tracking properties [2]. The translation uses type variables in System F as keys to access translated DCC data at a given security level so that the data can be accessed only with the right key. To support decrypting low-security data with a high-security key, keys can be downgraded along the security lattice ordering. To prove that the compiler is secure, they showed that the translation preserves and reflects the observer-sensitive equivalence formalized using logical relations. The proof that compiled terms are non-interfering should then follow from the parametricity theorem for System F. However, there was a flaw in Tse and Zdancewic's proof, pointed out by Shikuma and Igarashi. The latter then proved an analogous result for an extension of DCC with protection contexts, which they translate to the simply-typed λ-calculus extended with base types to represent each label ℓ in the source-language security lattice $\mathcal{L}_\ell$ [118].

A secure compilation scheme from DCC (without protection contexts) to F_ω was then developed by Bowman and Ahmed [26] who showed how to enforce source-level noninterference using parametric polymorphism at the target. To prove that the translation preserves noninterference, they developed a cross-language open logical relation (inspired by the work of Zhao *et al.* [142]). Their proof relies on back-translating compiled target-level terms of translation type—i.e., whose type is the translation, written τ^+ of some source type τ—into source terms of type τ, as in the aforementioned work on typed CPS translation [18]. Unlike the standard logical relations (as seen in Appendix A.4), their relation needs to be open as their translation produces types and terms with free variables. If they were closed, they would not be able to back-translate terms. All this work targets the terminating fragment of DCC; it is still unknown if DCC extended with recursion can be still securely compiled to a parametric calculus.

League et al. [75] developed a secure compilation scheme from Featherweight Java (FJ) [107] to F_ω that exploits the latter's higher order type system (extended with ordered records, fixed-point functionality, recursive types, existential types and row polymorphism) to be secure. The compilation scheme translates each FJ class into an F_ω term where fields are collected in one record and methods are collected in a separate record which represents the virtual method table shared by all instances of the class. In this type-preserving translation, compiled F_ω terms preserve the typing information of their source level counterparts. The type system of FJ is not the only security mechanism: classes can have **`private`** fields which are securely compiled by using existential types.

The dependent type system of DCIL has been adopted as the basis for a type-preserving compiler for *FINE* programs [30]. DCIL is an object-oriented bytecode format that extends Microsoft's Common Intermediate Language (CIL) with dependent types. *FINE* is an ML-like language deriving from *F#*; it provides dependent refinement types and affine (use at most once) types. These make *FINE* well-suited for writing security-critical components, as programmers can employ the advanced type system to specify complex access control policies. Chen *et al.* provided a compiler that exploits the dependent types of DCIL to ensure that the functional dependencies of *FINE* functions are respected and that affine-typed terms are used at most once [30]. Since types are the only security abstraction of *FINE*, the type-preserving compiler is secure. This secure compiler is the basis for the work of Swamy et al. [124], which focusses on the security features of F^* and their applicability in a distributed setting, though this work not provide formal security properties of the compiler.

New et al. [94] recently presented a fully abstract closure conversion result for a translation whose target language contains control effects but the source does not. They perform closure conversion from a simply-typed lambda calculus with recursive types to System F extended with a modal type system to track exceptions. The type translation ensures that compiled code can only be linked with computations that are well-behaved—specifically, they may not throw un-handled

exceptions. A key contribution of this work is a new back-translation technique, called *universal embedding*, that is needed for the proof of full abstraction (this is discussed in Appendix B.3.1).

5.2 Dynamic Secure Compilation

Compilers that achieve security dynamically do so in three main ways: using cryptography in the target language (Section 5.2.1), inserting runtime checks (Section 5.2.2) and exploiting security architectures (Section 5.2.3).

5.2.1 Cryptographic Primitives for the Target Language. This section describes work devising compilers for distributed and concurrent languages, which are subject to the Dolev-Yao attacker presented in Example 2.9. These papers achieve secure compilation by exploiting target-level cryptographic primitives to protect messages exchanged between target-level processes.

Abadi, Fournet and Gonthier extensively studied the application of cryptographic primitives to securely compile inter-process, message-passing-based communication, both in concurrent and distributed settings [5–7]. These authors adopt source languages that are variations of the join calculus, a model of concurrency where processes send and receive messages on first-class channels. The join calculus is reminiscent of the π-calculus [86, 114]; these languages are also equivalently expressive (up to weak barbed congruence [48]). The target language chosen in these secure-compilation results is the secure join calculus (or Sjoin calculus), which extends the join calculus with security primitives for encryption ($\{M\}_k$) and decryption (**case** L **of** $\{M\}_k$ **in** P) of message M with key k.

The first work [5] presents a fully abstract compilation scheme for processes which are given secure local and global communication primitives. Here, translated processes are wrapped in a "firewall" process that (i) maintains key pairs for cryptographic primitives and (ii) transforms communication on global channels into security protocols employing those primitives. The translation is proven to preserve and reflect weak bisimulation, which is the notion that frequently replaces contextual equivalence for concurrent calculi (Appendix A.2).

Subsequent work by the same authors [6] develops a secure compilation scheme for the join calculus extended with support for principals, so that the calculus has authentication primitives. In the target language, principals are translated into key pairs that are used to generate unforgeable certificates that prove principals identities. The translation is proven to preserve and reflect weak bisimulation, but only in the presence of noise in the network (i.e., enough encrypted messages) to prevent traffic analysis.

Additional work by these authors provided a secure compilation scheme for the join calculus extended with constructs to create secure channels [7]. In this translation, target processes are given a cryptographic key for each communication channel they define and placed behind a "firewall" that keeps track of key usage. Communication is translated into the execution of cryptographic protocols which use nonces and other techniques to thwart different kind of attacks. Full abstraction of the translation is again proven by showing that weak bisimulation is preserved and reflected, again relying on the presence of noise in the network. Given LAN or wireless network implementations, the assumption of noise in the network seems well justified.

This line of work was further expanded by Bugliesi and Giunti [27], who provided a secure compilation scheme from a dynamically typed π-calculus to the applied Spi calculus which relies on cryptographic operations to secure channel communication. The Spi calculus is an extension of the π-calculus with cryptographic primitives, much as the Sjoin calculus is to the join calculus. The usage of the π-calculus (as opposed to the join calculus of previous work) allows forward secrecy attacks (cfr Example 2.9) to be modelled. This is because in the π-calculus a communicated channel can also be used to perform input, while this is not possible in the join calculus. This work

presents a translation that protects against these attacks by extending translated processes with self-signed certificates and a proxy server. Self-signed certificates are the target-level implementation of source-level channels; these certificates include the channel identity and two encryption keys corresponding to the input and output capabilities. The proxy server keeps track of cryptographic keys related to channels to preserve the expected interactions between processes. Full abstraction of the compilation scheme is proven by showing it preserves and reflects weak bisimulation.

Techniques similar to those employed by Abadi, Fournet, and Gonthier are also employed by Adão and Fournet [13], who develop a secure compilation scheme for a π-calculus extended with secure channels, mobile names, and high-level certificates. The characteristic of their work is the target language and the adversary model. The target language is a set of machines that have input and output network interfaces and can perform cryptographic operations. The adversary is modelled as a probabilistic algorithm that controls that network and some corrupted machines. The compilation scheme is again proven to preserve and reflect weak bisimulation.

The work of Laud [74] also exploits encrypted and signed messages to securely compile the programming language ABS into the applied π-calculus extended with cryptographic operations. ABS is a concurrent, object-oriented language with asynchronous method calls and futures [63]. The compilation scheme translates each asynchronous method call into an explicit message which is uniquely identified by means of a fresh cryptographic key. Compiled objects are also uniquely identified by means of fresh keys associated to them. The semantics of both the source and the target languages are given in terms of LTSs, where attackers are modelled as other LTSs that can synchronise on visible actions. The translation is proven to preserve and reflect weak bisimulation defined on these LTSs.

Duggan [43] provide a secure compilation scheme from a π-calculus-like language with cryptographic types and principals to the Spi calculus. Cryptographic types express cryptographic guarantees on values at the type-system level since types indicate that certain values are encrypted or signed by certain principals. The information regarding principals is hidden for communication over an insecure medium, while it is exposed when the medium is trusted. The type system performs static checks to ensure that values are used in the expected manner. It also performs dynamic checks on cryptographic-typed values when principals data is unavailable. The compilation scheme inserts cryptographic operations only when a dynamic check is required, thus minimising the computational overhead introduced by cryptography. As in work mentioned above, principals are translated to pairs of keys which are used for encryption and signing of network messages. As before, contextual equivalence is replaced by weak bisimulation for fully abstract compilation.

Session types ensure that distributed parties adhere to a protocol. The latter is encoded as a session: a sequence of actions detailing what messages are communicated between various peers [29]. Corin et al. [33] presented a secure compiler from *F#* extended with session types to *F#* extended with libraries providing cryptographic primitives. The secure compiler produces code that is shielded from the attempts of other peers to deviate from their session by exploiting the cryptographic primitives of the target language. The compiler does not introduce additional messages but it maps each session action to a cryptographic message between the same sender and receiver. Each cryptographic message contains a unique session identifier and the signatures of the sender and of the senders of the previous messages, so it can be uniquely identified. Any attempt to tamper with the integrity of the session can thereby be detected and any such message can be dropped. The secure compiler is proven fully abstract by adopting a labelled operational semantics which makes explicit the communication between secure code and a potential attacker.

In the case of distributed languages, Fournet et al. [50], present a fully abstract compilation scheme for a distributed WHILE language featuring a type system with security levels. The target language is a WHILE language extended with cryptographic libraries and with threads that reside

at different locations. The compiler performs four passes of the source code in order to generate secure target code. First, source code with location annotations is sliced into local programs, each meant to run in a different location. Second, each local component is extended with global variables to keep track of its state. Then, for each global variable, a local replica is created, together with additional functionality for explicit updates between these replicas. Finally, global variable updates are protected with cryptographic operations and the keys that regulate these operations are disseminated to the threads. To prove that the translation is secure, each step is proven to be computationally sound, i.e., it preserves noninterference [51]; each step is also proven to be correct.

Another class of distributed languages is multi-tier languages, which are adopted to develop web applications split into several tiers (i.e., client, server, database, etc.) that can reside on different machines. Baltopoulos and Gordon [21] describe a secure compilation scheme for the multi-tier language TINYLINKS into *F*7, an ML dialect extended with refinement types. The secure compiler is proven to preserve data integrity and control integrity properties of well-typed source components in the generated target components by exploiting authenticated encryption mechanisms. Malicious attackers are modelled as untyped contexts which also have power over the network connecting the different tiers. The compiler is secure because it is proven to translate well-typed components into robustly-safe *F*7 expressions. These expressions are a subset of *F*7 expressions that are immune to attacks on data and control integrity.

5.2.2 Dynamic Check Insertion. This section describes work that achieves secure compilation *mainly* through the addition of dynamic checks in the generated target code. Specifically, while there is other work that adds dynamic checks to make their compilation schemes secure, since they rely primarily on mechanisms other than the checks, they will not be discussed here, but in the next section. When adding dynamic checks in the generated target code, it is crucial that the attacker must not be able to tamper with these checks as that would render them void. Each of the results presented below adopt different techniques to protect the inserted checks.

Ghica and Al-Zobaidi [54] describe a fully abstract compilation scheme from a λ-calculus extended with iteration to VHDL digital circuits. Security of compilation is achieved through the addition of a runtime monitor that forces external code communicating with the generated digital circuits to respect the expected communication protocol. The attacker is prevented from tampering with the hardware and thus cannot disrupt the runtime monitor.

Fournet, Swamy, Chen, Dagand, Strub, and Livshits [52] use defensive wrappers in concert with other techniques described below to securely translate a monomorphic ML-like language with mutable references and exceptions (dubbed f*), to an encoding (dubbed js*) of JavaScript in f*. The compiler first translates f* terms into js* terms, then defensively wraps these terms to protect them from features of js* that can be exploited by a malicious attacker. Defensive wrappers provide dynamic type checks for the untyped js* code. For the compiler to be secure, compiled code first makes a local copy of trusted values from the global namespace, to prevent an attacker from redefining these values. Second, it exports defensively-wrapped, translated terms into the global namespace to make them globally available. The authors prove that the translation is fully abstract using a labelled bisimulation, called applicative bisimulation, in place of contextual equivalence.

Using Fournet *et al.*'s wrappers, Swamy et al. [125] have developed a compiler that is type preserving for a gradually-typed language into JavaScript. The source language is TypeScript*, a gradually-typed [119] version of JavaScript made sound and extended with `UN` types to typecheck opponent code [49, 57]. The compiler uses wrappers around typed values to convert them safely from ? (the dynamic type of the gradual language) to `UN` types and vice-versa. Additionally, it provides extra objects that ensure that TypeScript* typing is sound (unlike normal TypeScript). By ensuring that specific JavaScript files are loaded first, the compiler prevents that the additional TypeScript*

objects and wrappers can be tampered with by a JavaScript attacker. The compiler preserves memory safety of the additional objects and typing; for the proof of the latter both TypeScript* and JavaScript are modelled in f*, which is typed and thus allows stating type-preservation.

Devriese et al. [37] devised a simple compiler from the simply-typed lambda calculus with a fix operator to the untyped lambda calculus. The compiler performs type erasure and dynamic typechecks on values that are received from outside the compiled function. Each value taken from the context is checked to match the expected type, e.g., a value of type **Unit** is checked to be $[\![\mathbf{unit}]\!]^{S}_{T}$, a pair is checked to syntactically be a pair (and each element is checked to be of the right type). While the result is unsurprising, the technique used to establish full abstraction is the first of its kind as they rely on an approximate backtranslation (as described in Appendix B.3.3). The proof was then mechanised, making it the first full abstraction result to be fully mechanised in Coq [36].

5.2.3 Security Architectures for Memory Protection. Memory-related attacks (such as those captured by Theorems 2.1, 2.2 and 2.4) have resulted in a large body of research on memory protection mechanisms. The most relevant of these mechanisms for secure compilation are Address Space Layout Randomisation and Protected Module Architectures. The remainder of this section presents each security mechanism, followed by the secure compilation results using them.

Address Space Layout Randomisation (ASLR). ASLR is a technique that randomises the memory layout of key data areas of a program such as the base of the stack, of the heap, of libraries, etc. when creating an executable. The executable is divided in segments whose order is randomised by the dynamic linker (i.e., just before running the executable). This technique is used to hinder an attacker from mounting "return to libc" attacks, and from using previously acquired knowledge of the location of certain data to access that data in subsequent runs of a program. When a "return to libc" attack is mounted, the attacker exploits a buffer overflow to overwrite the return address of a function to a known function, e.g., one residing in libc. This known function can then be used to mount the attack, e.g., the libc contains functions to execute shell commands that can be used to carry out the intended attack.

Considering Example 2.1, if the memory is large, and the location of `secret` is randomised at every run, an attacker will not be able to access the `secret` with high probability and thus cannot break its confidentiality. In the case of Example 2.4, the attacker will not be able to guess the location where `Object y` is allocated, so she cannot tamper with it.

Abadi and Plotkin [9] adopted ASLR to achieve probabilistic fully abstract compilation in a λ-calculus setting. Their source language is a simply-typed λ-calculus extended with an abstract memory: a mapping from locations to values; locations can be public or private. Their target language is a λ-calculus extended with a concrete memory: a mapping from natural numbers (in this case they assume an unbounded memory) to values. Abadi and Plotkin prove that with a large enough memory, ASLR ensures that an attacker operating at the target level has a negligible chance of guessing values, thus achieving probabilistic full abstraction.

Subsequently, Jagadeesan et al. [61] extended this secure compilation scheme to a source language with more complex features. The source language is $\lambda\mu$hashref-calculus, a λ-calculus extended with operations for testing the hash of a reference and with the following features: dynamic memory allocation, higher-order references and call-with-current-continuation. The target language is the $\lambda\mu$probref-calculus, a λ-calculus with the ability to reverse the hash of a reference. Reversing a hash succeeds when a reference is known, but it is complex when the reference is unknown due to the large memory layout and the random allocation of references in memory. The authors develop LTSs for both $\lambda\mu$hashref-calculus and $\lambda\mu$probref-calculus which yield trace semantics which is then used in place of contextual equivalence to prove full abstraction of the translation (as explained

in Appendix A.2). The hashref feature grants the attacker the power to compare references and to know the format of a reference so that he can guess reference locations. Although it is not what lambda calculi typically provide, this kind of knowledge is available in C-like languages, so hashref is a justified addition to the language. The probref feature does not have a corresponding language construct but it models an attacker trying to break a hash function (e.g., by running collision attacks [135]).

Protected Module Architectures (PMA). PMA is an assembly-level isolation mechanism implemented in several research [81, 82, 95, 122] and industrial prototypes such as the Intel SGX processor [83]. PMA logically partitions the memory in a protected and an unprotected section. The protected section is further divided into a code and a data section. The code section contains a variable number of entry points: the only addresses to which instructions in unprotected memory can jump and execute. The data section is accessible only from the protected section. Code running in the unprotected section has unconstrained access to unprotected memory.

Considering Example 2.2, if the variable `secret` is allocated in the protected data section, its integrity cannot be tampered with from code executing in the unprotected memory section.

A protection mechanism analogous to PMA is sandboxing [56, 141]: a trustworthy environment extended with a location (the sandbox) where non-trustworthy code is placed and monitored so as to detect any malicious action it performs. Conceptually, sandboxing seems to be dual of PMA, thus we expect that the same insights developed in the secure compilation work for PMA would lead to the development of a secure compiler for sandboxed programs. However, no such secure compiler has been devised yet.

Agten et al. [14] were the first to present a fully abstract compilation scheme that uses PMA to preserve confidentiality and integrity properties of their source language. They devised a secure compilation scheme for a language with objects, interfaces and first-class method references to an assembly language extended with PMA. The compilation scheme places the objects to be secured in the PMA protected memory partition. Then, it creates entry points for methods appearing in interfaces, so that external code can call them. Secure methods activation records are allocated on a secure stack that resides within the protected memory section. Dynamic checks are introduced for all values communicated to and from the unprotected section in order to prevent ill-formed values affecting the computation (analogous to the wrappers of Fournet, Swamy, Chen, Dagand, Strub, and Livshits [52]). For example, primitive-typed values are checked to be inhabitants of that type, so a **Bool** value is checked to be one of two values, the compiled versions of **true** and **false**.

Patrignani et al. [99, 101] expanded the work of Agten *et al.* to a source language with dynamic memory allocation, exceptions and inner classes. This source language is an extension of Java Jr. [62]: a Java-like object-oriented language that provides strong encapsulation of classes and objects, which are not visible outside the package that defines them. Moreover, packages communicate based on exported interfaces and exported objects. With the introduction of class types, the secure compilation scheme introduces more dynamic checks. Objects are checked to see whether they define the method they are called on, which is possible as objects are allocated in the secure memory partition alongside their type information (which is also used for dynamic dispatch). Similar checks are performed on parameters whose type is a securely-defined class. Moreover, the identities of objects passed to the unprotected code are replaced with natural numbers, so as to obscure the allocation strategy of objects in the protected memory section and prevent related attacks. When an exception is thrown from unprotected code, it is also checked to be among the exceptions that could be thrown (i.e., defined in the method signature) and not a maliciously crafted one.

This work was subsequently extended by Patrignani et al. [102] to support modular (secure) compilation. In fact, the authors showed that if the fully abstract compiler above is used to compile

two components and link the related PMA modules with each other, the resulting target program is vulnerable to attacks. These attacks went unnoticed since they rely on invariants being shared between multiple components and multiple PMA modules, but the compiler above only considers one of them. To support modular secure compilation, the compiler introduces a central module Sys that is a proxy for method calls and returns, so it enforces well-bracketed control flow. All compiled components divert their calls through it and they accept incoming jumps to entry points only if they come from Sys. Additionally, Sys tracks globally-shared objects, so runtime typechecks on them can be performed inside modules.

The work of Agten et al. [14] and the first result by Patrignani et al. [99, 101] both prove compiler full-abstraction while the second result Patrignani et al. [102] proves modular compiler full-abstraction. When proving these properties, they assume that reflection of contextual equivalence holds, claiming that most compilers achieve this. By relying on a fully abstract trace semantics for the target language [100], they prove the contrapositive of the preservation of contextual equivalence. This is proven by devising an algorithm that always constructs a source-level context that differentiates between two components that exhibit different target-level traces.

Still exploiting the PMA architecture, Larmuseau *et al.* lift the PMA mechanism into the operational semantics of a Matthews and Findler-style multi-language system [80]. They securely compile both a simply typed λ-calculus [72] and a lightweight ML featuring locations and basic data structures to this multi-language system [71]. For both results, they prove that the compilation scheme preserves and reflects weak bisimulation. They subsequently extended these results with the derivation of a secure abstract machine for ML that uses PMA to protect against a malicious context [73]. Each step of the abstract machine is proven to be fully abstract (much like a multi-pass compiler), so the whole abstract machine is fully abstract.

The Pump Machine. A general-purpose security architecture is the Pump machine [39, 40], which is the base of the Crash-Safe machine [20]. The architecture permits arbitrary meta-data tracking at the machine code level, which can be used to enforce security policies such as information flow. All registers and memory cells have tags linked to them which are checked upon execution of any instruction to match the policy bound to that instruction. If this check fails, then the instruction is not executed. Otherwise it is executed, the tags are updated and the tag combination is cached to improve later similar checks. Since tags are arbitrary logical statements, the Pump machine can isolate code and data and predefined entry points for executing code (analogous to PMA modules) which are called compartments.

An idealised version of this architecture has been targeted with a compiler for an unsafe source language with components procedures and buffers [64]. The target language is a compartmentalised RISC machine that creates isolated compartments that can only jump between each other end exchange data via registers. The compiler translates each source component into a target compartment, creating entry points to the compartment for each function that can be called from outside the compartment. At each jump towards the outside of a compartment, all registers that do not carry jump-related information are reset. The returning address is enforced to be linear, so it cannot be misused and this guarantees well-bracketed control flow. By instrumenting the target language with a fully abstract trace semantics, they prove secure compartmentalising compilation.

An analogous but much richer result was also presented by Juglaret et al. [65]. This work is still considering an attacker model of mutually-distrustful compartments that can be compromised by an attacker. The source language is an object-oriented language while the target is still a RISC machine with symbolic micro policies (i.e., the Pump machine with an abstraction over how micropolicies are implemented). The compiler also deals with an intermediate language that is an object-oriented stack machine. Each class is compiled to its own component, with a local stack and entry point for

public methods. As before, returning is handled linearly. Parameters' types are encoded in tags, so compiled code uses them to perform runtime typechecks upon receiving context parameters.

5.3 Remarks on the Static and Dynamic Techniques

The two families of techniques for achieving compiler security have two different models in mind. The static technique generally involves a larger trusted computing base, where often all the software up to the linker is trusted, and this often includes a typechecker. It is at this linking time that the attacker is typechecked and, should her code be ill-typed, no linking will happen (if the code is well-typed, no attack can arise). The attacker thus lies just above this software and he supplies malicious code that needs to be linked against the code to be protected. In contrast, the dynamic techniques generally consider a setting with a smaller trusted computing base, where the attacker potentially controls all the software running on the system except for the compiler and the linker. The code to be compiled runs no checks on the attacker code, and it simply links against any possible code. By relying on the additional dynamic checks, when the attacker code misbehaves, such misbehaviour is prevented from having security repercussions.

We envision that the two technique can be integrated in the development of compilers and we discuss this subject further now in Section 6.

6 OPEN CHALLENGES AND FUTURE PERSPECTIVES

This section discusses challenges and directions for future work on secure compilation.

One of the greatest challenges for secure compilation is the development of a secure compiler for a realistic programming language, comprising I/O, libraries and so forth. The existing approaches show promising results, but the application of secure compilation techniques to mainstream programming languages has not yet been achieved. A real compiler comprises many intermediate steps, where a source program is translated to a series of intermediate programs before being turned into a target one. These intermediate languages often have types, which are used to perform static analysis and guide code optimisation. Often, only the last step of the compiler goes from typed to untyped languages. We can therefore envision that both static and dynamic techniques are useful when devising a realistic compiler which preserves security throughout all intermediate steps.

Specifically, the fine line between static and dynamic approaches can be bridged using proof carrying code (PCC [92]). Some of the surveyed work envisage such a cooperation between PCC and secure compilation [21, 75, 90] and some papers use some form of PCC to guide the compiler during the translation [22, 23]. PCC is a mechanism that binds a component to a proof of its properties, so that a host can check the properties of a component before executing it. PCC can be integrated with secure compilation to allow the insecure code to prove that it is compliant to a pre-defined agreement, i.e., it is not malicious. This way, static techniques can be employed by compiling a component to a typed language, erase its types and deliver it with a proof that it is well-typed. This reduces the trusted computing base as only the proof checker is required to be trustworthy. Such a component will need to use dynamic techniques when linking against components without a proof, but it will need no additional feature to link against code that comes with a proof of its well-typedness, thus incurring in no overhead. This subject brings us to the next intriguing topic for secure compilers: efficiency.

Efficiency of securely-compiled code is rarely considered in existing work. Two main research paths are envisioned to provide more efficient secure compilers, besides the already-discussed use of PCC: code optimisation and changes to the criterion for compiler security. Code optimisation has rarely been considered explicitly as a step of the surveyed secure compilers. Investigating the interaction between code optimisation and security has been recently proposed in a call-to-arms paper but has not been thoroughly carried out [42]. We foresee that studying which optimisations

violate which security properties will be an interesting research trajectory for secure compiler development. Finally, compiler full-abstraction often requires compilers to insert checks that do not deal with security violations. Developing new secure compilation criteria that fine-tune the checks inserted by the compiler so that they only affect code security is a topic that researchers are starting to look at now through preliminary work [10, 11, 47, 53, 104].

From the perspective of source languages, we envision that studying the secure compilation of advanced type systems is necessary, as they are a primary origin of source-level security enforcement. We provide but two examples of interesting type systems whose secure compilation require study, though we believe that this concept can be generalised to any type system that enforces security policies.

Many programming languages provide a form of polymorphism (e.g., Java generics), which lets programmers both abstract from the details of specific types and achieve information hiding. The connection between source-level polymorphism and target-level encryption has been vastly studied [79, 108, 123], to the point where Sumii and Pierce conjectured that (parametric) polymorphism can be securely compiled using encryption. While this conjecture has been disproved [38], the mechanisms that allow secure compilation of polymorphism are still unknown. Further exploration of the lambda-cube (after standard polymorphic programs) is also necessary. In fact, dependently-typed intermediate languages seem a necessity for compilers as intermediate languages often need to accommodate the widest array of source-level abstractions [133]. For a compiler using a dependently-typed language it is necessary to be able to compile those type abstractions securely, so this research direction needs to be investigated.

From the target language perspective, two main research trajectories can be seen: supporting concurrency and studying more performant security architectures. In fact, no existing secure compiler targets concurrent untyped assembly language. As presented, concurrency-related secure compilation schemes have been studied in a distributed setting but only in a message-passing based model of concurrency or in a typed multithreaded assembly setting. However, *untyped* multithreaded assembly language is a reality in modern machines. Supporting such a target language seems necessary to bring secure compilation to mainstream audiences.

Concerning secure architecture development, recent advances in the field (e.g., PMA, ASLR and the Pump machine) provided architectures that can support secure compilation, though other interesting ones can be studied. Specifically, capability machines [136, 138] are a security architecture that embody the capability paradigm of Dennis and Van Horn [35]. The idea revolves around the concepts of *subjects*, *operations* and *objects*: subjects perform operations on objects. For example, Alice (a subject) can write (an operation) something to the filesystem (an object). With capabilities, Alice's write succeeds only if she is able to present a capability that allows her to *at least* write to the filesystem. In this case, we say that Alice has *permission* to write the filesystem. Were she not able to present such a capability, the write would fail. With capabilities, *only connectivity begets connectivity*. So if a subject cannot create a capability to an object, and if she does not receive a capability to an object, she cannot perform operations on the object. Capabilities are *not forgeable*, all Capability Machines provide a *supervisor* (to use the terminology of Dennis and Van Horn) that ensures this. The literature on the subject is vast, but for secure compilation purposes, we are interested in implementations of this machine. Few capability machine implementations exist: the M-Machine [28] Capsicum [136] and Cheri [138]. In the M-Machine, capabilities are words whose first bit is set to 1 (no instruction can set this bit to 1 without calling the supervisor). Capsicum introduces capabilities at the OS level in a BSD-like operating system. Cheri introduces capabilities at the hardware level like the M-Machine and adds a capability co-processor to the CPU in order to

handle capabilities; normal instructions then succeed only if an enabling capability is present in the co-processor. Preliminary work showed that Cheri is a viable candidate for secure compilation [128].

Finally from the formal methods perspective, these contributions are envisaged. First, further study of proof techniques for compiler full abstraction, e.g., integrating under- and over-approximating embeddings, will be necessary to study secure compilation of languages with advanced constructs. Second, as mentioned in Section 4.3.2, novel criteria for compiler security, e.g., property-directed [53], are necessary to reduce the overhead that current definitions incur into. Moreover, no criteria currently encompasses side channels, which are a serious attack vector for example for cryptanalisis. Studying the interplay between side channels and secure compilers, and integrating this in secure compiler definitions will be a challenging research goal.

Acknowledgements. The authors would like to thank Dominique Devriese, Akram El-Korashy, Deepak Garg, Cătălin Hriţcu, Adriaan Larmuseau, Frank Piessens, Tamara Rezk and the anonymous reviewers for useful feedback and discussions on an earlier draft.

This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

REFERENCES

[1] Martín Abadi. 1999. Protection in programming-language translations. In *Secure Internet programming*. 19–34.

[2] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, 147–160.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. A Theory of Secure Control Flow. In *Proceedings of the 7th International Conference on Formal Methods and Software Engineering (ICFEM'05)*. Springer-Verlag, 111–124.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (2009), 40 pages.

[5] Martín Abadi, Cédric Fournet, and Georges Gonthier. 1999. Secure Communications Processing for Distributed Languages. In *IEEE Symposium on Security and Privacy*. 74–88.

[6] Martín Abadi, Cédric Fournet, and Georges Gonthier. 2000. Authentication Primitives and their Compilation. In *Proceedings of the 27th Symposium on Principles of Programming Languages (POPL '00)*. ACM, 302–315.

[7] Martín Abadi, Cédric Fournet, and Georges Gonthier. 2002. Secure Implementation of Channel Abstractions. *Information and Computation* 174 (2002), 37–83.

[8] Martín Abadi, Jérémy Planul, and Gordon Plotkin. 2013. Layout Randomization and Nondeterminism. *Electron. Notes Theor. Comput. Sci.* 298 (Nov. 2013), 29–50.

[9] Martín Abadi and Gordon D. Plotkin. 2012. On Protection by Layout Randomization. *ACM Transactions on Information and System Security* 15, Article 8 (July 2012), 29 pages.

[10] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Cătălin Hriţcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. 2018a. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. arXiv:1802.00588. (Feb. 2018). https://arxiv.org/abs/1802.00588

[11] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jeremy Thibault. 2018b. Exploring Robust Property Preservation for Secure Compilation. arXiv:1807.04603. (2018). https://arxiv.org/abs/1807.04603

[12] Samson Abramsky. 1990. The Lazy Lambda Calculus. In *Research Topics in Functional Programming*, David A. Turner (Ed.). Addison-Wesley Longman Publishing Co., Inc., 65–116.

[13] Pedro Adão and Cédric Fournet. 2006. Cryptographically Sound Implementations for Communicating Processes. In *Proceedings of ICALP'06*.

[14] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. 2012. Secure Compilation to Modern Processors. In *CSF 2012*. IEEE, 171–185.

[15] Amal Ahmed. 2004. *Semantics of Types for Mutable State*. Ph.D. Dissertation. Princeton University.

[16] Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *ESOP'06*. 69–83.

[17] Amal Ahmed and Matthias Blume. 2008. Typed Closure Conversion Preserves Observational Equivalence. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. ACM, 157–168.

[18] 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 (ICFP '11)*. ACM, 431–444.

[19] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent Representation Independence. *SIGPLAN Not.* 44 (Jan. 2009), 340–353.

[20] Arthur Azevedo de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hriţcu, David Pichardie, Benjamin C. Pierce, Randy Pollack, and Andrew Tolmach. 2014. A Verified Information-flow Architecture. *SIGPLAN Not.* 49 (Jan. 2014), 165–178.

[21] Ioannis G. Baltopoulos and Andrew D. Gordon. 2009. Secure Compilation of a Multi-Tier Web Language. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation (TLDI '09).* ACM, 27–38.

[22] Gilles Barthe, Tamara Rezk, and Amitabh Basu. 2007. Security Types Preserving Compilation. *Computer Languages, Systems and Structures* 33 (2007), 35–59.

[23] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. 2010. Security of Multithreaded Programs by Compilation. *ACM Transactions on Information and System Security* 13, Article 21 (2010), 32 pages.

[24] Nick Benton and Chung-Kil Hur. 2009. Biorthogonality, step-indexing and compiler correctness. *SIGPLAN Not.* 44 (Aug. 2009), 97–108.

[25] Nick Benton and Chung-kil Hur. 2010. *Realizability and Compositional Compiler Correctness for a Polymorphic Language.* Technical Report. MSR.

[26] William J. Bowman and Amal Ahmed. 2015. Noninterference for free. In *ICFP '15*. 13.

[27] Michele Bugliesi and Marco Giunti. 2007. Secure Implementations of Typed Channel Abstractions. In *POPL '07.*

[28] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-based Addressing. *SIGPLAN Not.* 29 (1994), 319–327.

[29] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, Elena Giachino, and Luca Padovani. 2009. Foundations of Session Types. In *Proceedings PPDP '09.*

[30] Juan Chen, Ravi Chugh, and Nikhil Swamy. 2010. Type-Preserving Compilation of End-To-End Verification of Security Enforcement. In *PLDI '10.* ACM, 412–423.

[31] Andrey Chudnov and David A. Naumann. 2015. Inlined Information Flow Monitoring for JavaScript. In *CCS '15.*

[32] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (Sept. 2010), 1157–1210.

[33] Ricardo Corin, Pierre-Malo Deniélou, Cédric Fournet, Karthikeyan Bhargavan, and James Leifer. 2008. A Secure Compiler for Session Abstractions. *Journal of Computer Security* 16 (2008), 573–636.

[34] Pierre-Louis Curien. 2007. Definability and Full Abstraction. *Electron. Notes Theor. Comput. Sci.* 172 (2007), 301–310.

[35] Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9 (March 1966), 143–155.

[36] Dominique Devriese, Marco Patrignani, Steven Keuchel, and Frank Piessens. 2017. Modular, Fully-Abstract Compilation by Approximate Back-Translation. *Logical Methods in Computer Science* Volume 13, Issue 4 (Oct. 2017).

[37] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Secure Compilation by Approximate Back-Translation. In *POPL 2016.*

[38] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2018. Parametricity versus the Universal Type. In *Proceedings of POPL 2018.*

[39] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. 2015. Architectural Support for Software-Defined Metadata Processing. In *Proceedings ASPLOS '15.*

[40] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and André DeHon. 2014. PUMP: A Programmable Unit for Metadata Processing. In *HASP '14.* 8:1–8:8.

[41] D. Dolev and A. C. Yao. 1981. On the security of public key protocols. In *SFCS.* IEEE Computer Society, 350–357.

[42] Vijay D'silva, Daniel Kroening, and Georg Weissenbacher. 2008. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems* 27 (2008).

[43] Dominic Duggan. 2004. Type-based Cryptographic Operations. *J. Comput. Secur.* 12 (May 2004), 485–550.

[44] Úlfar Erlingsson. 2004. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* Ph.D. Dissertation. Advisor(s) Schneider, Fred B. AAI3114521.

[45] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: a retrospective. In *NSPW'99.*

[46] Úlfar Erlingsson and Fred B. Schneider. 2000. IRM Enforcement of Java Stack Inspection. In *IEEE S&P.* 246–255.

[47] Guglielmo Fachini, Cătălin Hriţcu, Marco Stronati, Ana Nora Evans, Theo Laurent, Arthur Azevedo de Amorim, Benjamin C. Pierce, and Andrew Tolmach. 2018. Formally Secure Compilation of Unsafe Low-Level Components. In *Workshop on Principles of Secure Compilation (PRISC 2018).*

[48] Cédric Fournet and Georges Gonthier. 1996. The Reflexive CHAM and the Join-Calculus. In *POPL.* 372–385.

[49] Cedric Fournet, Andrew Gordon, and Sergio Maffeis. 2007. A Type Discipline for Authorization in Distributed Systems. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF '07).* IEEE Computer Society, 31–48.

[50] Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. 2009. A Security-Preserving Compiler for Distributed Programs: from Information-Flow Policies to Cryptographic Mechanisms. In *Proceedings CCS '09.*

[51] Cédric Fournet and Tamara Rezk. Cryptographically Sound Implementations for Typed Information-flow Security. In *Proceedings POPL '08.*

[52] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully Abstract Compilation to JavaScript. In *Proceedings POPL '13*.
[53] D. Garg, C. Hritcu, M. Patrignani, M. Stronati, and D. Swasey. 2017. Robust Hyperproperty Preservation for Secure Compilation (Extended Abstract). *ArXiv e-prints* (Oct. 2017).
[54] Dan R Ghica and Zaid Al-Zobaidi. 2012. Coherent Minimisation: Towards Efficient Tamper-Proof Compilation. In *5th Interaction and Concurrency Experience Workshop (ICE 2012)*. 16.
[55] Dan R. Ghica and Nikos Tzevelekos. 2012. A System-Level Game Semantics. *Electr. Notes Theo. Comp. Sci.* 286 (2012).
[56] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. 1996. A Secure Environment for Untrusted Helper Applications Confining the Wily Hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium,*.
[57] Andrew D. Gordon and Alan Jeffrey. 2002. Authenticity by Typing for Security Protocols. *J. Comput. Secur.* 11, 4 (2002).
[58] Daniele Gorla and Uwe Nestman. 2014. Full Abstraction for Expressiveness: History, Myths and Facts. *Math Struct Comp Science* (2014).
[59] Nevin Heintze and Jon G. Riecke. 1998. The SLam calculus: programming with secrecy and integrity. In *POPL '98*.
[60] Chung-Kil Hur and Derek Dreyer. 2011. A Kripke Logical Relation Between ML and Assembly. *SIGPLAN Not.* 46 (2011).
[61] Radha Jagadeesan, Corin Pitcher, Julian Rathke, and James Riely. 2011. Local Memory via Layout Randomization. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium (CSF '11)*. IEEE Computer Society, 161–174.
[62] Alan Jeffrey and Julian Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. In *ESOP'05*.
[63] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. 2011. ABS: A Core Language for Abstract Behavioral Specification. In *FMCO'10*. 142–164.
[64] Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, and Benjamin C. Pierce. 2016. Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation. In *IEEE CSF*.
[65] Yannis Juglaret, Catalin Hritcu, Arthur Azevedo de Amorim, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. 2015. Towards a Fully Abstract Compiler Using Micro-Policies: Secure Compilation for Mutually Distrustful Components. *CoRR* abs/1510.00697 (2015). http://arxiv.org/abs/1510.00697
[66] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. 2016. Lightweight verification of separate compilation *(POPL 2016)*. 178–190.
[67] Andrew Kennedy. 2006. Securing the .NET Programming Model. *Theoretical Computer Science* 364 (2006), 311–317.
[68] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *POPL '14*. 179–192.
[69] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *OSDI'14*. USENIX Association, 147–163.
[70] James Laird. 2007. A Fully Abstract Trace Semantics for General References. In *Automata, Languages and Programming*. Lecture Notes in Computer Science, Vol. 4596. Springer Berlin Heidelberg, 667–679.
[71] Adriaan Larmuseau and Dave Clarke. 2015. Formalizing a Secure Foreign Function Interface. In *SEFM 2015 (LNCS)*.
[72] Adriaan Larmuseau, Marco Patrignani, and Dave Clarke. 2014. Operational Semantics for Secure Interoperation. In *Proceedings of PLAS '14*. ACM.
[73] Adriaan Larmuseau, Marco Patrignani, and Dave Clarke. 2016. Implementing a Secure Abstract Machine. In *SAC '16*.
[74] Peeter Laud. 2012. Secure Implementation of Asynchronous Method Calls and Futures. In *Trusted Systems*, Chris J. Mitchell and Allan Tomlinson (Eds.). LNCS, Vol. 7711. Springer Berlin Heidelberg, 25–47.
[75] Christopher League, Zhong Shao, and Valery Trifonov. 2002. Type-Preserving Compilation of Featherweight Java. *ACM Transactions on Programming Languages and Systems* 24 (2002), 112–152.
[76] Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. 42–54.
[77] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *J. Autom. Reasoning* 43, 4 (2009), 363–446.
[78] Andreas Lochbihler. 2010. Verifying a Compiler for Java Threads. In *ESOP'10*. Springer-Verlag, 427–447.
[79] Jacob Matthews and Amal Ahmed. 2008. Parametric polymorphism through run-time sealing or, theorems for low, low prices!. In *Proceedings ESOP'08/ETAPS'08*. Springer-Verlag, 16–31.
[80] Jacob Matthews and Robert Bruce Findler. 2009. Operational Semantics for Multi-Language Programs. *ACM Transactions on Programming Languages and Systems* 31, Article 12 (April 2009), 44 pages.
[81] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB Reduction and Attestation. In *SP '10*. IEEE, 143–158.
[82] Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. 2008. Flicker: an execution infrastructure for TCB minimization. *SIGOPS Oper. Syst. Rev.* 42, Article 24 (April 2008), 14 pages.
[83] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *HASP '13*.

[84] Ricardo Medel, Adriana B. Compagnoni, and Eduardo Bonelli. 2005. A Typed Assembly Language for Non-interference. In *ICTCS 2005*. 360–374.

[85] Robin Milner. 1977. Fully abstract models of typed Œª-calculi. *Theoretical Computer Science* 4, 1 (1977), 1 – 22.

[86] Robin Milner. 1999. *Communicating and mobile systems - the Pi-calculus*. Cambridge University Press.

[87] John C. Mitchell. 1986. Representation Independence and Data Abstraction. In *POPL '86*. ACM, 263–276.

[88] John C. Mitchell. 1993. On Abstraction and the Expressive Power of Programming Languages. *Sci. Comput. Program.* 21, 2 (1993), 141–163.

[89] Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *LICS*. IEEE Press, 14–23.

[90] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems* 21 (1999), 527–568.

[91] Magnus O. Myreen. 2010. Verified Just-in-time Compiler on x86. *SIGPLAN Not.* 45 (Jan. 2010), 107–118.

[92] George C. Necula. 1997. Proof-carrying code. In *POPL '97*. ACM, 106–119.

[93] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. 2015. Pilsner: A Compositionally Verified Compiler for a Higher-order Imperative Language. In *ICFP 2015*. ACM, 166–178.

[94] Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation via Universal Embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP '16)*. ACM.

[95] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. 2013. Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base. In *Proceedings of the 22nd USENIX conference on Security symposium*.

[96] Joachim Parrow. 2008. Expressiveness of Process Algebras. *Electr. Notes Theo. Comp. Sci.* 209 (2008), 173 – 186.

[97] Joachim Parrow. 2014. General conditions for Full Abstraction. *Math Struct Comp Science* (2014).

[98] Marco Patrignani. 2015. *The Tome of Secure Compilation: Fully Abstract Compilation to Protected Modules Architectures*. Ph.D. Dissertation. KU Leuven.

[99] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure Compilation to Protected Module Architectures. *ACM Trans. Program. Lang. Syst.* 37, Article 6 (April 2015), 50 pages.

[100] Marco Patrignani and Dave Clarke. 2015. Fully abstract trace semantics for protected module architectures. *Computer Languages, Systems & Structures* 42, 0 (2015), 22 – 45.

[101] Marco Patrignani, Dave Clarke, and Frank Piessens. 2013. Secure Compilation of Object-Oriented Components to Protected Module Architectures. In *Proceedings APLAS'13*. 176–191.

[102] Marco Patrignani, Dominique Devriese, and Frank Piessens. 2016. On Modular and Fully Abstract Compilation. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF 2016)*.

[103] Marco Patrignani and Deepak Garg. 2017. Secure Compilation as Hyperproperty Preservation. In *CSF'17*.

[104] Marco Patrignani and Deepak Garg. 2018. Robustly Safe Compilation or, Efficient, Provably Secure Compilation. *CoRR* abs/1804.00489 (2018). http://arxiv.org/abs/1804.00489

[105] Daniel Patterson and Amal Ahmed. 2018. On Compositional Compiler Correctness and Fully Abstract Compilation. In *Workshop on Principles of Secure Compilation (PRISC 2018)*.

[106] James T. Perconti and Amal Ahmed. 2014. Verifying an Open Compiler Using Multi-language Semantics. In *ESOP (Lecture Notes in Computer Science)*, Vol. 8410. 128–148.

[107] Benjamin Pierce. 2002. *Types and Programming Languages*. MIT Press.

[108] Benjamin Pierce and Eijiro Sumii. 2000. Relating Cryptography and Polymorphism. (2000). manuscript.

[109] Frank Piessens, Dominique Devriese, Jan Tobias Muhlberg, and Raoul Strackx. 2016. Security guarantees for the execution infrastructure of software applications. In *IEEE SecDev 2016*.

[110] Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *TCS* 5 (1977), 223–255.

[111] Jon G. Riecke. 1991. Fully Abstract Translations Between Functional Languages. In *POPL '91*.

[112] Eike Ritter and Andrew M. Pitts. 1995. A Fully Abstract Translation Between a Lambda-Calculus with Reference Types and Standard ML. In *Proceedings of TLCA '95*. Springer-Verlag, 397–413.

[113] Davide Sangiorgi. 2012. *Introduction to Bisimulation and Coinduction*. Cambridge University Press.

[114] Davide Sangiorgi and David Walker. 2001. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press.

[115] Manfred Schmidt-Schauß, David Sabel, Joachim Niehren, and Jan Schwinghammer. 2015. Observational program calculi and the correctness of translations. *Theoretical Computer Science* 577 (2015), 98 – 124.

[116] Fred B. Schneider. 2000. Enforceable security policies. *ACM TISSEC* 3, 1 (2000), 30–50.

[117] Jaroslav Sevcik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. 2011. Relaxed-memory Concurrency and Verified Compilation. *SIGPLAN Not.* 46 (Jan. 2011), 43–54.

[118] Naokata Shikuma and Atsushi Igarashi. 2007. Proving Noninterference by a Fully Complete Translation to the Simply Typed λ-Calculus. In *Advances in Computer Science - ASIAN 2006*. Vol. 4435. 301–315.

[119] Jeremy G. Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP*. 2–27.

[120] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram Rajamani, Sanjit A. Seshia, and Kapil Vaswani. 2016. A

Design and Verification Methodology for Secure Isolated Regions. *SIGPLAN Not.* 51, 6 (June 2016), 665–681.

[121] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *POPL '15*. ACM, 275–287.

[122] Raoul Strackx and Frank Piessens. 2012. Fides: selectively hardening software application components against kernel-level or process-level malware. In *Proceedings CCS '12*. ACM, 2–13.

[123] Eijiro Sumii and Benjamin C. Pierce. 2004. A Bisimulation for Dynamic Sealing. In *POPL*. 161–172.

[124] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure Distributed Programming with Value-Dependent Types. In *ICFP '11*. ACM, 266–278.

[125] Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. 2014. Gradual Typing Embedded Securely in JavaScript. *SIGPLAN Not.* 49, 1 (Jan. 2014), 425–437.

[126] W. W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic* 32 (1967).

[127] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. 1996. TIL: A Type-Directed Optimizing Compiler for ML. 181–192.

[128] Stelios Tsampas, Akram El-Korashy, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. 2017. Towards Automatic Compartmentalization of C Programs on Capability Machines. In *FCS 2017* .

[129] Stephen Tse and Steve Zdancewic. 2004. Translating Dependency into Parametricity. *SIGPLAN Not.* 39 (2004), 115–125.

[130] Stephen Tse and Steve Zdancewic. 2007. Run-time principals in information-flow type systems. *ACM Trans. Program. Lang. Syst.* 30, Article 6 (Nov. 2007).

[131] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security* 4 (1996), 167–187. Issue 2-3.

[132] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. *SIGOPS Oper. Syst. Rev.* 27, 5 (Dec. 1993), 203–216.

[133] David Walker. 2000. A Type System for Expressive Security Policies. In *POPL '00*.

[134] Mitchell Wand. 1998. The Theory of Fexprs is Trivial. *Lisp and Symbolic Computation* 10 (1998), 189–199.

[135] Xiaoyun Wang and Hongbo Yu. 2005. How to Break MD5 and Other Hash Functions. In *EUROCRYPT'05*. 19–35.

[136] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX.. In *USENIX Security Symposium*. USENIX Association, 29–46.

[137] Yannick Welsch and Arnd Poetzsch-Heffter. 2013. A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. *Science of Computer Programming* -, 0 (2013), –.

[138] 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 *ISCA '14*. 457–468.

[139] Dachuan Yu and Nayeem Islam. 2006. A Typed Assembly Language for Confidentiality. In *ESOP'06*.

[140] Stephan A. Zdancewic. 2002. *Programming Languages for Information Security*. Ph.D. Dissertation. Cornell University.

[141] Bin Zeng, Gang Tan, and Greg Morrisett. 2011. Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of CCS '11*. ACM, 29–40.

[142] Jianzhou Zhao, Qi Zhang, and Steve Zdancewic. 2010. Relational Parametricity for a Polymorphic Linear Lambda Calculus. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010*. 344–359.

A ADDITIONAL PROGRAM EQUIVALENCES FOR EXPRESSING SECURITY PROPERTIES

This section presents a probabilistic variant of contextual equivalence (Appendix A.1) Then it formalises alternatives to contextual equivalence, which have been devised since direct proofs of contextual equivalence are usually intractable. Specifically, this section discusses bisimilarity (Appendix A.2), trace semantics (Appendix A.3), and logical relations (Appendix A.4).

A.1 Probabilistic Contextual Equivalence

Contextual equivalence is not enough for languages with a notion of randomisation or cryptography, as the universal quantification over contexts models a too powerful attacker. If a strong cryptographic function is used to encrypt data, most contexts will not be able to decrypt that data and retrieve the contents. So, by considering all contexts, there is one that by definition has the right key to decrypt the data. That context can observe some behavioural difference that we know can be done only by breaking the cryptographic function. As the used cryptographic functions are generally assumed to be unbreakable, this model violates a key assumption of the system. Additionally, when keys are computable (e.g., they could be bitstrings) the context can just exhaust the search space and try all keys. However, in practice very large keys are used and these contexts would run for too long to calculate the right key. As these contexts also violate a key assumption of the system, often contexts are limited to be *polynomial* in the size of the key, so they cannot exhaust the key search space [50, 51].

In order to filter out contexts that violate these key assumptions, probabilistic contextual equivalence can be used. With probabilistic contextual equivalence, two programs are equivalent if they are equivalent up to a certain probability (i.e., in *the majority* of contexts) for polynomial contexts.[6] In the example above, contexts that have ways to break the cryptographic functions must not qualify as the majority of contexts, so they can be safely discarded when considering program equivalence.

To model the semantics of randomisation, oracles can be made part of the semantics [8]; they provide infinite streams of random values for the randomisation function to use.

Examples A.1 to A.2 describe the usage of oracles. Example A.3 discusses how guessing (either random numbers or cryptographic keys) affects the definition of contextual equivalence.

Example A.1 (Obvious equivalences). With an oracle, obvious equivalences must be respected. For example, a program P_r that returns a random value must be equivalent to itself.

```
1 public getRandom(){    //P_r
2   return rand.next();
3 }
```

In this case, for any oracle O, there exists an oracle (the same oracle O) that makes the behaviour of two copies of P_r coincide.

P_r is also equivalent to the following snippet P_{rr}, which generates two random numbers and only returns the second.

```
1 public getRandom(){    //P_rr
2   rand.next();
3   return rand.next();
4 }
```

[6] We state this together with the assumption on polynomial contexts, though in cases when secrets are not computable, this condition can be dropped [9, 61].

The oracle that must be used with P_{rr} is one that has all elements of O interleaved with other elements.

Example A.2 (Obvious inequivalences). Consider a program that returns two random numbers and a program that returns the same random number twice.

```
1 public getRandom(){
2   var x = rand.next();
3   return new Pair(x, rand.next());
4 }
```

```
1 public getRandom(){
2   var x = rand.next();
3   return new Pair(x, x);
4 }
```

For any oracle used with the right-hand side snippet, the oracle that contains the same value twice will make the left-hand side snippet equivalent to it. However, there is no oracle that will make the opposite true: given an oracle for the left-hand side snippet, there is no way to build one that will make the right-hand side one behave the same. In fact, an oracle that provides two different values will make the left-hand side return a pair with two different values, and this cannot be done by the right-hand side snippet.

We can thus conclude that if any two components are equivalent, then for any oracle used with the first one there must exist one for the second such that they exhibit the same behaviour *and vice versa.*

Example A.3 (Guessing). Consider two programs that store a random number and then receive input from the external code via variable `guess`. If that input matches the random number, one returns 0 and the other returns 1, otherwise they both return 2.

```
1 public getRandom(){
2   var x = rand.next();
3   var guess = callback();
4   if (guess == x)
5     return 0;
6   return 2;
7 }
```

```
1 public getRandom(){
2   var x = rand.next();
3   var guess = callback();
4   if (guess == x)
5     return 1;
6   return 2;
7 }
```

If the random function is strong enough, the context has very little chance of telling these programs apart. As already said however, the contexts considered in the definition of contextual equivalence are universally quantified. Thus there is also a context that differentiates between the two programs by guessing the number, so the definition of probabilistic contextual equivalence needs to incorporate probability concepts.

The probabilistic notion of contextual equivalence states that two programs are equivalent if, when the context is polynomial, they behave the same to a certain probability. Denote the probability of a certain event with $\mathbb{P}(\cdot)$ and let σ range over the [0,1] interval. We indicate a context being polynomial given the size s of intended secrets as $\vdash \mathbb{C} : \text{poly } s$.

Definition A.4 (Contextual preorder).

$$P_1 \sqsubseteq_\sigma P_2 \triangleq \mathbb{P}(\forall \mathbb{C}. \vdash \mathbb{C} : \text{poly } s,\ O_1,\ \exists O_2.\ \mathbb{C}[P_1, O_1]\Uparrow \iff \mathbb{C}[P_2, O_2]\Uparrow) > \sigma$$

Probabilistic contextual equivalence can be defined as following.

Definition A.5 (Probabilistic contextual equivalence [9]). $P_1 \simeq_{\text{ctx}\,\sigma} P_2 \triangleq P_1 \sqsubseteq_\sigma P_2$ and $P_2 \sqsubseteq_\sigma P_1$.

A.2 Bisimilarity

Bisimilarity has been frequently employed in the field of process algebra in order to state when two processes exhibit the same behaviour [113]. It has also been frequently used to prove compilation schemes for process calculi secure [5–7, 13, 27, 43, 74]. Intuitively, two processes have the same

behaviour when they perform the same "*actions*" and become new processes that continue to have the same behaviour.

The notion of bisimilarity (Definition A.8) relies on the concept of a labelled transition system (LTS), which is used to provide a model of the process.

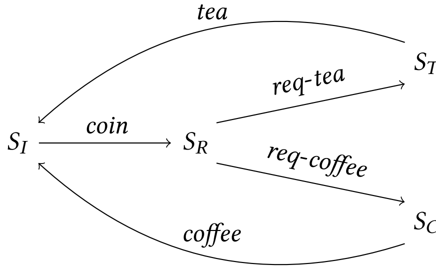
Definition A.6 (LTS). A labelled transition system is a triple $(S, \Lambda, \rightarrow)$ where S is a set of states, Λ is a set of labels and $\rightarrow \subseteq S \times \Lambda \times S$ is a ternary relation of labelled transitions.

A transition between two states S_1 and $S_2 \in S$ on a label $\lambda \in \Lambda$ is indicated as $S_1 \xrightarrow{\lambda} S_2$. Labels represent what an entity external to S can observe from the states of S, as S performs computations; labels often concern inputs and outputs, as presented in the following example.

Example A.7 (LTS [113]). Consider the LTS of a vending machine that produces tea or coffee after receiving coins, after the appropriate request is made. It is formalised as

$$(\{S_I, S_R, S_T, S_C\}, \{\mathit{coin}, \mathit{req\text{-}tea}, \mathit{req\text{-}coffee}, \mathit{tea}, \mathit{coffee}\},$$
$$\{S_I \xrightarrow{\mathit{coin}} S_R, S_R \xrightarrow{\mathit{req\text{-}tea}} S_T, S_R \xrightarrow{\mathit{req\text{-}coffee}} S_C, S_T \xrightarrow{\mathit{tea}} S_I, S_C \xrightarrow{\mathit{coffee}} S_I\})$$

and it is depicted below.



S_I models the state of a vending machine waiting for input, *coin* expresses the user input and S_R models the state in which the machine waits for the type of product to deliver. Based on the two different inputs from S_R, the machine can reach two states: S_T and S_C, the states where the machine produces tea and coffee, respectively. Then, both S_T and S_C transition back to S_I, labelled with the output it provides to the user: *tea* or *coffee*.

Often, labels are also equipped with decorations that indicate the direction of the action: ! is an observable produced from the program, ? is an observable received by it. The aforementioned transitions can thus be decorated as follows $S_I \xrightarrow{\mathit{coin}?} S_R$, $S_R \xrightarrow{\mathit{req\text{-}coffee}?} S_C$ and $S_T \xrightarrow{\mathit{tea}!} S_I$.

Definition A.8 (Bisimilarity). Given a LTS $(S, \Lambda, \rightarrow)$, a relation $\mathcal{R} \subseteq S \times S$ is a bisimulation if, for any pair $(S_1, S_2) \in \mathcal{R}$, for all $\lambda \in \Lambda$, the following holds:

(1) for all S_1' such that $S_1 \xrightarrow{\lambda} S_1'$, there exists S_2' such that $S_2 \xrightarrow{\lambda} S_2'$ and $(S_1', S_2') \in \mathcal{R}$;
(2) for all S_2' such that $S_2 \xrightarrow{\lambda} S_2'$, there exists S_1' such that $S_1 \xrightarrow{\lambda} S_1'$ and $(S_1', S_2') \in \mathcal{R}$.

Bisimilarity is the union of all bisimulations.

Two components P and Q are thus bisimilar if there exists a bisimulation relation $\mathcal{R}$ such that $P \mathrel{\mathcal{R}} Q$.

A more permissive variant of bisimilarity that is often used for program equivalence is weak bisimilarity. Its definition is the same as that of bisimilarity except that $\overset{\lambda}{\Longrightarrow}$ is used in place of

$\xrightarrow{\lambda}$. Relation $\xRightarrow{\lambda}$ abstracts away from transitions that model internal computations (often called "silent" transitions and defined as a τ). Formally $\xRightarrow{\lambda}$ is defined as $\xrightarrow{\tau}^* \xrightarrow{\lambda} \xrightarrow{\tau}^*$. Thus, programs `x = 0 ; x += 1` and `x = 1` are weakly bisimilar (if `x` is not observable) even though they perform a different number of internal steps; while they are not bisimilar according to Definition A.8.

When working with bisimulation in place of contextual equivalence for secure compilation, labels model what the external code (i.e., the context in contextual equivalence) can observe about a program. The external code is modelled as a black box that triggers transitions. So, bisimilarity abstracts from the behaviour of the attacker but it captures the reaction of the component to certain actions of the attacker. This abstraction is the great advantage of bisimulation over contextual equivalence. Thus it is crucial, when replacing contextual equivalence with bisimulation, that all possible attacker behaviour is captured by the labels, so as to have a precise characterisation of what the attacker can do. Finally, the bisimulation proof technique is more amenable to proofs than the unstructured induction over all possible reductions that is offered by contextual equivalence.

Example A.9 (Security properties via bisimilarity: memory size). Consider the code snippets of Figure 3, the following LTSs describe their behaviour in a language with unbounded memory size (Figure 5). States *ext* and *cb* indicate that control is outside the snippets. State $S0$ indicates the beginning of the code snippets, states Oi indicate the state where the i-th object is allocated, and states S indicate the end of the code snippets. States are given a subscript l or r to indicate whether they describe the behaviour of the left-hand side or of the right-hand side snippet.

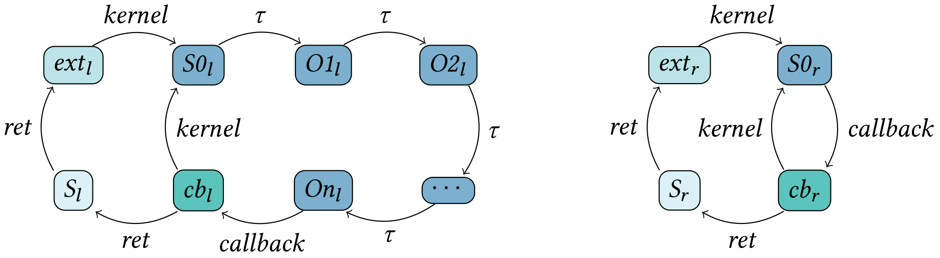


Fig. 5. LTSs describing weakly bisimilar snippets of Figure 3. Nodes coloured with the same colour are bisimilar.

To indicate that the code snippets are weakly bisimilar, the following bisimulation relation can be built:

$$\mathcal{R}_{mem} = \{(ext_l, ext_r), (S0_l, S0_r), (O1_l, S0_r), (O2_l, S0_r), \cdots, (On_l, S0_r), (cb_l, cb_r), (S_l, S_r)\}$$

Relation $\mathcal{R}_{mem}$ is a weak bisimulation as it satisfies Definition A.8 when relation $\xRightarrow{\lambda}$ is used. In fact, for all pairs in $\mathcal{R}_{mem}$, the first element can perform a possibly-empty sequence of τs followed by an action λ ending up in a state. The second element can also perform a possibly-empty sequence of τs followed by the same action λ and end up in a state that is related to the previous one in $\mathcal{R}_{mem}$. And vice versa.

A.3 Trace Equivalence

Like bisimilarity, trace equivalence has been used successfully to replace contextual equivalence in secure compilation work [61, 64, 99, 102].

Given the LTS of a component P, the behaviour of P can be described with sequences of labels that can be generated according to the LTS. These sequences of labels, denoted with $\overline{\lambda}$, are called

traces. The trace semantics of a program is the set of of all traces that can be associated with a component, based on its labelled transition system.

Definition A.10 (Trace semantics). The trace semantics of a component P is indicated as $\text{Traces}(P)$. $\text{Traces}(P) = \{\overline{\lambda} \mid \exists P'.P \xRightarrow{\overline{\lambda}} P'\}$.

Two programs are trace equivalent, denoted with $P_1 \stackrel{\text{T}}{=} P_2$, if their traces coincide.

Definition A.11 (Trace equivalence). $P_1 \stackrel{\text{T}}{=} P_2 \triangleq \text{Traces}(P_1) = \text{Traces}(P_2)$.

The same considerations made for bisimulation apply also here: since labels capture attacker actions, it is crucial that all that the attackers can observe is captured in the labels. This is captured by a proof stating that trace equivalence is as precise as contextual equivalence [62, 100].

Trace equivalence also comes with a somewhat simpler proof technique than the one of contextual equivalence. Traces are often defined inductively, so they are particularly amenable as a proof technique as they enable a neat, structural argument for proofs adopting them.

Before seeing an example of trace semantics at work for defining security properties (Example A.13), Example A.12 informally presents a trace semantics for a Java-like program. The example is reminiscent of the work of Jeffrey and Rathke [62]. The interested reader is referred to work on fully-abstract trace semantics for a more in-depth discussion of the subject [62, 70, 100, 137].

Example A.12 (Trace semantics for a Java-like program). Consider the class of the following snippet to be the component whose trace semantics we are interested in. The context this component interacts with is the `Log` class, which we assume to just implement function `addLogEntry()`. The other code does not need to be specified, as contexts are for contextual equivalence; the trace semantics needs to capture only its power in terms of the actions it can perform.

```
1  import Log;
2
3  public class Account{
4    private int balance = 0;
5
6    public int deposit( int amount ) {
7      this.balance += amount;
8      return balance;
9    }
10   public void logUsage( String user ) {
11     log.addLogEntry( user );
12     return unit;
13   }
14 }
15 public Account acc;
```

Listing 7. Example of Java source code.

A trace semantics for such a Java-like language needs to capture how classes interact with each other, i.e., method calls and returns (assume all fields are private for simplicity). The labels below indicate when they are generated. We use a convention of decorating actions that are performed

by the component with a "!", while ?-decorated actions are performed by the context.

`log` calling method `deposit` on line 6	`call acc.deposit( n )?`
executing line 8	`ret n!`
`log` calling method `logUsage` on line 10	`call acc.logUsage( u )?`
executing line 11	`call log.addLogEntry( u )!`
`log` returning to line 12	`ret unit?`
executing line 12	`ret unit!`

A trace will therefore be a concatenation of those labels in a way that the semantics allows, e.g.,:

`call acc.deposit( n )? ret n!`

`call acc.deposit( n )? ret n! call acc.deposit( m )? ret n+m!`

`call acc.logUsage( u )? log.addLogEntry( u )! ret unit? ret unit!`

So `call deposit( n )?` can be followed by `ret n!` but not from `call addLogEntry( u )!`.

Example A.13 (Security properties via traces: integrity). Consider the code snippets of Figure 2 and a trace semantics describing the behaviour of the code snippets based on labels for calls and returns as in Example A.12. If the behaviour of those code snippets can be described by (a concatenation of) the following trace, for any possible value v, then the programs are equivalent:

`call proxy(callback())? call callback()! ret` v`? ret 0!`

On the other hand, if there exists a trace that belongs to the trace semantics of a program but not to the trace semantics of the other, then they are not equivalent. For example, the following trace can be a valid trace of the left-hand side snippet but not of the right-hand side one.

`call proxy(callback())? call callback()! ret` v`? ret 1!`

Consider a language with pointers, which can be used to modify the contents of the stack (e.g., C), this trace is a valid description of the behaviour of the left-hand side snippet. This trace highlights a difference in the behaviour of the two snippets. As those snippets modelled integrity concerns, this trace captures a violation of the integrity property of the `secret` variable.

A.4 Logical Relations

Logical relations are a proof technique that has been used to prove when programs have nontrivial properties such as normalisation (of the simply-typed λ-calculus) [126], noninterference [59], equivalence of modules [19, 87] and compiler correctness [24, 60, 106]. Additionally, it has frequently been used to prove compiler security [17, 18, 26, 37, 94, 129].

Logical relations can be set up to prove contextual equivalence of components written in the same language. Once such a logical relation is defined, it must be proved sound (and perhaps also complete) with respect to contextual equivalence. But logical relations can also be used to formally express a desired notion of equivalence between components from two different languages—we refer to these as *cross-language* logical relations. Note that the peculiar thing about cross-language logical relation is that we cannot prove them sound with respect to any notion of contextual equivalence since there is no such notion that spans the two different languages. Below, to illustrate how logical relations are defined, we present a cross-language logical relation between the source and (highly idealized) target language of a compiler. Some of the aforementioned papers on secure compilation use cross-language logical relations, but others use relations between terms of the same language or between terms of multi-languages; we discuss these details later in the paper.

Example A.14 (Cross-language logical relations). Consider the simply-typed λ-calculus with booleans as source language. The types of the language conform to the following grammar $\tau ::= \texttt{Bool} \mid \tau \to \tau$, the syntax of values and expressions is omitted for the sake of brevity. Reduction steps are indicated with $\twoheadrightarrow$, their reflexive, transitive closure is denoted with $\twoheadrightarrow^*$; the capture-avoiding substitution of a value $\mathbf{v}$ for variable $\mathbf{x}$ in $\mathbf{t}$ is denoted as $\mathbf{t}[\mathbf{v}/\mathbf{x}]$.

Consider an untyped λ-calculus with booleans as the target language.

The logical relation that states whether two terms from the two languages are contextually equivalent is defined below.

$$\mathcal{V}[\![\texttt{Bool}]\!] \stackrel{\text{def}}{=} \{(k, \mathbf{v}, \mathsf{v}) \mid (\mathbf{v} \equiv \texttt{true} \text{ and } \mathsf{v} \equiv \texttt{true}) \text{ or } (\mathbf{v} \equiv \texttt{false} \text{ and } \mathsf{v} \equiv \texttt{false})\}$$

$$\mathcal{V}[\![\tau' \to \tau]\!] \stackrel{\text{def}}{=} \left\{ (k, \mathbf{v}, \mathsf{v}) \;\middle|\; \begin{array}{l} (\mathbf{v}, \mathsf{v}) \in \texttt{oftype}(\tau' \to \tau) \\ \text{and } \exists \mathbf{t}, \mathsf{t}. \mathbf{v} \equiv \lambda \mathbf{x} : \tau'. \mathbf{t} \text{ and } \mathsf{v} \equiv \lambda \mathsf{x}. \mathsf{t} \\ \text{and } \forall k' \sqsupset_{\triangleright} k, (k', \mathbf{v}', \mathsf{v}') \in \mathcal{V}[\![\tau']\!]. (k', \mathbf{t}[\mathbf{v}'/\mathbf{x}], \mathsf{t}[\mathsf{v}'/\mathsf{x}]) \in \mathcal{E}[\![\tau]\!] \end{array} \right\}$$

$$\mathcal{K}[\![\tau]\!] \stackrel{\text{def}}{=} \{(k, \mathbb{C}_{\mathsf{S}}, \mathbb{C}_{\mathsf{T}}) \mid \forall k' \leq k, (k', \mathbb{C}_{\mathsf{S}}[\mathbf{v}], \mathbb{C}_{\mathsf{T}}[\mathsf{v}]) \in \mathcal{OBS}\}$$

$$\mathcal{E}[\![\tau]\!] \stackrel{\text{def}}{=} \{(k, \mathbf{t}, \mathsf{t}) \mid \forall (k, \mathbb{C}_{\mathsf{S}}, \mathbb{C}_{\mathsf{T}}) \in \mathcal{K}[\![\tau]\!], (k, \mathbb{C}_{\mathsf{S}}[\mathbf{t}], \mathbb{C}_{\mathsf{T}}[\mathsf{t}]) \in \mathcal{OBS}\}$$

$$\mathcal{G}[\![\emptyset]\!] \stackrel{\text{def}}{=} \{(k, \emptyset, \emptyset)\}$$

$$\mathcal{G}[\![\Gamma, (\mathbf{x} : \tau)]\!] \stackrel{\text{def}}{=} \{(k, \gamma[\mathbf{v}/\mathbf{x}], \gamma[\mathsf{v}/\mathsf{x}]) \mid (k, \gamma, \gamma) \in \mathcal{G}[\![\Gamma]\!] \text{ and } (k, \mathbf{v}, \mathsf{v}) \in \mathcal{V}[\![\tau]\!]\}$$

$$\mathcal{OBS} \stackrel{\text{def}}{=} \{(k, \mathbf{t}, \mathsf{t}) \mid (\mathbf{t}, \mathsf{t} \text{ terminate}) \vee (\exists \mathbf{t}', \mathsf{t}'. \mathbf{t} \twoheadrightarrow^{k} \mathbf{t}', \mathsf{t} \twoheadrightarrow^{k} \mathsf{t}')\}$$

This logical relation is indexed by a natural number k that, in essence, represents the number of remaining steps for which the terms are related—after k steps, there is no guarantee that the terms are related. This influences the notion of observation ($\mathcal{OBS}$), which says that either the two terms converge or they are still running after k steps have elapsed. Step-indexed logical relations are an instance of Kripke logical relations (as noted by Ahmed [15]). Here the world is comprised solely of the step index, which describes the number of steps still available in the current world. For languages with more advanced features, such as dynamically allocated mutable memory, the logical relation is indexed by worlds that keep track of the remaining number of steps as well as sophisticated relational invariants on the memory locations of the two programs being related.

Two booleans are related when they are the "same" boolean. Two functions are related when given related argument, they evaluate to related terms. Also, two functions are checked to be in the right syntactic form by function $\texttt{oftype}(\tau' \to \tau)$. Two contexts are related when supplied related values, they are observationally-equivalent terms. Two expressions are related when we can embed them in related contexts and this results in observationally-equivalent terms. Finally, two substitutions are related when they replace related variables (i.e., with the same name) with related values.

With this relation, we can define when two open terms are related as follows:

$$\Gamma \vdash \mathbf{t} \propto \mathsf{t} : \tau \stackrel{\text{def}}{=} \forall k, \Gamma \vdash \mathbf{t} \propto_k \mathsf{t} : \tau$$

$$\Gamma \vdash \mathbf{t} \propto_k \mathsf{t} : \tau \stackrel{\text{def}}{=} \forall j \leq k, \forall (j, \gamma, \gamma) \in \mathcal{G}[\![\Gamma]\!], (j, \mathbf{t}\gamma, \mathsf{t}\gamma) \in \mathcal{E}[\![\tau]\!]$$

B PROOF TECHNIQUES FOR EQUIVALENCE-PRESERVING COMPILATION

Most of the secure compilation results that are discussed in Section 5 establish secure compilation by proving fully abstract compilation, so we now focus on proof techniques to prove the more difficult part of compiler full abstraction, namely equivalence preservation. Existing techniques all rely on the general concept of *back-translation of a target context* (Appendix B.1). There are two

broad categories of back-translation techniques: the first is based on partial evaluation of the target context (Appendix B.2), while the second is based on an embedding of the target context into the source language (Appendix B.3).

B.1 Back-translation, Informally

As stated in Section 4.3.1, preservation of contextual equivalence is stated as follows:

$$\mathit{Preservation} = \forall \mathbf{P_1}, \mathbf{P_2} \in \mathbf{S}, \mathbf{P_1} \simeq_{ctx} \mathbf{P_2} \Rightarrow [\![\mathbf{P_1}]\!]^{\mathbf{S}}_{\mathsf{T}} \simeq_{ctx} [\![\mathbf{P_2}]\!]^{\mathbf{S}}_{\mathsf{T}}$$

For the sake of simplicity, this statement is re-stated in contrapositive form, as it turns the universal quantifiers hidden inside the definition of contextual equivalence into existentials:

$$\mathit{Preservation}\ (\mathit{contrapositive}) = \forall \mathbf{P_1}, \mathbf{P_2} \in \mathbf{S}, [\![\mathbf{P_1}]\!]^{\mathbf{S}}_{\mathsf{T}} \not\simeq_{ctx} [\![\mathbf{P_2}]\!]^{\mathbf{S}}_{\mathsf{T}} \Rightarrow \mathbf{P_1} \not\simeq_{ctx} \mathbf{P_2}$$

To prove this statement, it is sufficient to consider two source programs whose compiled counterparts are different and build a source context that differentiates those source programs.

The following example gives an informal intuition of what it means to generate the differentiating source-level context.

Example B.1 (Informal context back translation). Consider these two STLC terms:

$$\mathbf{t_l} \stackrel{\text{def}}{=} \lambda\mathbf{x} : \mathbf{Bool}.\mathtt{false} \qquad\qquad \mathbf{t_r} \stackrel{\text{def}}{=} \lambda\mathbf{x} : \mathbf{Bool}.\mathtt{true}$$

Assume they are compiled to an untyped λ-calculus simply by erasing their types as follows:

$$[\![\mathbf{t_l}]\!]^{\mathbf{S}}_{\mathsf{T}} \stackrel{\text{def}}{=} \lambda \mathsf{x}.\mathtt{false} \qquad\qquad [\![\mathbf{t_r}]\!]^{\mathbf{S}}_{\mathsf{T}} \stackrel{\text{def}}{=} \lambda \mathsf{x}.\mathtt{true}$$

The compiled programs are contextually-inequivalent; the following context in fact terminates when it interacts with $[\![\mathbf{t_l}]\!]^{\mathbf{S}}_{\mathsf{T}}$ and it diverges when it interacts with $[\![\mathbf{t_r}]\!]^{\mathbf{S}}_{\mathsf{T}}$:

$$\mathbb{C}_{\mathsf{T}} \stackrel{\text{def}}{=} \mathtt{if}\ ([\cdot]\ \mathtt{true})\ \mathtt{then}\ \mathtt{unit}\ \mathtt{else}\ \Omega$$

$$\begin{array}{ll} \mathbb{C}_{\mathsf{T}}[[\![\mathbf{t_l}]\!]^{\mathbf{S}}_{\mathsf{T}}] & \qquad \mathbb{C}_{\mathsf{T}}[[\![\mathbf{t_r}]\!]^{\mathbf{S}}_{\mathsf{T}}] \\ \to \mathtt{if}\ (\mathtt{false})\ \mathtt{then}\ \mathtt{unit}\ \mathtt{else}\ \Omega & \qquad \to \mathtt{if}\ (\mathtt{true})\ \mathtt{then}\ \mathtt{unit}\ \mathtt{else}\ \Omega \\ \to \Omega \to^{*} \Omega \cdots & \qquad \to \mathtt{unit} \end{array}$$

In this case we can generate a source-level context that also witnesses the difference in behaviour between $\mathbf{t_l}$ and $\mathbf{t_r}$ as follows:

$$\mathbb{C}_{\mathbf{S}} \stackrel{\text{def}}{=} \mathtt{if}\ ([\cdot]\ \mathtt{true})\ \mathtt{then}\ \mathtt{unit}\ \mathtt{else}\ \Omega$$

To prove preservation of contextual equivalence we need to prove that such a back-translated context $\mathbb{C}_{\mathbf{S}}$ exists *for any* $\mathbb{C}_{\mathsf{T}}$ that can differentiate $[\![\mathbf{t_l}]\!]^{\mathbf{S}}_{\mathsf{T}}$ and $[\![\mathbf{t_r}]\!]^{\mathbf{S}}_{\mathsf{T}}$.

B.2 Back-translation by Partial Evaluation

The first technique to back-translate target contexts is partial evaluation. This technique rests on the idea that only the interaction between the context and the compiled terms are of interest, and it is precisely those interactions that the back-translated context must capture. Whatever reductions the target context was doing internally does not matter.

Two ways exist to make sure that these context-internal reductions do not matter: considering only normal-form contexts (Appendix B.2.1) and relying on target-level trace semantics (Appendix B.2.2).

B.2.1 Type-directed back-translation by partial evaluation. Ahmed and Blume [18], and later Bowman and Ahmed [26] both make use of a type-directed back-translation by partial evaluation where a key idea is that only subterms of translation type need to be back-translated. The technique requires a complex well-foundedness argument as the back-translation is not inductively defined.

Back-translation by partial evaluation relies on the fact that in some languages, if a term has translation type, then any uses of non-translation type in a subterm are inessential and can therefore be eliminated by partial evaluation. However, this is not a realistic assumption for non-terminating languages or languages with information hiding. For instance, a diverging program may do arbitrary computation using non-translation type that cannot be eliminated by partial evaluation. Furthermore, this technique alone will not work when both languages have state or existential types since programs of target type can use values of non-translation type in their closure or as the existential witness. For this reason, back translation by partial evaluation has only been applied to purely functional, normalizing languages. However, it has the benefit of being fairly systematic when it is applicable.

B.2.2 Back-translation via Traces. This kind of back-translation can be done when target-level trace equivalence is used to replace target-level contextual equivalence, and the preservation statement is expressed as follows:

$$\textit{Preservation (contrapositive)} = \forall \mathbf{P_1}, \mathbf{P_2} \in \mathsf{S}, [\![\mathbf{P_1}]\!]^{\mathsf{S}}_{\mathsf{T}} \not\stackrel{T}{=} [\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}} \Rightarrow \mathbf{P_1} \not\simeq_{ctx} \mathbf{P_2}$$

Since $[\![\mathbf{P_1}]\!]^{\mathsf{S}}_{\mathsf{T}} \not\stackrel{T}{=} [\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}}$ we know that the traces describing the behaviour of $[\![\mathbf{P_1}]\!]^{\mathsf{S}}_{\mathsf{T}}$ is different from the traces of $[\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}}$. So there is a single trace $\overline{\lambda}_{diff}$ that is in the trace semantics of $[\![\mathbf{P_1}]\!]^{\mathsf{S}}_{\mathsf{T}}$ and not in the trace semantics of $[\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}}$. By definition, $\overline{\lambda}_{diff} \equiv \overline{\lambda}\lambda'!$ and there exist a trace in the traces of $[\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}}$ that has the form $\overline{\lambda}\lambda''!$ with $\lambda'! \neq \lambda''!$. Trace $\overline{\lambda}$ is called the *common prefix* (note that it can be as small as a single action) that accounts for possibly-equivalent behaviour of $[\![\mathbf{P_1}]\!]^{\mathsf{S}}_{\mathsf{T}}$ and $[\![\mathbf{P_2}]\!]^{\mathsf{S}}_{\mathsf{T}}$. Single actions $\lambda'!$ and $\lambda''!$ are called *differentiating actions*.

The back-translation (often called "algorithm"in the literature [14, 62, 64, 98, 99, 101, 102]) must produce a context $\mathbb{C}_{\mathsf{S}}$ that performs all interactions in the common prefix and then reacts to the differentiating actions in two different ways. More precisely, the context $\mathbb{C}_{\mathsf{S}}$ must preform all ?-decorated actions in $\overline{\lambda}$ since the !-decorated ones are done by $\mathbf{P_1}$ or $\mathbf{P_2}$.

B.3 Back-translation by Embedding

The second technique to back-translate target contexts is to embed them into source ones. The kind of embedding required depends, intuitively, on the gap in expressive power between the source and target languages of the translation. Back-translation is easiest when the source and target are syntactically identical and gets harder as the target language contains features not directly expressible in the source.

B.3.1 Precise Embedding. In the work of Ahmed and Blume [18] and Fournet, Swamy, Chen, Dagand, Strub, and Livshits [52], the target language is the same as the source. Hence, the back-translation can be done using boundaries—called "wrappers"—encoded in the same language. These wrappers are witness to a type isomorphism and consequently the translation is fully abstract. New et al. [94] characterize this form of back-translation as *precise* because the embedding of the target language is into isomorphic types.

B.3.2 Over-approximating Embedding. New et al. [94] prove full abstraction of closure conversion from the simply typed lambda calculus with recursive types to a target language with type abstraction and a modal type system to track exceptions. These languages are respectively called $\mathsf{STLC} + \mu$ and $\mathsf{SystemF} + \mathsf{E}$ in the explanatory Figure 6. Since they have recursive types in the

source, they back-translate target contexts $\mathbb{C}_\mathsf{T}$ (denoted as $\langle\langle \mathbb{C}_\mathsf{T} \rangle\rangle$) to a universal type **Univ** and show that the boundaries between types τ and **Univ** are retractions. These boundaries are two functions: **proj**:**Univ** $\to \tau$, which projects from the universal type into normal types, and **embed**:$\tau \to$ **Univ** which embeds from a normal type into the universal one, such that the following holds:

$$\textbf{proj embed}\langle\langle \mathbb{C}_\mathsf{T} \rangle\rangle \simeq_{\text{ctx}} \langle\langle \mathbb{C}_\mathsf{T} \rangle\rangle$$

Their back-translation is over-approximating in that it embeds the target language into the source at types that include many more behaviours (the grey area in Figure 6), but due to the boundaries the code is only run on "good" values, i.e., those that represent source values.

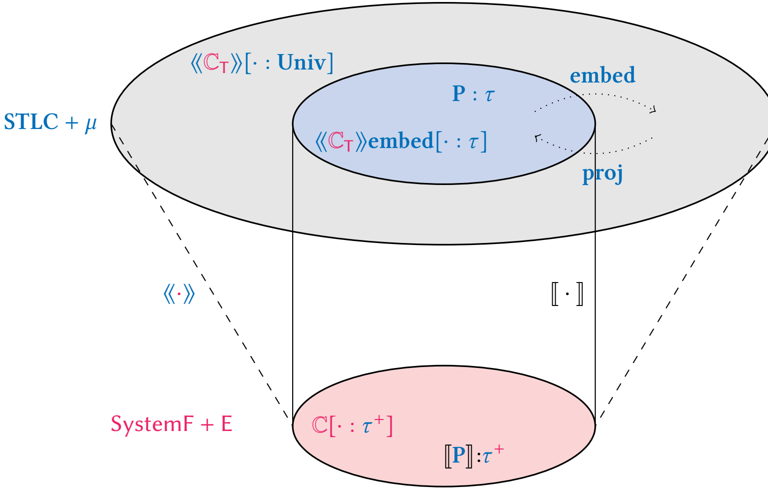


Fig. 6. Diagrammatic representation of the compiler ($[\![\cdot]\!]$) and of the precise backtranslation ($\langle\langle\cdot\rangle\rangle$) of New et al. [94].

B.3.3 Under-approximating Embedding. Devriese et al. [37] present a translation from the simply typed lambda calculus with recursive functions to the untyped lambda calculus. To prove that the translation is fully abstract, they must back-translate the untyped language to a simply typed language *without* recursive types, so they cannot construct a universal type like New et al. [94]. However, they can construct arbitrarily large approximations to the universal type, and a family of increasingly precise approximations to it. They show that since for any particular program and context observing termination only takes a finite number of steps, they can find a large enough approximation (based on the number of steps) to show that equivalence is preserved. We say their back-translation is *under-approximating* since it embeds the target language at types which include only a subset of the behaviours of the target.

Such a technique is generally useful when the source types are less expressive than the target by resorting to additional information present in the formal tools (e.g., steps). For example, when compiling System F to an untyped lambda calculus, the back-translation type needs to account for type variables and their instantiation. This turns the universal type into a universal type operator, which is expressible in System-ω but not in System-F. To overcome these kinds of issues, under-approximating back-translation can be used.[7]

[7] Personal communication with Devriese, Patrignani and Piessens, who are devising a fully-abstract compiler from System F to lambda seal [108].

Independently from the work of Devriese et al. [37], the idea of using an approximate back-translation was also mentioned recently by Schmidt-Schauß et al. [115]. In this work, the authors present a framework for reasoning about fully abstract compilation and related notions using families of translations, i.e., an approximate back-translation. They apply the idea to show that a simply-typed lambda calculus without fix but with stuck terms can be embedded into a simply-typed lambda calculus with fix. Although not very detailed, their proof seems simpler than that of Devriese et al. [37]. Partly this is because the proof addresses a simpler problem, but the idea of approximate back-translation also seems simpler to use for a language embedding. This suggests that proofs based on under-approximate back-translation can be simplified by factoring the proof into two separate passes:

(1) embedding STLC into STLC$^\mu$ (i.e., with recursive types) and using an under-approximate embedding;
(2) compiling STLC$^\mu$ with recursive types into the untyped lambda calculus using an over-approximating embedding.

To provide further clarification of this proof technique, we provide a picture from the work of Devriese et al. [37].

approx. compiler security

$$\mathbf{t_1} \simeq_{ctx} \mathbf{t_2}$$

$$\langle\langle \mathbb{C}_T \rangle\rangle_n \left[\mathbf{t_1}\right] \Downarrow_{_} \quad \overset{(2)}{\Rightarrow} \quad \langle\langle \mathbb{C}_T \rangle\rangle_n \left[\mathbf{t_2}\right] \Downarrow_{_}$$

$$\begin{array}{l} \langle\langle \mathbb{C}_T \rangle\rangle_n \gtrsim_n \mathbb{C}_T \\ \mathbf{t_1} \gtrsim_{_} [\![\mathbf{t_1}]\!] \end{array} \quad \Uparrow (1) \qquad (3) \Downarrow \quad \begin{array}{l} \langle\langle \mathbb{C}_T \rangle\rangle_n \lesssim_{_} \mathbb{C}_T \\ \mathbf{t_2} \lesssim_{_} [\![\mathbf{t_2}]\!] \end{array}$$

$$\mathbb{C}_T\left[[\![\mathbf{t_1}]\!]\right] \Downarrow_n \quad \overset{?}{\Rightarrow} \quad \mathbb{C}_T\left[[\![\mathbf{t_2}]\!]\right] \Downarrow_{_}$$

$$[\![\mathbf{t_1}]\!] \overset{?}{\simeq}_{ctx} [\![\mathbf{t_2}]\!]$$

This picture proves the hard part of compiler full abstraction by dividing it in 3 sub steps. The approximation relation is indicated with $\lesssim$ and $\gtrsim$ to express that a term (or context) $\mathbf{t}$ terminates whenever t does ($\mathbf{t} \gtrsim t$) and vice versa ($\mathbf{t} \lesssim t$). The approximation is equipped with a subscript indicating how many steps it is known to hold for.

The proof states that given a target context $\mathbb{C}_T$, we can construct its back-translation $\langle\langle \mathbb{C}_T \rangle\rangle_n$ that approximates $\mathbb{C}_T$ for n steps. This, together with the knowledge that $\mathbf{t} \gtrsim [\![\mathbf{t}]\!]$, lets us deduce implication (1). Implication (2) follows directly from the source terms being contextually equivalent.

The second condition on the back translated context approximation $\langle\langle \mathbb{C}_T \rangle\rangle_n$ is that it is *conservative*, to deduce implication (3). Conservativeness means that the source-level context may diverge in situations where the original did not, but not vice versa, as expressed by $\langle\langle \mathbb{C}_T \rangle\rangle_n \lesssim_{_} \mathbb{C}_T$. This implies that if $\langle\langle \mathbb{C}_T \rangle\rangle_n[\mathbf{t}]$ terminates in any number of steps, then so must $\mathbb{C}_T[[\![\mathbf{t}]\!]]$.

Received ; revised ; accepted