2012 IEEE 25th Computer Security Foundations Symposium

Secure Compilation to Modern Processors

Pieter Agten, Raoul Strackx, Bart Jacobs and Frank Piessens
IBBT-DistriNet
Katholieke Universiteit Leuven
Leuven, Belgium
firstname.lastname@cs.kuleuven.be

***Abstract*—We present a secure (fully abstract) compilation scheme to compile an object-based high-level language to low-level machine code. Full abstraction is achieved by relying on a fine-grained program counter-based memory access protection scheme, which is part of our low-level target language. We discuss why standard compilers fail to provide full abstraction and introduce enhancements needed to achieve this goal. We prove that our enhanced compilation scheme provides full abstraction from our high-level source language to our low-level target language. Lastly, we show by means of a prototype implementation that our low-level language with fine-grained memory access control can be realized efficiently on modern commodity platforms.**

***Keywords*-software security, compilation, full abstraction, hypervisors**

I. Introduction

High-level programming languages such as Java, C#, ML or Haskell offer protection facilities such as abstract data types, the private field modifier, or module systems. Such abstractions have long been used in programming languages, at least since the 1970s [1], [2]. They were mainly designed to enforce software engineering principles such as information hiding and encapsulation, but they can also be used as building blocks for providing security properties of programs. For instance, declaring a field private in Java can protect the confidentiality of that field towards less trusted code running in the same Java Virtual Machine.

When such protection features are used for the purpose of security, it is important to maintain the resulting security properties when the program is compiled. The classical way to formalize this notion of secure compilation is *full abstraction* [3]. Roughly speaking, compilation from a source language to a target language is fully abstract if the contextual equivalence of source programs implies the contextual equivalence of target programs and vice versa. In other words, a source-level context can distinguish two source programs if and only if a target-level context can distinguish the two corresponding target programs.

Full abstraction (and more specifically the preservation of contextual equivalence) is a good definition for secure compilation, because contextual equivalence of programs can express important security properties, such as confidentiality and integrity properties. For instance, the fact that the value of a static field f in a Java class C is confidential can be expressed by saying that class C is contextually equivalent to a class C' that only differs from C in its value for f. Full abstraction entails the preservation of all security properties that can be expressed using contextual equivalence.

Unfortunately, it is notoriously hard to securely compile higher-level languages to lower-level languages. Even the compilation of Java to JVM bytecode, or of C# to the .NET intermediate language is known not to be fully abstract [4] – even if for these cases the source and target languages are relatively close. No state-of-the-art compiler of Java-like or ML-like languages towards machine code on classic Von Neumann computer architectures is even close to fully abstract. As a consequence, security properties the source program might have are possibly lost towards attackers that can interact with the program at machine code level. Unfortunately, this is a real and important issue, as attacks in practice often rely on injecting machine code into a process' address space [5]. Also, kernel-level malware can attack any process in the system at the machine code level.

But recently, some important progress has been made. At CSF 2010, Abadi and Plotkin [6] have shown how address space layout randomization can achieve a probabilistic variant of full abstraction when compiling towards a low-level language in which memory addresses are numbers. At CSF 2011, Jagadeesan et al. [7] have extended these results to a more expressive programming language.

The main contribution of our paper is the proposal of a secure compilation technique towards low-level machine code. Instead of relying on randomization as Abadi and Plotkin, or Jagadeesan et al., our compilation technique builds on low-level memory access control techniques. It is inspired by recently developed systems for the fine-grained protection of small pieces of applications such as Flicker [8] or TrustVisor [9]. These systems show that it is possible to efficiently implement relatively fine-grained memory access control on modern processors. In this paper, we show that such fine-grained memory access control can in turn be used to support fully abstract compilation from a simple Java-like language to machine code.

More specifically, this paper makes the following contributions:

- We formalize a simplified computer architecture that models a modern processor with fine-grained memory

DOI 10.1109/CSF.2012.12

access control.

- We show by means of a prototype implementation that such fine-grained memory access control can be implemented efficiently on modern Intel processors.
- We define a compilation from a simple Java-like language to this computer architecture, and we prove that it is fully abstract.

The remainder of this paper is structured as follows. First we give an informal overview of our high- and low-level languages and our compilation scheme in Section II. Next, we formalize these languages and prove our full abstraction theorem in Section III. We then introduce our prototype implementation in Section IV. Next, we discuss our approach in Section V. We compare related work in Section VI and we conclude in Section VII.

II. Informal overview

This section presents a precise but informal overview of our approach. We first introduce the high-level language, and illustrate by means of examples in that language how security properties can be expressed using contextual equivalence. We then describe the low-level platform with its fine-grained memory access control model. Finally, we describe our compilation scheme. We first describe a basic, straightforward compilation scheme and illustrate that it is not fully abstract by means of counterexamples. We then describe the more involved, fully abstract compilation by discussing how it handles the counterexamples.

A. High-level language

Our high-level language is a small, single-threaded, object-based language. It supports the basic constructs one would expect of a modern programming language, including branches, loops and local variables. Indirect method calls are supported through method references (also known as typed function pointers or delegates). The language does not support dynamic allocation. Objects should be thought of as compilation units that encapsulate private state. The language is safe; one can prove progress and preservation using standard methods [10].

Each high-level program consists of a number of objects, each of which consists of private fields and public methods. The supported base types are Unit, Int and the method reference type $\mathrm{M}\langle\overline{U} \to T\rangle$. The language formally uses an assembly-like syntax for method bodies, but, for readability, the code examples in this paper are written using a Java-like syntax instead. Although this alternative syntax significantly changes the appearance of the language, it should be considered only a cosmetic change, as it does not influence the safety or expressivity of the high-level language. Figure 1 illustrates this syntax by showing an example object that encapsulates a value and notifies a listener through an indirect call whenever this value changes.

```
object o {
  M<(Int, Int)->Unit> listener = null;
  Int value = 0;

  Unit setListener(M<(Int,Int)->Unit> l) {
    listener = l;
    return unit;
  }

  Int getValue() {
    return value;
  }

  Unit setValue(Int v) {
    if (listener != null && value != v) {
      listener(value, v);
    }
    value = v
    return unit;
  }
}
```

Figure 1. Example of a high-level object

Execution of a high-level program starts in the `main` method of the object named $\mathrm{o_t}$. The main method must be typed $\epsilon \to \mathrm{Int}$. Execution either ends with an integer result c or gets stuck in an infinite loop. A program ends its execution with a result c by returning c from the main method.

B. Contextual equivalence and security properties

Like in any object-based language, the internal representation of an object is hidden from outside of that object's definition. This means some objects are equivalent from an external point of view, even though they have a different implementation. That is, two objects might have a different internal representation, but no third object is able to differentiate them. We say any two such objects are *contextually equivalent* and we call a third object that tries to differentiate them a *test object*, denoted O_T.

We can use contextual equivalence to express important security properties, such as the confidentiality and integrity of private fields and the integrity of object invariants. This is illustrated by the following examples (the first two examples are taken from [6] but are adapted to our programming language).

Example 1 (Confidentiality):

```
object o {
  Int secret = 0;

  Int m() {
    secret = 0;
    return 0;
  }
}
```

```
object o {
  Int secret = 0;

  Int m() {
    secret = 1;
    return 0;
  }
}
```

These two programs differ only in the value that they store in the `secret` field. By saying these objects are contextually equivalent, we are effectively saying that no external object can read or deduce the value of the `secret` field. □

Example 2 (Integrity):

```
object o {                    object o {
 Int zero = 0;                 Int zero = 0;

 Int m(M<ϵ->Unit> cb){         Int m(M<ϵ->Unit> cb){
   zero = 0;                     zero = 0;
   Unit x = cb();                Unit x = cb();
   if (zero == 0)
     return 0;                   return 0;
   else return 1;
 }                               }
}                             }
```

The left object checks whether changes were done to the `zero` field during the callback `cb()`. By saying that these objects are equivalent, we are expressing that the external code that is called through the callback function cannot modify the `zero` field. □

Example 3 (Invariants):

```
object o {                          object o {
  Int min = 0;                        Int min = 0;
  Int max = 0;                        Int max = 0;

  [...]                               [...]

  Int m() {                           Int m() {
    if (min ≤ max) {
      return 0;                           return 0;
    } else {
      return 1;
    }
  }                                   }
}                                   }
```

By saying these objects are equivalent, we are expressing that no external object can break the invariant `min ≤ max`. This is a more general kind of integrity property on the data encapsulated by an object. □

For the high-level language, these contextual equivalences (and their corresponding security properties) clearly hold, because the only way a high-level test object can interact with another object is through method calls and returns. No high-level test object O_T can distinguish the left object from the right object for any of these three examples.

To actually execute a high-level program however (without relying on an interpreter), it needs to be compiled into a lower level assembly-language program. From the viewpoint of an attacker, this low-level language is much more powerful than the high-level language, because there is no type system to make it safe. For instance, an attacker that can inject code to interact with the compiled program at the low level can read and write arbitrary memory locations. As a consequence, none of the contextual equivalences in these three examples would continue to hold at the low level, and hence also the corresponding security properties are lost.

Our objective is to define a fully abstract compilation scheme from the high-level language to a realistic low-level assembly language. The compiler must ensure that if any two high-level objects are contextually equivalent, then so are their corresponding low-level translations. We limit ourselves to the contextual equivalence of single objects in this paper. Hence, we define a context to be an arbitrary test object O_T, which can be *linked* to a single object O. Linking[1] a context O_T with an object-under-test O yields a program $O_T[O]$. If no O_T can distinguish two implementations of O then these two implementations are contextually equivalent. The notion of contextual equivalence can be generalized to talk about multiple objects, which can all interact with each other as well as with the test object. We leave this generalization for future work; many interesting security properties can already be expressed using single objects.

The validity of our full abstraction theorem implies that any high-level security property that can be expressed using contextual equivalence also holds at the low level. The power of a low-level attacker is effectively reduced to that of a high-level attacker, because any vulnerability that can be exploited at the low level can also be exploited at the high level.

C. Low-level language

The low-level language, which will be the target language of our compiler, models a Von Neumann computer architecture that offers fine-grained, program counter-based memory access control.

The basic machine model consists of a program counter, a register file, a flags register and a memory space. The program counter indicates the address of the next instruction to execute. The register file contains 12 general purpose registers R0 to R11 and a stack pointer register SP. The stack grows down, i.e. from high to low memory addresses. The flags register contains a zero flag ZF and a sign flag SF, which are set or cleared by arithmetic instructions and are used by branching instructions. The memory space is a function mapping *addresses* to *words* and contains all code and data. Addresses, words, registers and instructions are all 32 bits wide and memory is also addressed in multiples of 32 bits. The supported machine instructions are shown in Table I. So far, the low-level platform is effectively a simple model of the Intel x86 platform.

In order to support fully abstract compilation, some protection mechanism is necessary at the low level. We propose to use a fine-grained, program counter-based memory access control scheme. The scheme is inspired by existing low-level memory protection systems [8], [9], [11].

Memory is logically divided into *protected* and *unprotected* memory. The former is further divided into a *code* and a *data* section. Within the code section, a variable number of memory addresses are designated as *entry points*. These

[1]In this paper the [] symbols are used as a simple syntactic operator for constructing a program out of two objects, whereas in related work it is used as a meta-level operator mapping a context and an expression to another expression.

`movl` r_d r_s	Load the word from the memory address in register r_s into register r_d.
`movs` r_d r_s	Store the word from register r_s at the address in register r_d.
`movi` r_d i	Load the constant value i into register r_d.
`add` r_d r_s	Write $(r_d + r_s) \bmod 2^{32}$ in register r_d and set the ZF flag according to the result.
`sub` r_d r_s	Write $(r_d - r_s) \bmod 2^{32}$ in register r_d and set the ZF flag according to the result.
`cmp` r_1 r_2	Calculate $r_1 - r_2$ and set the ZF and SF flags according to the result.
`jmp` r_i	Jump to the address located in register r_i.
`je` r_i	If the ZF flag is set, jump to the address located in register r_i.
`jl` r_i	If the SF flag is set, jump to the address located in register r_i.
`call` r_i	Push the value of the program counter onto the top of the stack and jump to the address contained in register r_i.
`ret`	Pop a value from the top of the stack and jump to the popped location.
`halt`	Stop execution with the result in register R0.

Table I
LOW-LEVEL INSTRUCTIONS

from \ to	*Protected*			*Unprotected*
	Entry point	*Code*	*Data*	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x

Table II
READ-WRITE-EXECUTE MEMORY PERMISSIONS ENFORCED BY THE LOW-LEVEL LANGUAGE

addresses are the only points through which execution of code in protected memory can start. Table II shows the memory access control rules enforced by the low-level language. The size and location of each of the memory sections and the location of the entry points are specified by a *memory descriptor*, which can be considered a configuration structure for the low-level language. In Section IV we show how this memory access control scheme can be implemented efficiently on modern commodity hardware.

Execution of a low-level program starts at the first address of unprotected memory. It either ends with an integer result or gets stuck in an infinite loop. To end with a result c, the `halt` instruction must be executed with register R0 containing the value c. If an invalid memory access attempt is made, the value 0 is placed in register R0 and execution is halted.

D. Compilation

We now get to our main result, the fully abstract compilation scheme. We describe the compilation of a single high-level object O. This is sufficient to study full abstraction for our definition of contextual equivalence. High-level contextual equivalence was already defined above: two high-level objects are equivalent if no test object can distinguish them. At the low level, two compiled objects are contextually equivalent if no arbitrary machine code placed in the unprotected area of memory can distinguish the compiled objects.

We introduce our compilation scheme in two steps. First we describe a basic, straightforward compilation that places the code and data of the compiled object in the protected memory area and configures the entry points such that control flow can only enter at the start of each method. This scheme is sound, in the sense that two nonequivalent high-level objects will be compiled into two nonequivalent low-level modules. It also provides some basic protection; for instance, the low-level context cannot just scan memory to find the values of object fields as this is prevented by the low-level memory access control scheme.

However the basic scheme fails to be fully abstract. We show this by means of counterexamples. These counterexamples then motivate the final definition of our compilation scheme, for which we will prove full abstraction in Section III.

1) Basic compilation: The compilation of a high-level object O results in a low-level module $O{\downarrow}$, consisting of a partial memory space and a memory descriptor. We should prevent the low-level context from being able to distinguish two modules just by their size. Hence, a constant amount of memory is reserved for each translated object, independent of the actual memory space required. The translated object will be placed in protected memory and the memory descriptor divides the reserved space equally over the code and the data section. The compiler assumes the stack pointer register is set up by the context and is pointing to free space in unprotected memory.

The compilation process consists of translating each field and each method of the input object. To prevent a low-level module from being distinguished by the order of its methods in memory, all methods are first sorted alphabetically. Fields and methods are then given a unique index number starting at 0, based on their order of occurrence. Parameters and local variables are given a method-local index number. For a field f_i, one word of memory is reserved at the ith memory address of the data section. Integer-typed constants are translated to their corresponding numeric value. Unit-typed constants are translated to 0 and the null method reference is translated to the highest address `0xFFFFFFFF`.

To translate a method body, the compiler processes each high-level statement in turn, translating it into a list of instructions that performs the corresponding operation. Registers R0 to R3 are used as general working registers and return values are passed through R0 as well. The first eight parameters are passed through registers R4 to R11 and additional parameters are *spilled* onto the run-time stack. A prologue is prepended to each translated method body and an epilogue is appended to it. The prologue allocates and initializes a new activation record on the stack, which

contains the method's local variables and parameters. The epilogue deallocates this activation record when the method is done. This code is placed in free space in the code section.

In addition to translating each method's body, an entry point is generated for each method as well. The entry point for method m_i is placed at address $i*128$ of the code section. The offset of 128 memory locations is chosen arbitrarily, with the only condition that there is enough space between entry points to perform a number of simple operations, as will be described in section II-D3. The code at each entry point consists of two parts: (1) a call to the method's body and (2) a return instruction. When the call to the body returns, the return instruction will simply return control to the location from which the entry point was called.

Because the low-level language allows protected memory to be entered only through one of the entry points, an additional *return entry point* is generated to support returning from a callback (i.e. a call back to the context). To perform a callback, first the actual return address is placed on the stack, followed by the address of the return entry point. Control is then transferred to the context by a `jmp` instruction. When the context returns from the callback, control will first be transferred to the return entry point, which will then subsequently return back to the actual return address.

The compilation scheme as described above ensures that a module is exited either through a callback, or through the return statement at the end of an entry point. Therefore, we name the second part of each entry point as an *exit point*.

2) Limitations of the basic compilation scheme: The compilation scheme defined so far is not fully abstract, as illustrated by the examples below:

Example 4 (Stack security):

```
object o {                    object o {
 Int secret = 0;               Int secret = 1;

 Int m(M<ϵ->Unit> cb)          Int m(M<ϵ->Unit> cb)
 {                             {
   Int x = secret;               Int x = secret;
   Unit y = cb();                Unit y = cb();
   return 0;                     return 0;
 }                             }
}                             }
```

These high-level objects are equivalent but their low-level translations are not. Because local variables are placed on the runtime stack (in unprotected memory) and a low-level attacker can read unprotected memory, he can read the value of `x` during the callback `cb()`. This variable `x` contains the value of `secret`, which is different for both objects.

The current compilation scheme does not entail the confidentiality or integrity of the run-time stack, which allows attackers to read and write local variables. An attacker can use this vulnerability to read secrets from the stack, similar to a buffer-overread attack [12], or he can even tamper with control flow by overwriting a return address, similar to a classic return address clobbering attack [5]. □

Example 5 (Illegal addresses):

```
1  object o {                 object o {
2   Int f = 1;                 Int f = 1;
3
4   Int m(M<ϵ->Unit> cb)       Int m(M<ϵ->Unit> cb)
5   {                          {
6     Unit x = cb();             Unit x = cb();
7     f += 1;                    f -= 1;
8     f -= 1;                    f += 1;
9     return f;                  return f;
10  }                          }
11 }                         }
```

These high-level objects are equivalent, as in both objects the method `m` always returns 1. A low-level attacker can differentiate their translations however, by giving the address of the instructions corresponding to line 8 as the callback `cb`. In this case, the left object will decrement `f` without first incrementing it, while the right object will increment `f` without first decrementing it. This will result in `f` having a value of 0 in the left object and 2 in the right. This attack is similar to a return-oriented programming attack [13]. □

Example 6 (Information leakage):

```
object o {                    object o {
  Int m() {                     Int m() {
    Int x = 0;                    Int x = 1;
    if (x == 0) {                 if (x == 0) {
      return 0;                     return 0;
    } else {                      } else {
      return 0;                     return 0;
    }                             }
  }                             }
}                             }
```

These high-level objects are equivalent, as in both objects the method `m` always returns 0. A low-level attacker can differentiate their translations however, due to the equality test in the condition of the if-statement. This test sets the ZF flag in the first object and clears it in the second. This example illustrates that the flags register can leak information. Information can also be leaked through the general purpose registers R0 to R11 or through the stack pointer register SP. □

Example 7 (Value of unit):

```
object o {                    object o {
  Unit m(Unit x) {              Unit m(Unit x) {
    return unit;                  return x;
  }                             }
}                             }
```

These high-level objects are equivalent, because the only possible value of type `Unit` is `unit`. Their low-level translations however are not, because a low-level attacker can use any 32-bit value for parameter `x`.

As there is no purpose for having `Unit`-typed method parameters, this might seem to be an artificial problem. However, this problem is similar to a full abstraction failure for the .NET C# compiler reported by Kennedy [4], where the boolean type is two valued in C# but is byte valued in the .NET virtual machine. □

3) Secure compilation: To counter the potential vulnerabilities described above, we make a number of enhancements to the compilation scheme. We show in Section III that these enhancements make the compilation scheme fully abstract.

Stack security: The compiler must ensure the confidentiality and integrity of variables and control structures on the run-time stack. Instead of storing the entire stack in unprotected memory, it is split into an unprotected stack in unprotected memory and a secure stack in the data section of protected memory. The protected module places its activation records exclusively on the secure stack.

At the start of each entry point, the stack is switched to the secure stack and the spilled parameters for the method call (if any) are moved from the unprotected to the secure stack. At each exit point, the stack is restored to its previous address in unprotected memory. To implement these stack switches, the compiler introduces a *shadow stack pointer* field in the data section. It is initialized to the address of the middle of the data section, where the base of the secure stack will be located. At each entry and exit point, code is added to swap the value of the stack pointer register with the value of the shadow field.

A call from the protected module to unprotected memory is performed by first pushing the actual return address onto the secure stack. Next, the stack is switched to the unprotected stack and the address of the return entry point and any spilled parameters are pushed onto it. Control is then transferred to the context by a jump. When the callback returns, control will first be transferred to the return entry point, which switches back to the secure stack and subsequently transfers control back to the actual return address. Because data are written to the unprotected stack during this process, the compiler must ensure that the location of the unprotected stack is valid before it writes to it. That is, the address of the unprotected stack must lie out of protected memory, for otherwise parts of protected memory might get overwritten. Therefore, at each entry point, before swapping the value of the stack pointer register with the value of the shadow field, a run-time check is added to verify that the SP register is pointing to unprotected memory. If this check fails, the value 0 is placed into R0 and the halt instruction is executed.

To prevent the context from tampering with control flow by jumping to the return entry point when there is no callback to return from, the compiler initializes the first location of the secure stack to the address of a procedure that writes 0 to the R0 register and then halts execution. The return entry point will jump to this address if it is called when there is no callback to return from.

Because the first half of the data section is now reserved for the secure stack, the memory space for a field f_i will now be located at the ith address of the second half of the data section. Figure 2 illustrates the memory layout used by the secure compilation scheme. Note that this memory layout ensures that an overflow of the secure stack will result in a memory access violation, as the code section is non-writable.

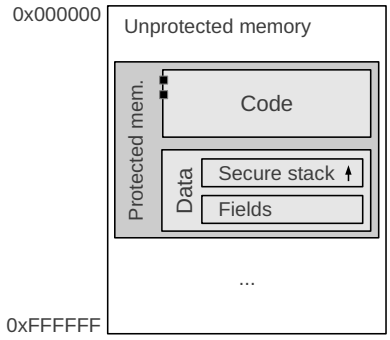

Figure 2. Memory layout of the secure compilation scheme

Illegal addresses: The compiler must ensure the integrity of control flow when jumping from a protected module to an externally supplied address. Such a jump occurs at each indirect call and at each exit point. For an indirect call, a valid destination address is (1) an address outside of the module's memory bounds, or (2) the address of one of the module's own methods with a correct signature. For an exit point, only addresses outside of the module's memory bounds are valid. A call or return to the address `0xFFFFFFFF` is also not allowed, because it corresponds to the null method reference.

The compiler adds run-time checks for these conditions at each indirect call and exit point. A check for the first type of addresses is straightforward to implement, while a check for the second type of addresses is more complicated, because it requires run-time type information. A simpler alternative solution would be to forbid indirect calls from an object to the same object in the high-level language. The compiler could then add a simple run-time check that any indirect call must go outside the protected module's memory bounds. This would not be a significant restriction, as any indirect call to a local method m can be replaced by an indirect call to a wrapper method of the context that calls m.

Information leakage: In the high-level language, the only way for two objects to communicate, is through method calls and returns. The compiler must ensure that a low-level attacker cannot use any other communication channels, as this might leak information that should be kept private to the module under protection.

The low-level computer model inherently provides three ways to exchange information: (1) through unprotected memory, (2) through the flags register and (3) through the general purpose registers R0 to R11 and SP. The first method is already precluded, because only spilled parameters are written to the unprotected stack and these values are also available at the high level. The SP register does not convey

private information, because it is restored to the location of the unprotected stack whenever control leaves the protected module. The compiler constrains the other communication methods as follows:

- The flags are cleared at each callback and exit point.
- Every general purpose register except R0 is cleared at each exit point.
- Every general purpose register is cleared at each callback, except if it is used for passing a parameter.

The compiler generates code at each callback and exit point to enforce these constraints.

Value of unit: The compiler must ensure that all memory locations corresponding to high-level fields and variables contain only values for which there is a corresponding high-level value. The only value of type `Unit` is `unit` and the corresponding low-level value was chosen to be 0. The compiler enforces this constraint by adding a run-time check at each entry point, to verify that the value of any `Unit`-typed parameter is 0. The same check is added at each callback to a method with return type `Unit`. If the check fails, the value 0 is placed into register R0 and the `halt` instruction is executed.

III. Formalization

In this section, we formalize the concepts defined in the previous section and prove that our compilation scheme is fully abstract. We first introduce a number of basic definitions. We then make a number of simplifications to the high-level language, which allow us to focus on the essence of our formalization. Next we introduce *traces* and finally we discuss our full abstraction proof.

A. Definitions

Any high-level program can be written as $O_T[O]$, where O_T is an object representing the *test context* and O is the *test subject* (i.e. the object to protect). The context is modeled as a single object, because from the viewpoint of the subject, the entire context is considered as a (potentially malicious) black box. The internal structure of this black box is irrelevant. Similarly, we write a low-level program as $M_T[M]$, with M_T the test context and M the test subject (i.e. the module under protection).

The execution of a high-level program $O_T[O]$ is written as $O_T[O] \rightarrow^* c$ if it ends with result c and as $O_T[O] \rightarrow^* \lightning$ if it gets stuck in an infinite loop. The same notation is used for low-level programs. We can now formally define what it means for two objects or modules to be contextually equivalent:

Definition 1 (High-level equivalence): For any two high-level objects O_1 and O_2, we define $O_1 \simeq O_2$ as:

$$\forall O_T : O_T[O_1] \rightarrow^* c \iff O_T[O_2] \rightarrow^* c$$

Definition 2 (Low-level equivalence): For any two low-level modules M_1 and M_2, we define $M_1 \simeq M_2$ as:

$$\forall M_T : M_T[M_1] \rightarrow^* c \iff M_T[M_2] \rightarrow^* c$$

Note that these definitions of high- and low-level contextual equivalence are based purely on the *results* of executions. We do not consider side channel attacks that could trivially break full abstraction in practice, such as timing attacks or attacks on platform-specific features such as caches or I/O channels.

B. Simplifications

For our formalization, we make three simplifications to the languages defined in the previous section. These simplifications impose no fundamental limitations but allow us to focus on the essence of our formalization.

1) The number of parameters: We limit the number of parameters of each method to eight. This prevents any parameter from being spilled onto the run-time stack and allows the compiler to pass all parameters through registers. If more than eight parameters need to be passed, their values can be stored in fields and a reference to a method that returns the value of each of these fields can be passed instead.

2) Indirect method calls: We disallow indirect method calls from the test subject to one of its own methods; only references to methods of the test context can be created. As explained in Section II-D3, this is not a fundamental limitation but it avoids the compiler from having to add run-time type information to check whether an indirect call is valid.

3) The exit statement: We allow the test context O_T to use an `exit(`c`)` statement, to end execution abruptly with a result c. Without this statement, the context might not be able to signal a detected difference between two subjects to the environment before the execution ends up in an infinite loop. This is illustrated by the following example.

Example 8 (Use for the exit statement):

```
object o {                    object o {
 Int m(M<Int->Unit> cb)        Int m(M<Int->Unit> cb)
 {                             {
   Unit x = cb(1);               Unit x = cb(2);
   while(1) { };                 while(1) { };
   return 0;                     return 0;
 }                             }
}                             }
```

These objects are not equivalent because the left object calls the callback `cb` with argument 1, while the right object calls it with argument 2. However, the context would not be able to signal this detected difference to the environment without the `exit` statement, because both objects end up in an infinite loop after returning from the callback. □

C. Languages and compiler

Our high- and low-level languages are formalized using standard semantic techniques such as BNF grammars for syntax definitions and inference rules for operational semantics and typing. Our compiler has been formalized in the form of an OCaml implementation. Due to space constraints, we do not give these formalizations here, but they can be found in our extended technical report [14].

D. Traces

Before proving our full abstraction theorem, we first introduce high- and low-level traces, which let us reason about executions at a higher level of abstraction. A formal definition of high- and low-level traces can be found in our technical report [14]. Proving full abstraction using execution traces was inspired by Jeffrey and Rathke's fully abstract trace semantics for Java Jr [15].

1) High-level traces: The trace of a high-level program $O_T[O]$ describes the interactions made between O_T and O during its execution. That is, a trace $\overline{a}$ consists of the sequence of basic actions that occur during the execution of a program. Each high-level basic action a_h is either:

- a call '$\mathrm{call}\ m(\overline{v})?$' from O_T to O, where m is the method that was called and $\overline{v}$ are the values passed as parameters
- a return '$\mathrm{ret}\ v!$' from O to O_T, where v is the return value
- a callback '$\mathrm{call}\ m(\overline{v})!$' from O to O_T, where m is the method that was called and $\overline{v}$ are the values passed as parameters
- a returnback '$\mathrm{ret}\ v?$' from O_T to O, where v is the return value

By design of the high-level language, the only way two objects can communicate is through method calls and returns. Therefore, high-level traces capture all communication between a test context and a test subject in a high-level execution.

2) Low-level traces: The low-level trace of a program $M_T[M]$ describes the interactions made between M_T and M during its execution. Each low-level basic action a_l is either:

- a call '$\mathrm{call}\ p(\overline{v})?$' from M_T to M, where p is the address to which the call was made and $\overline{v}$ are the values passed as parameters
- a return '$\mathrm{ret}\ v!$' from M to M_T, where v is the return value
- a callback '$\mathrm{call}\ p(\overline{v})!$' from M to M_T, where p is the address to which the callback was made and $\overline{v}$ are the values passed as parameters
- a returnback '$\mathrm{ret}\ v?$' from M_T to M, where v is the return value

Due to the information leakage countermeasures described in Section II-D3, low-level traces capture all communication between a test context and a test subject in a low-level execution.

E. Full abstraction

1) Overview: We can now define what it means for our compilation scheme to be fully abstract. Full abstraction breaks down into two parts: soundness and completeness.

Theorem 1 (Soundness): For any two high-level objects O_1 and O_2, we have:

$$O_1{\downarrow} \simeq O_2{\downarrow} \Rightarrow O_1 \simeq O_2$$

The soundness theorem can be considered as stating that the compiler is correct. Proving this theorem is nontrivial, but is not the main focus of this paper. Hence, we do not prove the soundness theorem here.

Theorem 2 (Completeness): For any two high-level objects O_1 and O_2, we have:

$$O_1 \simeq O_2 \Rightarrow O_1{\downarrow} \simeq O_2{\downarrow}$$

To prove the completeness theorem, we will prove the equivalent statement $O_1{\downarrow} \not\simeq O_2{\downarrow} \Rightarrow O_1 \not\simeq O_2$. Suppose $O_1{\downarrow} \not\simeq O_2{\downarrow}$, then (wlog) there exists an M_T such that $M_T[O_1{\downarrow}] \rightarrow^* c$ and $M_T[O_2{\downarrow}] \not\rightarrow^* c$. For our full abstraction proof, we will assume that M_T does this without overflowing the secure stack. Let $\overline{a_1}$ and $\overline{a_2}$ be the traces of $M_T[O_1{\downarrow}]$ and $M_T[O_2{\downarrow}]$ respectively. As the only way for M_T to differentiate $O_1{\downarrow}$ and $O_2{\downarrow}$ is by communicating with them through the basic actions described in the previous section, we know $\overline{a_1} \neq \overline{a_2}$. We will describe an algorithm that, when given O_1, O_2, $\overline{a_1}$ and $\overline{a_2}$ as input, will construct an O_T such that $O_T[O_1] \rightarrow^* c$ and $O_T[O_2] \not\rightarrow^* c$. The existence of this algorithm proves the completeness theorem. The algorithm relies on the following two propositions:

Proposition 1: If we number the actions of a high- or low-level trace starting at 0, then each even-numbered action is a call or a returnback and each odd-numbered action is a return or a callback.

Proof: Control initially is in the test context and each time an action is performed, control is switched from the test context to the test subject or vice versa. When the context is in control, it can only perform a call or a returnback and when the subject is in control, it can only perform a return or a callback. ■

Proposition 2: Any two unequal traces $\overline{a_1}$ and $\overline{a_2}$, generated by $M_T[O_1{\downarrow}]$ and $M_T[O_2{\downarrow}]$ respectively, differ for the first time at an odd-numbered action.

Proof: By Proposition 1, all even-numbered actions are calls or returnbacks, which originate from M_T. Suppose the first differing action originates from M_T, then right before that differing action, the program counter or the value of some memory location of M_T would have had to be different in the two executions, which can only be caused by a prior differing action. ■

2) Interpreting low-level values as high-level values: Before defining the algorithm, we first define a function $v{\uparrow_T}$ mapping low-level values to a corresponding high-level construct. We assume this function is part of the main algorithm, so it has access to the O_T under construction. When given a low-level value v and a type T, the function $v{\uparrow_T}$ returns a corresponding high-level value of the given type and possibly extends O_T with one additional method. The function uses a *method table*, which maps (address, method type) pairs to methods of O_T. This table is *static*, i.e., it is kept intact across different calls to this function. The result depends on the given type T:

- if $T = \text{Unit}$, then $v{\uparrow_T} = \text{unit}$
- if $T = \text{Int}$, then $v{\uparrow_T} = v$
- if $T = \text{M}\langle\overline{U} \to T'\rangle$, then $v{\uparrow_T}$ depends on whether the method table contains an entry $(v, \overline{U} \to T') \to o_t.m_i$
  - if so, then $v{\uparrow_T} = o_t.m_i$
  - if not, then O_T is extended with a new method m_j, with type $\overline{U} \to T'$ and $v{\uparrow_T} = o_t.m_j$. The method table is extended with the entry $(v, \overline{U} \to T') \to o_t.m_j$.

If O_T is extended with a new method of type $\overline{U} \to T'$, that method must contain a return statement to be syntactically correct:

- if $T' = \text{Unit}$, then the return value is unit
- if $T' = \text{Int}$, then the return value is 0
- if $T' = \text{M}\langle\overline{U'} \to T''\rangle$, then the return value is null

3) Algorithm: The algorithm for constructing O_T uses an integer *step counter* variable i, a stack of *return locations* $\overline{r}$ and a *current method* m_c. The step counter i determines the two current basic actions $a_1^{(i)}$ and $a_2^{(i)}$. The algorithm uses an interpreter for the high-level language as a subcomponent.

Initialization: O_T is initialized to:

```
object o_t {
  Int step = 0;

  Int main() {
    return 0;
  }
}
```

The return location stack $\overline{r}$ is empty, the current method m_c is set to `main` and the step counter i is set to 0. A scan is made of O_1, O_2 and their low-level translations, to create a list of all method reference constants and their corresponding type. For each method reference $o_t.m_k$ with type $\overline{U} \to T$ and corresponding low-level value p, a method m_k is created in O_T and an entry $(p, \overline{U} \to T) \to o_t.m_k$ is added to the method table of the $v{\uparrow_T}$ function defined above.

After initialization, the algorithm alternates between a *construction* mode and an *execution* mode and increments i by 1 at each alternation. It maintains the following invariant: $\forall j < i : a_1^{(j)} = a_2^{(j)}$.

Construction mode: Whenever the algorithm is in this mode, i is even, so by Proposition 2 we have $a_1^{(i)} = a_2^{(i)} \equiv a^{(i)}$. The algorithm will add a block of code right before the return statement of the current method m_c. The code to add depends on the type of $a^{(i)}$:

- *for a call*: $a^{(i)} = \text{call } p(\overline{v})?$, the code to add is:

  ```
  if (step == <i>) {
    step += 1;
    T_0 x_0 = <v_0↑T_0>;
    ...
    T_j x_j = <v_j↑T_j>;
    T ret = <o.m_k>(x_0,...,x_j);
  ⇒
  }
  ```

  Where `<i>` is replaced with the current value of the step counter[2] i, the $\langle v_i{\uparrow_{T_i}}\rangle$'s are replaced with the result of the function described above and the arrow indicates the return location l_r. The method $\texttt{o.m}_k$ is determined by the address p: it must correspond to one of the method entry points of $O_1{\downarrow}$ and $O_2{\downarrow}$, for otherwise the two low-level executions that generated $\overline{a_1}$ and $\overline{a_2}$ would both have halted with result 0 after performing this call, which would implicate $\overline{a_1} = \overline{a_2}$, contradicting the precondition that they are differentiating traces. The method index k is determined as: $k = \frac{p-b}{128}$, with b the base address of the protected code section. By inspecting the method signature of $o.m_k$ in O_1 and O_2, the parameter types of $o.m_k$ can be determined, allowing the algorithm to determine the types T_i for the calls to the $v{\uparrow_T}$ function.

  After generating this block of code, the algorithm increments its internal step counter i, pushes l_r onto the return stack $\overline{r}$ and switches to the *execution* mode, passing the location of $o.m_k$ in O_1 and the location of $o.m_k$ in O_2 as arguments.
- *for a returnback*: $a^{(i)} = \text{ret } v?$, the code to add is:

  ```
  if (step == <i>) {
    step += 1;
    T x = <v↑T>;
    return x;
  }
  ```

  Where `<i>` and $\langle v{\uparrow_T}\rangle$ are replaced as described above. The type T to use in the call to $v{\uparrow_T}$ is the return type of the current method m_c.

  After generating this block of code, the algorithm increments its internal step counter i, pops two locations l_1 and l_2 from the the return stack $\overline{r}$ and switches to the *execution* mode, passing l_1 and l_2 as arguments.

Execution mode: Whenever the algorithm is in this mode, the step counter i is odd. This mode takes two arguments l_1 and l_2, pointing to code locations in O_1 and O_2 respectively. The algorithm first checks if the current low-level actions $a_1^{(i)}$ and $a_2^{(i)}$ exist. At least one of them must exist, for

[2]The `step` field has a limited 32-bit range, but we can simulate a field with an arbitrarily large range using multiple fields.

otherwise $\overline{a_1}$ = $\overline{a_2}$. If $a_1^{(i)}$ exists, the algorithm runs its interpreter from location l_1 until it encounters a callback or a return to O_T. If $a_2^{(i)}$ exists, it does the same for location l_2. If only one exists, we assume (wlog) it is $a_1^{(i)}$. The algorithm continues based on the type of the encountered action:

If only $a_1^{(i)}$ exists:

- *for a return*: ret v!
  The algorithm first determines l_r, which is the code location in O_T this return returns to, by popping it from the top of the return stack. It then adds the statement '`exit(1)`' at this location and ends.
- *for a callback*: call $o_t.m_k(\overline{v})$!
  The algorithm adds the following code right before the final return statement in method m_k of O_T:
  ```
  if (step == <i>) {
    exit(1);
  }
  ```
  Where `<i>` is replaced with the current value of the step counter i. The algorithm ends after adding this code.

If both $a_1^{(i)}$ and $a_2^{(i)}$ exist:

- *for two returns*: ret v_1! and ret v_2!
  The algorithm first determines l_r, which is the code location in O_T the return statements return to, by popping it from the top of the return stack. The current method m_c is set to the method containing this location. The algorithm then checks whether the values of v_1 and v_2 are equal. If so, the algorithm adds the statement '`step += 1`' at location l_r, it then increments its internal step counter i and returns to the construction mode.
  If the values are different, the algorithm adds the following code at location l_r:
  ```
  if (ret == <v1>) {
    exit(1);
  } else {
    exit(2);
  }
  ```
  Where `<v1>` is replaced by v_1. The algorithm ends after adding this block of code.
- *for two callbacks*: $o_t.m_{k1}(\overline{v_1})$ and $o_t.m_{k2}(\overline{v_2})$
  If $m_{k1} = m_{k2} \equiv m_k$ and $\overline{v_1} = \overline{v_2}$, the algorithm adds the following code right before the final return statement in m_k:
  ```
  if (step == <i>) {
    step += 1;
  }
  ```
  Where `<i>` is replaced with the current value of the step counter i. It then pushes the locations in O_1 and O_2 right after the callbacks onto the return stack, sets m_k as the new current method and increments its internal step counter i, before returning to the construction mode.
  If $m_{k1} = m_{k2} \equiv m_k$ but $\overline{v_1} \neq \overline{v_2}$, the algorithm adds the following code right before the final return statement in m_k:
  ```
  if (step == <i>) {
    if (<xj> == <v1>) {
      exit(1);
    } else {
      exit(2);
    }
  }
  ```
  Where `<i>` is replaced with the current value of the step counter i, `<xj>` is replaced with the name of the formal parameter with the differing value and `<v1>` is the differing value given by O_1. The algorithm ends at this point.
  If $m_{k1} \neq m_{k2}$, the algorithm adds the following code to m_{k1} and m_{k2}:
  ```
  if (step == <i>) {
    exit(<num>);
  }
  ```
  Where `<i>` is replaced with the current value of the step counter i and `<num>` is 1 in m_{k1} and 2 in m_{k2}. The algorithm ends at this point.
- *for two different actions*: ret v! and call $o_t.m_k(\overline{v})$!
  The algorithm pops the return location l_r from the return stack and adds the statement '`exit(1)`' at that location. It then adds the following code right before the final return statement in $o_t.m_k$:
  ```
  if (step == <i>) {
    exit(2);
  }
  ```
  Where `<i>` is replaced with the current value of the step counter i. The algorithm ends at this point.

4) Proof: We will now prove that the O_T constructed by the algorithm above can indeed differentiate between O_1 and O_2. Due to space constraints, we only give a sketch of the proof in this section, but we give the full proof in our technical report [14].

The algorithm relies on maintaining an equivalence between the states of O_1 and $O_1{\downarrow}$ in the executions $O_T[O_1]$ and $M_T[O_1{\downarrow}]$, and similarly between the states of O_2 and $O_2{\downarrow}$. We first define what we mean by this equivalence.

Definition 3 (Equivalent states): We say that the state of O at a certain point in the execution of $O_T[O]$ is equivalent to the state of $O{\downarrow}$ at a certain point in the execution of $M_T[O{\downarrow}]$, when:

1) If the next statement to be executed at the high-level is part of O, then the low-level program counter points to the first instruction corresponding to that statement in $O{\downarrow}$.
2) For any Int-typed field, local variable or parameter in O with value v, the corresponding memory location in $O{\downarrow}$ contains the same value v.
3) For any Unit-typed field, local variable or parameter in O, the corresponding memory location in $O{\downarrow}$ contains the value 0.
4) For any two fields, local variables or parameters in O with the same method reference type $\mathrm{M}\langle\overline{U} \to T\rangle$ with

values $o_t.m_{k1}$ and $o_t.m_{k2}$ respectively, the values of the two corresponding memory locations in $O\downarrow$ are equal iff $o_t.m_{k1} = o_t.m_{k2}$.

We can now sketch our proof of why the algorithm described above is correct.

Theorem 3 (Algorithm correctness): When given two high-level objects O_1 and O_2 and two low-level traces $\overline{a_1}$ and $\overline{a_2}$ belonging to the executions of $M_T[O_1\downarrow]$ and $M_T[O_2\downarrow]$ respectively, where M_T is a low-level module that differentiates between $O_1\downarrow$ and $O_2\downarrow$ without overflowing the secure stack, the algorithm described above constructs an O_T that differentiates between O_1 and O_2.

Proof sketch: Because the compiler translates each initial high-level field value to a corresponding low-level value, the state of O_1 and $O_1\downarrow$ are equivalent at the start of the executions of $O_T[O_1]$ and $M_T[O_1\downarrow]$. The same applies to the states of O_2 and $O_2\downarrow$. Because O_T mimics all the basic actions performed by M_T, we argue that O_T maintains this correspondence of high- and low-level subject states throughout the executions. Because of this state correspondence, O_1 and O_2 will perform high-level basic actions similar to their low-level counterparts $O_1\downarrow$ and $O_2\downarrow$ respectively. Therefore, when the low-level actions $a_1^{(2i+1)}$ and $a_2^{(2i+1)}$ differ, so will the corresponding high-level actions $b_1^{(2i+1)}$ and $b_2^{(2i+1)}$ generated by $O_T[O_1]$ and $O_T[O_2]$ respectively. By the construction of O_T, any detected difference between O_1 and O_2 is reported to the environment immediately, using the `exit` statement. Hence, the O_T constructed using the algorithm described above differentiates O_1 from O_2. ■

Theorem 4 (Full abstraction): For any two high-level objects O_1 and O_2, we have (up to the limited size of the secure stack):

$$O_1 \simeq O_2 \Leftrightarrow O_1\downarrow \simeq O_2\downarrow$$

Proof: Follows directly from our assumption of soundness and from Theorem 3. ■

IV. Implementation

Our full abstraction result relies on the program counter-based memory access control scheme of the low-level language. This result is only useful in practice if it has an efficient real-world implementation. Efficiency is important, as the main motivation to compile a language is performance; if one does not care about performance, one can simply interpret the high-level program, which is safe assuming the interpreter and underlying operating system are safe. At least three types of implementations are possible: (1) a hardware implementation, (2) a software implementation based on the virtualization support offered by modern processors and (3) a software kernel-level implementation.

An important difference between these three types is the size of the trusted computing base (TCB). Having a small TCB is important because this gives better assurance that there are no vulnerabilities in the implementation that could be exploited to bypass the access control scheme. Recent research [11], [16] has proposed modifications to processors for embedded systems to provide a memory access control scheme that is very similar to the one provided by our low-level language. Such hardware implementations have a very small TCB, as only the hardware itself needs to be trusted. However, a hardware implementation would make the security measure unsuitable for commodity platforms available today. Conversely, a virtualization-based implementation is supported by currently available commodity hardware and, as we will show below, can also be achieved with a small TCB. A kernel-level implementation does not require virtualization support and hence can even run on older hardware. However, this solution would include the entire kernel in its TCB. Given that kernel-level malware is a realistic threat on internet-connected computers today, a kernel-level implementation will most likely not provide sufficient security. This is an additional reason (besides performance) why an interpreter would not provide a good solution.

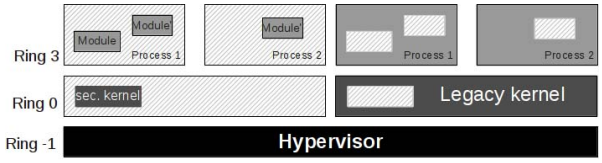

Figure 3. The architecture of our prototype is based on two virtual machines to minimize performance overhead

Given these conditions, we followed the second implementation strategy, which is inspired by implementation techniques used in other security architectures that support fine-grained isolation of pieces of application logic [8], [9]. The key idea is to build on the virtualization support offered by modern-day commodity hardware. Since memory permissions only need to change whenever a protected module is entered or exited, we can trap such entries and exits in a small hypervisor, and reconfigure the standard hardware memory access control unit (MMU) as necessary. We first describe the overall architecture of our implementation and then report on the size of its TCB and provide some performance benchmarks.

A. Architecture

Our prototype implementation is based on a small hypervisor that runs two virtual machines, called the Legacy VM and the Secure VM (see Figure 3). Both VM's have the same view of physical memory, but have different memory access control configurations.

1) Legacy VM: The Legacy VM executes all legacy applications and other code in unprotected memory. Using virtualization techniques, this virtual machine is able to execute commodity operating systems and legacy applica-

tions without any modification. From the point of view of the Legacy VM, the only difference compared to running on bare hardware is that certain memory locations are inaccessible. More specifically, two memory regions are inaccessible to the Legacy VM: (1) the memory region reserved for the hypervisor and (2) the *protected* memory region as defined in our low-level machine model. Whenever an access to these memory locations is attempted, execution traps to the hypervisor.

2) Hypervisor: The hypervisor serves two simple purposes. First, it offers a coarse-grained memory protection: it prevents *any* code executing in the Legacy VM from accessing the protected module or the security measure itself (as discussed above) and it prevents the Secure VM from accessing the hypervisor.

Second, the hypervisor implements a simple scheduling algorithm. When the Legacy VM calls an entry point in the protected module, control goes to the hypervisor who then schedules the Secure VM. Execution control only returns to the Legacy VM when the protected module either returns or performs a callback to unprotected memory.

3) Secure VM: The Secure VM can access all memory, with the exception of memory containing the hypervisor. The fine-grained memory access control mechanism is implemented by a security kernel running in this VM, as follows. First, when a request is received from the hypervisor to execute a method in the protected module, the requested entry point is checked against a list of valid entry points provided in the module's memory descriptor. When this check passes, the hardware memory management unit is set up to allow memory accesses to the module's memory region and execution proceeds from the entry point that was called. When execution tries to jump back out of the protected module, a page fault is generated, which causes the security kernel to return execution control to the Legacy VM.

B. Trusted Computing Base

An important factor for the security assurance provided by our system is the size of its TCB. Table III shows the code size of the different parts of our prototype's TCB, as measured by SLOCCount[3]. Only the hypervisor (VMM) and the secure kernel are trusted. They contain 1,045 and 1,947 lines of C and assembly code respectively. This does not include the 4,167 lines of code that is shared between both parts. This totals the size of the implementation of the TCB to only 7,159 lines of code.

C. Performance

We performed two benchmarks to quantify the performance of our memory access control implementation. First, we measured the impact on the overall system. Next, we measured the cost of transitioning between unprotected and protected memory.

[3]http://www.dwheeler.com/sloccount/

VMM	*Secure kernel*	*Shared*	*Total*
1,045	1,947	4,167	7,159

Table III
THE TCB CONSISTS OF ONLY 7K LINES OF C AND ASSEMBLY CODE.

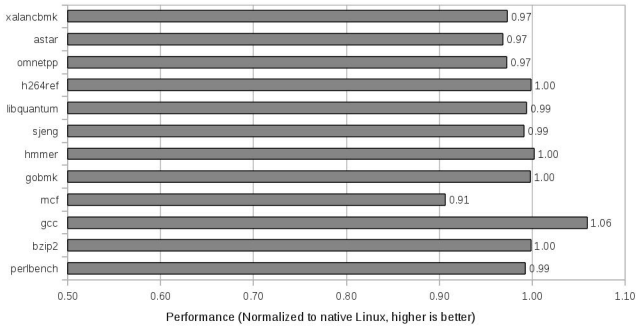

Figure 4. The spec2006 benchmarks show a very low overhead on legacy applications.

All our experiments were performed on a Dell Latitude E6510, a mid-end consumer laptop equipped with an Intel Core i5 560M processor running at 2.67 GHz and 4 GiB of RAM. Due to limitations of our prototype, we disabled all but one core in the BIOS. An unmodified version of KUbuntu 10.10 running the 2.6.35-22-generic x86_64 kernel was used as the operating system.

1) System-wide performance cost: Our implementation uses a small hypervisor, which effects the performance of both the protected module as well as all legacy code. To measure the performance impact of the hypervisor on legacy applications, we ran the SPECint 2006 benchmarks. Figure 4 displays the results. With the exception of the `mcf` application (10.36%), all applications have an overhead of less than 3.28%. We contribute the performance increase of gcc to cache effects.

As our implementation does not require any computation when the protected module is not under execution, this performance overhead can be contributed completely to the hardware virtualization support. We expect that as this support matures, performance overhead will be reduced further. Note that our hypervisor can be unloaded when it is no longer required, reducing the overhead to 0%.

2) Microbenchmarks: To measure the impact of crossing the protected/unprotected memory boundary, we implemented a *PingPong* protected module that immediately returns control back to the caller after being called. Using a hardware high-frequency timestamp, we calculated both the time to enter the module as the round trip time. Each test was executed 100,000 times. Results (see Table IV) show an overhead of 8,167% and 8,782% respectively compared to a similar module implemented as a driver. So a transition between the two protection domains in our system is about

	Module	*Driver*	*Overhead*
Entry	4.35	0.05	8,167%
Round trip	6.58	0.07	8,782%

Table IV
MODULE VS. DRIVER ACCESS OVERHEAD (IN μS).

80 times slower than calling a driver in the operating system. This overhead is mainly caused by the fact that each time the protected/unprotected memory boundary is crossed, a different virtual machine needs to be scheduled. This measurement is an upper bound: as the hardware implementations of virtualization mature, the performance cost of VM transitions will decrease further. Nevertheless, even this upper bound seems small enough to lead to a negligible overhead on the overall application when the protected module is relatively large. For instance, a sensible application design would be to put cryptographic code in the protected module. Calls to this crypto module would cost a few microseconds of performance overhead, which is negligible if the cryptographic operations are computationally intensive.

3) Conclusions: Our benchmarks indicate that our fine-grained, program counter-based memory access control scheme can be implemented efficiently on commodity hardware. At the same time, it should be clear that the main contribution of this paper is our full abstraction result, and a mature implementation with rigorous performance micro- and macro-benchmarks is future work.

V. DISCUSSION

In this section, we first discuss our choice of using full abstraction as the definition of secure compilation. We then discuss limitations of our high- and low-level languages.

A. Security by full abstraction

As illustrated in Section II-B, full abstraction can be used as the definition of secure compilation, because the preservation of contextual equivalence expresses important security properties. Full abstraction ensures that programs that are safe at the source-code level, remain safe after compilation. Though, one could argue that full abstraction is too restrictive as a definition of secure compilation. For instance, sorting methods alphabetically before compilation to hide the order of an object's methods in memory is required to provide full abstraction, yet unnecessary if we are only interested in providing confidentiality and integrity of data and in maintaining control flow integrity.

There are two ways to deal with this problem: (1) we can use a different definition for secure compilation that is not based on full abstraction or contextual equivalences or (2) we can adjust our source and/or target language to better match our desired definition of security. As an example of the latter, consider again the issue described above, where we unnecessarily hide the order of methods in memory. We could add an operation to the high-level language that, when given two methods of an object, returns whether or not the first one is defined before the second one. This would make the high-level language more powerful, without compromising its safety, and it would do away with the need to hide the order of the methods at the low level in order to provide full abstraction.

B. Language limitations

The high-level language as described in this paper has a number of limitations as compared to modern programming languages. Most noticeably, it does not have support for multiple interacting objects or dynamic memory allocation. We shortly discuss both of these extensions below. Other extensions, such as support for dynamic dispatch, are also worth discussing, but we leave them for future work.

1) Multiple objects: Extending our languages to support multiple interacting objects that are part of a single trust domain is straightforward. As these objects trust each other, they can be placed together in protected memory and can share a single secure stack and return entry point.

However, in a more realistic situation, each object is in its own trust domain, i.e., each object only trusts itself. This situation is more complicated, as each object would require its own protected memory area, including its own secure stack. The memory access protection scheme would have to take this into account. A number of changes to the compilation scheme would have to be performed as well. For instance, the stack switch on entry and exit of a module would have to be modified, because spilled parameters and return addresses of calls between two protected modules cannot be written on either of their stacks, as neither stack is accessible by both modules. Furthermore, new attack vectors might exist due to the increased complexity of multiple interacting modules. For instance, an attacker could try to make two low-level modules interact in ways that could never occur at the high level, leading to undefined behavior that is dependent on the specific implementation of a module. New compiler measures would have to be installed, to protect against these new attacks.

Nevertheless, we believe a memory access protection scheme very similar to the one presented in Section II-C is powerful enough to support a fully abstract compilation scheme from a high-level language with multiple interacting objects to an assembly-like low-level language. The details of these extensions, however, are left for future work.

2) Dynamic memory allocation: For simplicity, our high-level language does not support dynamic memory allocation. However, adding a simple form of dynamic memory allocation would not pose any fundamental problems.

To support dynamic memory allocation, a predetermined amount of protected data memory can be reserved as heap space in each module, similar to the memory reserved for

the secure stack. An object could use this space to safely store internal, private data, for instance in the form of structured records. Similar to statically allocated fields, the high-level language should prevent an attacker from having direct access to these heap-allocated records, as this would violate the encapsulation of an object.

Supporting the dynamic allocation of full-fledged protected objects is more difficult. For this strategy, the heap could be placed in (initially) unprotected memory. To create a new protected object, first a sufficiently large amount of space must be allocated on the heap and loaded with the desired code and initial data values. Next, the newly allocated space must be marked as a protected memory area. This would allow the new object to be accessed in the same way as other protected objects. Note that this requires the memory protection scheme to support multiple protected memory areas and the dynamic creation of such areas. As explained above, supporting multiple protected objects is possible but nontrivial. Furthermore, if objects are created dynamically, references to these objects will need to be passed around for other objects to access them. Supporting such references without breaking full abstraction is also nontrivial, for one because they leak information about the order in which objects are allocated and possibly about their size. Hence, this extension is also left for future work.

VI. Related work

There is a huge amount of research on secure compilation to machine code, but in most works the emphasis is on hardening the compilation of unsafe languages such as C to protect against exploitation of the compiled program by feeding it malicious input. Younan et al. [17] give an extensive survey. Some notable examples that can provide formal guarantees include control-flow integrity (CFI) [18], and obfuscation [19].

In our work, attackers can do more than just supply input; attackers can execute arbitrary code in the low-level language. The idea to formalize secure compilation to lower level languages as full abstraction (and thus protect against this more powerful type of attacker) was pioneered by Abadi [3]. In that paper, Abadi illustrates this idea in two settings: the compilation of Java to bytecode, and the implementation of secure channels in the π-calculus by means of cryptographic protocols. The second setting, proving the soundness of cryptographic implementations, has received a lot of attention but it is less related to the work reported in this paper. The first setting, secure compilation to lower level languages, was studied in the context of compilation to .NET bytecode by Kennedy [4]. But for compilation to low-level code with natural number addressing for memory, only very recently Abadi and Plotkin [6] have shown that Address Space Layout Randomization (ASLR) is a sufficiently strong software protection technique to achieve fully abstract compilation in a probabilistic sense. Jagadeesan et al. [7] extend the results of Abadi and Plotkin to a richer programming language with dynamic memory allocation, first class and higher order references and unstructured control flow. Instead of relying on randomization as the fundamental protection mechanism, our work shows that program counter-dependent low-level memory access control is also a sufficiently strong protection mechanism to achieve fully abstract compilation. Our proof technique of showing full abstraction via traces was strongly inspired by the work of Jeffrey and Rathke on Java Jr [15]. They show that traces are a fully abstract semantic model for a Java-like language similar to the one we study in this paper.

The fact that such fine-grained memory isolation could be achieved with reasonable performance was shown by a second line of related research that influenced our work. Several authors have proposed security architectures with a closely related low-level isolation mechanism. While there are significant differences between these security architectures and our own prototype (in for instance the implementation techniques used), their access control mechanisms are comparable. For example, Nizza [20], Flicker [8], TrustVisor [9] and SICE [21] also provide isolation of small pieces of application logic. The memory access control mechanisms enforced by these architectures are a special case of our model, where accesses to unprotected memory from modules is not allowed. P-MAPS [22], like our approach, does allow access to unprotected memory. All these papers are systems-papers: they report on working systems without providing formal security guarantees. It is likely that several of these proposed security architectures could be low-level target platforms for a secure compilation process as we developed in this paper. In addition, these papers provide evidence that the low-level memory access control we need in our model is efficiently implementable on today's computer platforms.

VII. Conclusion

Protection facilities in high-level programming languages can be used to enforce confidentiality and integrity properties for the data managed by one component towards other (potentially malicious) components that it interacts with. Maintaining such security properties after compilation to a low-level language requires some protection features in the low-level language as well. Randomization is one such protection feature that is known to be strong enough to support secure compilation. We have shown in this paper that program counter-based memory access control is also suitable as a low-level protection feature. We developed a model of such a low-level platform, and have proven that a Java-like high-level language can be securely compiled to this platform. We have also shown that the low-level platform is realistic, in the sense that it can be implemented on today's computers with acceptable performance. We believe our secure compilation technique can help address the pervasive threat of kernel-level malware.

ACKNOWLEDGMENTS

This research is partially funded by the Research Fund K.U.Leuven and the EU-funded FP7-project NESSoS. Pieter Agten is a Ph.D. fellow of the Fund for Scientific Research - Flanders (FWO). Raoul Strackx is a Ph.D. fellow of the agency for Innovation by Science and Technology (IWT).

REFERENCES

[1] J. H. Morris, Jr., "Protection in programming languages," *Commun. ACM*, vol. 16, no. 1, pp. 15–21, Jan. 1973. [Online]. Available: http://doi.acm.org/10.1145/361932.361937

[2] B. Liskov and S. Zilles, "Programming with abstract data types," *SIGPLAN Not.*, vol. 9, no. 4, pp. 50–59, Mar. 1974. [Online]. Available: http://doi.acm.org/10.1145/942572.807045

[3] M. Abadi, "Protection in programming-language translations," in *ICALP*, 1998, pp. 868–883.

[4] A. Kennedy, "Securing the .net programming model," *Theor. Comput. Sci.*, vol. 364, no. 3, pp. 311–317, 2006.

[5] U. Erlingsson, Y. Younan, and F. Piessens, "Low-level software security by example," in *Handbook of Information and Communication Security*. Springer, 2010, pp. 663–658. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/267049

[6] M. Abadi and G. D. Plotkin, "On protection by layout randomization," in *CSF*, 2010, pp. 337–351.

[7] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely, "Local memory via layout randomization," in *CSF*, 2011, pp. 161–174.

[8] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*. ACM, Apr. 2008, pp. 315–328. [Online]. Available: http://www.ece.cmu.edu/~jmmccune/papers/mccune_parno_perrig_reiter_isozaki_eurosys08.pdf

[9] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010. [Online]. Available: http://www.ece.cmu.edu/~jmmccune/papers/MLQZDGP2010.pdf

[10] B. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[11] R. Strackx, F. Piessens, and B. Preneel, "Efficient isolation of trusted subsystems in embedded systems," in *SecureComm*, 2010, pp. 344–361.

[12] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *EUROSEC*, 2009, pp. 1–8.

[13] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *Trans. Info. & Sys. Sec.*, 2011.

[14] P. Agten, R. Strackx, B. Jacobs, and F. Piessens, "Secure compilation to modern processors: extended version," KU Leuven, Tech. Rep. CW 619, 2012. [Online]. Available: http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW619.pdf

[15] A. Jeffrey and J. Rathke, "Java jr: Fully abstract trace semantics for a core java language," in *ESOP*, 2005, pp. 423–438.

[16] K. El Defrawy, A. Francillon, D. Perito, and G. Tsudik, "Smart: Secure and minimal architecture for (establishing a dynamic) root of trust," in *Proceedings of the Network & Distributed System Security Symposium (NDSS), San Diego, CA*, 2012. [Online]. Available: http://francillon.net/~aurel/papers/2012_SMART.pdf

[17] Y. Younan, W. Joosen, and F. Piessens, "Runtime countermeasures for code injection attacks against c and c++ programs," *ACM Computing Surveys*, no. to appear, 2012.

[18] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, pp. 4:1–4:40, November 2009. [Online]. Available: http://doi.acm.org/10.1145/1609956.1609960

[19] R. Pucella and F. B. Schneider, "Independence from obfuscation: A semantic framework for diversity," in *CSFW*, 2006, pp. 230–241.

[20] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth, "Reducing tcb complexity for security-sensitive applications: three case studies," in *EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. New York, NY, USA: ACM, 2006, pp. 161–174. [Online]. Available: http://www.cs.kuleuven.ac.be/conference/EuroSys2006/papers/p161-singaravelu.pdf

[21] A. Azab, P. Ning, and X. Zhang, "Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 375–388. [Online]. Available: http://www4.ncsu.edu/~amazab/SICE-CCS11.pdf

[22] D. P. Sahita R, Warrier U., "Protecting Critical Applications on Mobile Platforms," *Intel Technology Journal*, vol. 13, pp. 16–35, 2009. [Online]. Available: http://www.cs.unh.edu/~it666/reading_list/Hardware/intel_techjounral_security.pdf