# Towards Safe Enclaves

Neline van Ginkel Raoul Strackx Jan Tobias Mühlberg
Frank Piessens

iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Belgium

**Abstract**

Protected module architectures, like the recently launched Intel Software Guard Extensions (Intel SGX), make it possible to protect individual software modules of an application against attacks from other modules of the application, or from the operating system. But if the code of the protected module (the *enclave* in Intel SGX terminology) has vulnerabilities itself, that module can still be exploitable. Programming enclaves in a safe programming language can prevent a wide range of vulnerabilities within the enclave.

However, the simple approach of programming enclaves in a safe programming languages gives less security guarantees than one might expect. A safe language only provides safety guarantees for whole programs, not for individual modules that are part of a bigger program. If the context of the module is malicious, additional defensive measures are required to guarantee the safety of the enclave. This paper illustrates this problem, and reports on work-in-progress towards a solution.

## 1 Introduction

Software systems are often attacked by exploiting low-level details of their implementation. Attacks that exploit memory safety errors are an obvious example [EYP10]: by triggering a memory safety bug in a C or C++ program, the program writes to memory cells that it is not supposed to write to, and the effect of this memory corruption depends on low-level details of the compiler, operating system and hardware. By carefully choosing inputs sent to the program, the attacker can often make the program misbehave.

But also more recent attacks like memory scraping [Huq15] are an example: an attacker that can compromise the operating system can exfiltrate secret information (e.g. keys or credit card information) by scanning the virtual memory space of applications running on the operating system.

An important defense against memory corruption attacks are *safe programming languages* like Java, Scala or Rust [MK14]. For an attacker model where the attacker can interact with a complete program by providing input and by

reading output, such safe languages provide complete protection against memory corruption. Roughly speaking, a safe language ensures this by making sure that behavior of programs is always well-defined whatever input the attacker provides, thus making it impossible for the program to run into *undefined* behavior that might be implementation-dependent and potentially dangerous.

An important defense against memory scraping attacks are *protected module architectures* [MPP+08, SP12, NAD+13], now also supported in recent Intel processors through the new Intel Software Guard Extensions, Intel SGX [Int14, MAB+13]. For an attacker model where the attacker can control all infrastructural software, including the operating system and language runtime libraries, a protected module architecture can execute a software component in a *protected module* (called *enclave* in Intel SGX). Execution as a protected module provides strong assurance that only a module's code can access that module's memory, thus countering memory scraping attacks [CBB+01, SYP+09] and many other layer-below attacks.

These two defense mechanisms, safe languages and protected module architectures, complement each other well: protected module architectures limit what a malicious *context* can do to a software module, and language safety avoids memory safety vulnerabilities *within* a module. This paper reports on work in progress on the combination of these two mechanisms. We show how a module that is programmed in a safe language and executes in an enclave can still be subject to low-level attacks that exploit implementation details and can cause memory corruption within the module. This is due to the fact that a software module within an enclave may offer a richer API than just input and output of primitive values (like integers or strings). Methods or functions callable from the malicious context might also return (or accept as parameters) references to mutable objects, or function pointers.

The objective of our work is to extend the notion of safety to handle these additional cases: we investigate what defensive measures a compiler must take to ensure that a protected module that is written in a safe language never runs into undefined behavior. Note that this problem is related to – but different from – *fully abstract* compilation [Aba99], and safe language interoperability [MF07]. We discuss the similarities and differences in Section 5.

The remainder of this paper is structured as follows: first, in Section 2, we briefly recap the two mechanisms, language safety and protected module architectures, which we combine in this paper. Section 3 shows that the straightforward combination of these two does not give us safe enclaves: enclaves programmed in a safe language can still be unsafe in the sense that they can run into undefined, hence implementation-dependent, behavior. Section 4 informally sketches the additional checks a compiler should do to ensure safety, and outlines the status of this work in progress. Finally, in Sections 5 and 6, we discuss related work and conclude.

We emphasize that this paper reports on *work in progress*. We illustrate problems and solutions informally. A full formalization and implementation of the ideas in this paper is future work.

# 2 Background

## 2.1 Safe languages

Safe languages, like Java or C#, avoid implementation-dependent behavior and hence also any kind of attack that exploits such implementation-dependent behavior, including memory corruption attacks, by ensuring that the execution of programs is always well-defined. Examples of program constructs that could lead to undefined behavior include (1) accessing an array out of bounds (a spatial memory safety error), (2) accessing memory that has been deallocated (a temporal memory safety error), or (3) treating a data pointer as a function pointer (a type safety error).

Safe languages use a combination of compiler-enforced bounds checks (to counter spatial memory safety errors), automatic memory management (to counter temporal memory safety errors) and type checking (to counter type safety errors) to make sure that no program can ever run into such undefined behavior. Many safe languages compile to a virtual machine (like the Java Virtual Machine), but there is currently a growing interest in more C-like languages that compile to machine code and give the programmer more control over memory management, while still providing substantial safety guarantees. A prototypical example is the Rust programming language [MK14], strongly influenced by the Cyclone language [JMG+02].

Programming language safety is well understood [Pie02], but new challenges arise if such languages are used to build protected modules, as we will show further in the paper.

For this work-in-progress report, we will use a simple safe single-threaded dialect of C that supports global variables that are module-private (like static global variables in C), top-level function definitions, and type safe function pointers (like C# delegates). The only types are integers, the void type and function types. Section 3 contains examples of programs in this language. Formalizing the syntax and semantics of the language is straightforward.

## 2.2 Protected Module Architectures

Protected Module Architectures [MPP+08, MLQ+10, SP12] are a relatively recent countermeasure to protect software modules against attacks from the software context in which a module executes. This context includes both other modules of the program as well as higher privileged infrastructural software such as the operating system. They have been designed both for higher-end processors [MPP+08, SP12] – Intel's most recent Skylake processors provide support under the name of Intel Software Guard Extensions (Intel SGX) [Int14, MAB+13] – as well as for small micro-processors [SPP10, NAD+13, KSSV14, EDFPT12]. Essentially, a Protected Module Architecture provides hardware enforced memory access control that can be used to ensure that the state of a protected module can only be accessed by the code of that protected module.

As a small representative example, consider the program in Figure 1. The

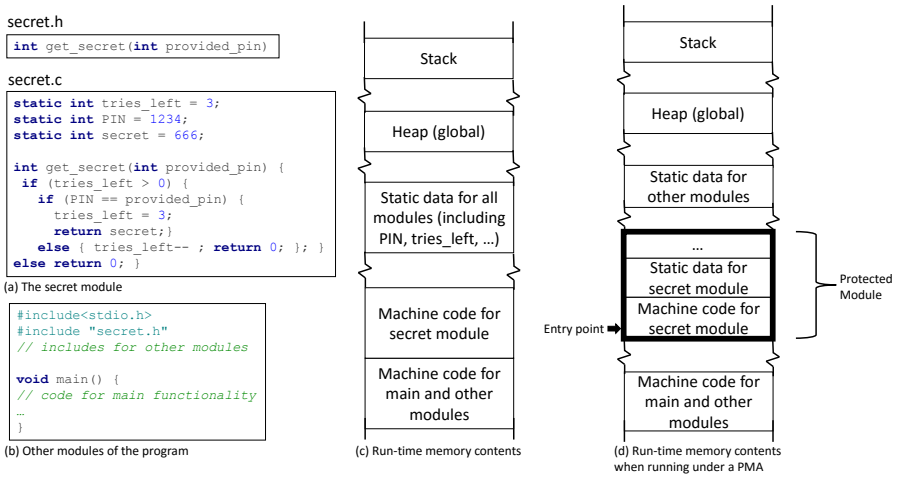


Figure 1: A program with a security critical module, and how it looks in memory at run-time with and without the protection of a Protected Module Architecture.

program has a single module (implemented in `secret.c`) that manages sensitive information, in this case the secret variable that should only be shown to users of the program who can provide a correct PIN. After three tries with an incorrect PIN, the module will refuse further attempts to protect against brute force attacks. The secret module exposes (in it's header file `secret.h`) only the `get_secret()` function, and hence access to the global variables in `secret.c` is restricted to the secret module at source code level. Other modules (including for instance the main() function) can only interact with the module through the `get_secret()` function. This module is a very simple example of a security critical module, like for instance the password manager in a browser or the implementation of a cryptographic protocol. These modules can be subject to memory scraping attacks.

To see this, consider the compiled version of this program at run-time. Part (c) of Figure 1 shows a schematic picture of the memory contents at run-time. If we now consider an attacker that can compromise some of the other modules of the program, or that can compromise the operating system, it is clear that the attacker can easily violate both integrity as well as confidentiality of the variables of the secret module.

Part (d) of Figure 1 shows how the secret module could be loaded in a protected module (or enclave). A protected module is essentially a segment of memory, that can contain both code and data. In addition, the module has one or more *entry points*[1], that are code addresses in the protected module.

[1]Some protected module architectures support only one entry point, but then multiple *logical* entry points can be layered on top of a single *physical* entry point.

The memory access control model of protected module architectures essentially enforces the following rules:

- When the Instruction Pointer (IP) is outside of the protected module, access to memory in the protected module is prohibited.
- When the IP is inside a protected module, access to memory within that module is allowed, but access to memory belonging to other protected modules is still prohibited.
- The only way for the IP to enter a protected module is by jumping to one of the designated entry points.

This simple access control model makes it possible for modules to guard access to their private state. As shown in Part (d) of Figure 1, the secret module could be compiled such that its machine code and its static data is stored within the protected module. Depending on the exact compilation algorithm, other state including for instance stack activation records for functions defined in the module can also be stored in the protected module. If we provide a single entry point to call the `get_secret()` function, then the variables `PIN`, `tries_left` and `secret` can only be accessed by the `get_secret()` function. Because of the memory access control, they can no longer be "scraped" from memory by malicious machine code in one of the other modules nor by kernel-level malware.

Given that Intel SGX is and will remain the dominant Protected Module Architecture for the foreseeable future, we will use the term "enclave" for the remainder of the paper.

# 3 Safe and unsafe modules

Safe languages protect against vulnerabilities *within* a module and Protected Module Architectures guard modules against attacks from *outside* of the module. Thus, it appears natural to combine both mechanisms. However, since the code in an enclave is typically only *part* of a program, and since safe languages only provide safety guarantees for whole programs written in that safe language, enclaves that are programmed in safe languages are *not* necessarily safe.

Consider the following example module in the simple safe language introduced in Section 2.1:

```
1 int count = 0;
2
3 void inc() {
4   count = count + 1;
5 }
6
7 int get_count() {
8   return count;
9 }
```

This implements a simple counter module where the context can only increment the counter, or read it out. If one compiles this module to an enclave in the way described in Section 2.2, the resulting module is safe in the sense that the context can never drive it into undefined behavior (assuming that the source code semantics specifies what should happen on integer overflow).

Now consider the following, somewhat more elaborate module:

```
1  int count = 0;
2  void (*obs)(int);
3
4  void set_observer(void o(int)) {
5    obs = o;
6  }
7
8  void inc() {
9    count = count + 1;
10   obs(count);
11 }
12
13 int get_count() {
14   return count;
15 }
```

If this module is compiled as outlined in Section 2.2, it is *not* safe. The entry point `set_observer()` accepts a function pointer as argument from the context. If the context is malicious, it can pass in an invalid function pointer, and then further behavior of the module will be undefined. By choosing the invalid function pointer carefully, an attacker could possibly succeed in making the module misbehave. Suppose for instance that the compiler represents function pointers as addresses of the start of the compiled code of the function. Then a malicious context can pass in any address within the module and have the module start executing code at that address on the call to `obs()`, thus launching a return-oriented-programming style attack [Sha07, CDD+10] against the module. For instance, if the module contains code like in Figure 1, such an attack could pass in a function pointer that points into the middle of the get_secret() function, and hence executes this function while skipping the validation tests.

Issues similar to the example above (where the context provides invalid function pointers), can also occur when the context passes in data pointers (the context could for instance tamper with the bounds information associated with a data pointer), or object references. Additional complications arise when code within the module reads from memory residing outside the module.

In this paper, we focus on the issue of handling invalid function pointers only to illustrate the essence of the problem.

# 4 Towards provably safe enclaves

Low-level attacks like the example above exist because of the abstraction gap between the machine code level and the source code level. The key to defending against these attacks is to make sure that any interaction that a malicious context could have with a compiled module (running in an enclave) at the machine code level will always have well-defined behavior according to the source code level semantics.

Proving such safety requires us to (1) model the context-enclave interactions at machine code level, (2) define the source code semantics modularly, and (3) show that *any* context-enclave interaction at machine code level can be interpreted as a source code level interaction for which the semantics defines corresponding behavior. We address these three items in the following three subsections.

## 4.1 Machine-code level context-enclave interactions

To ensure that behavior of an enclave is always well-defined, we have to make sure that *any* potential interaction the context could have with the enclave is covered by the definition of the semantics of the module that executes inside the enclave. For our simple model of enclaves, the following interactions are possible:

1. **jump-in** $e(v_i)$: jump from the context to an entry point $e$ of the enclave, with values $v_i$ in the processor registers. Assuming a 32-bit processor, both $e$ and the $v_i$ will be 32-bit words. $i$ ranges from 0 to $N-1$ where $N$ is the number of registers in the processor, and $v_i$ is the sequence of values in these registers at the time of the jump.

2. **jump-out** $a(v_i)$: jump from within the enclave to an address $a$ outside of the module, with values $v_i$ in the processor registers.

3. **read-out** $v = *a$: read from within the enclave the contents of memory at an address $a$ outside of the module, resulting in value $v$.

4. **write-out** $*a = v$: write the value $v$ from within the enclave to an address $a$ outside of the module.

Only **jump-in** interactions are initiated by the context, the other three are initiated by the enclave. For simplicity, we will limit our attention in this paper to compilers that never perform **read-out** or **write-out** interactions. Handling them is left for future work; the key additional challenge is to make sure that data read from the context is checked as defensively as data passed into the module by means of a **jump-in** event. Under this simplifying assumption, the only possible interactions between enclave and context are **jump-in** or **jump-out** interactions.

An *enclave specification* is a labeled transition system, consisting of:

- A set $ES$ of enclave states, the disjoint union of active enclave states $AS$ (the processor is executing in the enclave) and passive enclave states $PS$ (the processor is executing outside the enclave).
- A set of actions $A$, the disjoint union of entry actions **jump-in** $e(v_i)$, exit actions **jump-out** $a(v_i)$ and $\tau$ (the internal action).
- A transition relation $\subset ES \times A \times ES$, written $E \rightarrow^{\alpha} E'$, such that:
  - Entry actions transition from a passive state to an active state
  - Exit actions transition from an active state to a passive state
  - The internal action transitions from active states to active states

An enclave specification is *safe* (or *complete*), if the following holds:

- For any passive state $P$ of the enclave, and for any entry action $e$, $P$ transitions under $e$ to some (necessarily active) state $A$.
- If $P$ transitions under $e$ to $A$, $A$ never gets stuck, i.e. either infinitely performs $\tau$ actions, or will eventually perform an exit action.

Our objective is to make sure that a source code module defines a safe or complete enclave specification: the behavior of the enclave on any (even maliciously chosen) interaction with the context should follow from the source code semantics or lead to a well-defined error state.

Note that this notion of safety guarantees the absence of low-level vulnerabilities that are caused by undefined behavior, but it does *not* guarantee the absence of (for instance) undesired information leaks from the enclave, or it does not give any availability guarantees. We discuss this in more detail in Section 5.

## 4.2 A modular semantics of the source code

Since enclaves only contain a single software module, not necessarily a whole program, we have to define a *modular* semantics of the source language. A good example of such a modular semantics is the trace-based semantics of Java modules defined by Jeffrey and Rathke [JR05].

For our simple source language, we define the set of *values*[2] to be the disjoint union of (1) the signed 32-bit integers, (2) the value void, and (3) the set of function values. The set of function values consists of the disjoint union of (1) the function names of the functions defined in the module (the *internal* function values), and (2) an infinite set of abstract *external* function values.

The modular semantics defines the source code level interactions that the module can have with its source code context, and defines the traces of such interactions that are considered valid behaviors of the module. For instance, interactions for our simple language will be: (1) calls from the context of functions

[2]Note that we are now defining values at the level of the source code. These depend on the source language and will usually be more abstract than values at machine code level.

defined in the module (in-calls), (2) returns from such function calls (return-outs), (3) calls from within the module to a function defined outside the module (out-calls) and (4) returns from these calls (return-ins).

The semantics of a module $M$ defines the valid traces of such interactions for $M$. For instance, the following traces are valid traces of the simple Counter module above:

- in-call inc(), return-out void, in-call get_count(), return-out 1
- in-call inc(), return-out void, in-call inc(), return-out void, in-call get_count(), return-out 2.

The following trace is a valid trace for the Counter module with an observer, where $f_1$ is an external function value:

- in-call set_observer($f_1$), return-out void, in-call inc(), out-call $f_1(1)$, return-in void, return-out void

The definition of a formal modular semantics for our simple C dialect should be a straightforward adaptation of the modular semantics for Java Jr. [JR05].

## 4.3 Defensive validation and interpretation of context-enclave interactions as source code interactions

Safety of an enclave requires well-defined reaction of the enclave to *any* **jump-in** action, but the modular source code semantics only defines the reaction to (well-typed) source-code level in-call or return-in actions. Therefore, the key to make a compiled enclave safe is to define the machine-code level *calling conventions*, and to defensively validate every **jump-in** action to make sure that under these calling conventions the **jump-in** action can be interpreted as a well-typed source code in-call or return-in.

For our simple language we define the following. A source code module will be compiled to an enclave that has an entry point $e_f$ for each function $f$ defined in the module, and one additional entry point $e_{return}$ that we call the return entry point. The calling convention to call function $f$ with arity $n$ will be to jump to $e_f$ passing arguments to $f$ in registers 0 to $n-1$, and the return address in register $n$. For simplicity, we assume that the processor has a sufficiently large number of registers. A return is performed by jumping from the module to the specified return address, with the result value in register 0.

Now (assuming a 32-bit processor), we define how to interpret machine code values (i.e. 32-bit words) as source code values of a specific expected type:

- Where the source module expects an integer value, the word is interpreted as a signed 32-bit integer.
- Where the source module expects void, any 32-bit word is interpreted as the value void.

- Where the source module expects a function pointer of a specific type, the word is interpreted as follows:
  - If the word is an address outside the enclave, the word is interpreted as an abstract external function value of the appropriate type.
  - If the word is an address inside the enclave, then it is considered valid only if it is equal to an entry point $e_f$ for a function $f$ defined in the module and of the appropriate type, and it is then interpreted as the internal function value $f$.

Next, we define how low-level context-enclave interactions relate to source code module interactions, making sure that *any* possible entry from the context to the enclave (that could hence possibly be malicious) either finishes execution with a run time error during validation, or can be related to a valid source code level module entry.

- A **jump-in** $e(v_i)$ interaction correspond to source code interactions as follows:
  1. If $e$ is the entry point $e_f$ corresponding to function $f$ with arity $n$, then $v_0$ up to $v_{n-1}$ are interpreted as source code values $s_0$ to $s_{n-1}$ as specified above using the signature of $f$ to determine what source code types to expect. If some of these values cannot be given a valid interpretation, the **jump-in** interaction is interpreted as a run time error.

     Next, the word $v_n$ (the return address according to the calling conventions) is checked to be an address outside the enclave. If this check fails, the **jump-in** interaction is interpreted as a run time error. If the check succeeds, then:
     - this **jump-in** interaction corresponds to an in-call of $f$ with actual parameters $s_i$, and
     - $v_n$ is used as the address to jump to on completion of this in-call. More precisely, when the source code semantics specifies that the return-out corresponding to this in-call should happen, the compiler will generate a **jump-out** event to address $v_n$, with the return value in the first register of the processor.
  2. If $e$ is the entry point $e_{return}$, then $v_0$ is interpreted as a source code value $s_0$ using the return type of the most recent out-call as the expected source code type. If this interpretation is valid, the interaction is interpreted as the return-in interaction with return value $s_0$, otherwise the **jump-in** interaction is interpreted as a run time error.
- When the source code semantics specifies that an out-call interaction should happen, the compiler generates a **jump-out** $a(v_i)$ interaction where $a$ is the external function value (which is an address outside the enclave as specified above), passing the values $v_i$ in the processor registers as actual parameters, and the $e_{return}$ entry point as return address.

Consider again the Counter with observer example. With the defensive checks specified informally above in place, the module is now safe: if the context provides an invalid function pointer within the module to the `set_observer()` function, this will lead to an immediate run time error, and hence launching a return-oriented-programming style attack against the module is now also ruled out. Of course, the context can still provide invalid *external* function pointers, but these can only lead to undefined behavior in the context, never to undefined behavior in the module.

### 4.4 Proving safety

Formalization of the problem, and proofs are work-in-progress. The steps required are:

1. Formalization of the source language. We should define a modular semantics for the source, possibly also showing that this modular semantics is *compatible* with pre-existing definitions of a whole program semantics.

2. Formalization of the algorithm for defensive validation of **jump-in** events (informally described in Section 4.3).

3. A proof that the combination of this algorithm with the modular source code semantics defines a safe enclave specification.

We are confident that the proofs work out for the case of the simple language we considered in this paper, but significant challenges remain to address more advanced language features. We intend to gradually extend the simple source language with language features such as bounds-checked arrays, objects and multi-threading, as well as with typing features such as ownership types.

## 5 Related Work

The work reported on in this paper is related to existing work on secure (fully abstract) compilation, and work on language interoperability.

Fully abstract compilation [Aba99] studies the problem of compiling from source languages to target languages (including, e.g. machine code) such that, roughly speaking, attackers at the target level have no more power than attackers at the source level. More formally, fully abstract compilation preserves and reflects contextual equivalence. Fully abstract compilation has recently been studied intensively [ASJP12, BA15, PAS+15, DPP16, JHA+15], including compilation towards protected module architectures. Compared to our approach, a fully abstract compiler provides strictly stronger guarantees. For instance, confidentiality properties (like noninterference properties) are preserved by fully abstract compilation, but not by our safe compiler. Yet, full abstraction is much harder to achieve, and is a property of the entire compiler, whereas safety can be proven without formalizing a full compiler. We also conjecture that safe

compilers can be more efficient and would incur less runtime overhead on the generated code than fully abstract compilers.

Work on language interoperability, foreign function interfaces or contracts [MF07, FF02, AJP15, Ahm15] studies interoperability between different languages, but in most work the emphasis is either (1) on interoperability between two safe languages where one is untyped and one is typed, and where contracts check the interactions between the untyped and typed world, or (2) on interoperability ("plumbing") between a safe language and a trusted unsafe language that implements a native API for the safe language.

Finally, an important additional motivation for studying enclave safety is the support for state continuity that researchers are developing [PLD+11, SJP14]. A state-continuous enclave is protected from so-called rollback attacks that revert the enclave's internal state to a stale version of that state, but at the same time state-continuous enclaves are enforced to process all their inputs and this can be problematic if an input is provided that makes the enclave crash. Safety of enclaves provides stronger assurance that the enclave implements sufficiently defensive input checking.

# 6 Conclusion

Protected module architectures open the possibility to execute software modules in enclaves and rely on a very small trusted computing base for their correct execution. However, these architectures do not protect against vulnerabilities in the modules themselves. We have reported on work in progress towards guaranteeing *safety* of protected enclaves, in the sense that the behavior of the enclave at machine code level is fully determined by the source code for the enclave. Although our approach provides weaker security guarantees than fully abstract compilation, we expect that safe enclaves would execute more efficiently. This enables the use of safe languages together with protected module architectures in domains where the performance overhead would otherwise be prohibitive and where the stronger guarantees provided by fully abstract compilation are not required. Use cases in the fields of embedded computing, ubiquitous computing, or the Internet of Things can benefit from safe enclaves.

**Acknowledgments**

This research is partially funded by project grants from the Research Fund KU Leuven, and from the Research Foundation Flanders (FWO).

# References

[Aba99] Martín Abadi. Protection in programming-language translations. In *Secure Internet Programming, Security Issues for Mobile and Distributed Objects*, pages 19–34, 1999.

[Ahm15] Amal Ahmed. Verified compilers for a multi-language world. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, pages 15–31, 2015.

[AJP15] Pieter Agten, Bart Jacobs, and Frank Piessens. Sound modular verification of C code executing in an unverified context. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 581–594, New York, NY, USA, 2015. ACM.

[ASJP12] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors. In *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 171–185, 2012.

[BA15] William J. Bowman and Amal Ahmed. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 101–113, New York, NY, USA, 2015. ACM.

[CBB+01] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. Formatguard: automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th conference on USENIX Security Symposium*, SSYS'01, pages 1–9, Berkeley, CA, USA, 2001. USENIX Association.

[CDD+10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS'10, pages 559–572, New York, NY, USA, 2010. ACM.

[DPP16] Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 164–177, 2016.

[EDFPT12] 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.

[EYP10] Úlfar Erlingsson, Yves Younan, and Frank Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer, 2010.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Not.*, 37(9):48–59, September 2002.

[Huq15] Numaan Huq. PoS RAM scraper malware: Past, present, and future. Technical report, Trend Micro, 2015.

[Int14] Intel. *Intel Software Guard Extensions Programming Reference.* 2014.

[JHA+15] Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, Benjamin C. Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Towards a fully abstract compiler using Micro-Policies: Secure compilation for mutually distrustful components. Technical Report, arXiv:1510.00697, 2015.

[JMG+02] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[JR05] Alan Jeffrey and Julian Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. In *Proceedings of the 14th European Conference on Programming Languages and Systems*, ESOP'05, pages 423–438, Berlin, Heidelberg, 2005. Springer-Verlag.

[KSSV14] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 10:1–10:14, New York, NY, USA, 2014. ACM.

[MAB+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP'13, page 8, New York, NY, USA, 2013. ACM.

[MF07] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 3–10, New York, NY, USA, 2007. ACM.

[MK14] Nicholas D. Matsakis and Felix S. Klock, II. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, New York, NY, USA, 2014. ACM.

[MLQ+10] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[MPP+08] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, pages 315–328. ACM, April 2008.

[NAD+13] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 479–498, Washington, D.C., 2013. USENIX.

[PAS+15] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. *ACM Trans. Program. Lang. Syst.*, 37(2):6:1–6:50, April 2015.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PLD+11] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2011.

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[SJP14] Raoul Strackx, Bart Jacobs, and Frank Piessens. ICE: A passive, high-speed, state-continuity scheme. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14. ACM, 2014.

[SP12] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 2–13, New York, NY, USA, 2012. ACM.

[SPP10] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. In Sushil Jajodia and Jianying Zhou, editors, *SecureComm*, volume 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 344–361. Springer, 2010.

[SYP+09] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EuroSec'09, pages 1–8, New York, NY, USA, 2009. ACM.