Towards Automatic Compartmentalization of C Programs on Capability Machines

Stelios Tsampas[1], Akram El-Korashy[2], Marco Patrignani[2], Dominique Devriese[1], Deepak Garg[2] and Frank Piessens[1]
[1] imec-Distrinet, KU Leuven, Belgium; email name.surname@cs.kuleuven.be
[2] MPI-SWS, Germany; email {elkorashy, marcopat, dg}@mpi-sws.org

***Abstract*—Capability-based protection mechanisms can offer fine-grained memory protection (through *memory capabilities*), as well as fine-grained protection of general software-defined objects (through *object capabilities*). Because of the similarity that capabilities have to the notion of *pointer* in C, compilers can use the capability mechanisms offered by the target platform to generate code that is more resilient to attack. For instance, C arrays can be compiled to memory capabilities thus providing hardware-enforced spatial safety guarantees and hence strong resilience against buffer overflow attacks. State-of-the-art capability-based systems (like for instance the CHERI system [1]) come with a C compiler that provides such safety guarantees.**

But such safe compilation does not provide security guarantees for an attacker model where an attacker can compromise part of the code of an application, for instance by providing a malicious library, possibly in compiled form. An application is still executed in a *single* protection domain. The mechanism of object capabilities can be used to remedy this: object capabilities support *compartmentalization* of an application where different parts of the application can be executed in different protection domains, and hence one part of the application can be protected against malicious behaviour in other parts. However, to the best of our knowledge, state-of-the-art C compilers provide no automatic support for such compartmentalization. In CHERI, support for such compartmentalization is offered as an API [2].

This paper reports on our work-in-progress on the definition, implementation and evaluation of a compiler that *automatically* compartmentalizes the programs it compiles, essentially by executing each C compilation unit in a separate protection domain. We provide a formal definition of our compiler, and an implementation on CHERI as a source-to-source compiler that can detect and insert the necessary invocations to CHERI's API for compartmentalization. We illustrate the security properties of the compiler by means of examples and discuss our work-in-progress on formalizing and proving these security properties.

This paper uses colours to distinguish elements of different languages. For a better experience, please print/view it in colour.

I. Introduction

Hardware and OS level protection primitives like virtual memory, ASLR, or SGX enclaves play an important role in protecting against the exploitation of low-level software vulnerabilities. One interesting application of such protection primitives is to have a compiler for a higher-level language use them to enforce abstractions offered by the higher-level language. Such *secure compilers* can provide interesting security guarantees, such as the preservation of certain classes of security properties of the source program against strong attacker models where an attacker can interact with the compiled program at the hardware or OS level of abstraction. Over the past decade, several researchers have investigated techniques for secure compilation based on ASLR [3], [4], protected module architectures like Intel SGX enclaves [5]–[7], or general metadata tracking hardware such as the PUMP machine [8], [9].

Capabilities are a hardware level protection mechanism with a very rich history in OS security [10]–[13]. Capabilities can be thought of as hardware protected unforgeable pointers, either to memory segments (code and data capabilities), or to software defined objects (object capabilities). Since capabilities can potentially provide very fine-grained protection at reasonable performance cost, there has recently been renewed interest in them. A prominent recent example is the CHERI system [1], [14]. The CHERI processor is a capability-based variant of the MIPS architecture [15] that offers both virtual memory as well as instruction-level support for fine-grained protection within each virtual memory address space. It comes with a software stack including a capability-aware variant of FreeBSD and a CHERI-targeted LLVM compiler port. In its most secure mode, this CHERI compiler represents C pointers as bounds-checked unforgeable memory capabilities at run-time, and thus provides strong spatial safety guarantees [2]. The compiler also supports less secure modes that make interoperability with legacy code easier.

CHERI also supports object capabilities. The key difference with memory capabilities is that these object capabilities can also enforce some notion of *encapsulation*: a code module can hand out an object capability that provides access to a data structure defined by that code module, and maintain the guarantee that the data structure can only be accessed through functions provided by the code module - thus enabling the module to enforce invariants or do information hiding.

Object capabilities are a very powerful primitive. CHERI implements them using a combination of hardware support, kernel support and a user-space library, and it offers them to software developers as an API that developers can use to create protection domains within a single virtual address space. The CHERI papers outline several interesting design patterns and use cases of that API for exploit mitigation. However, the existing CHERI compiler does not use object capabilities for securing the compiled code itself: source programs must be modified and/or annotated to benefit from the security offered by object capabilities.

The main contribution of this paper is a compiler that

provides additional security properties by using the target platform's support for object capabilities. For a C program consisting of several *modules* where some modules can be provided by an attacker in binary (compiled) form, our compiler protects the integrity and secrecy of the private data of the other modules (those not provided by the attacker). We define a module as C code that is *compiled* together. (An executable binary is constructed by *linking* one or more such compiled units together.) The private data of a module is all global variables declared using the modifier "static" within the module. Under C semantics, such variables are not visible outside the module in which they are defined, but neither common C compilers nor the existing CHERI compiler guarantee this property for compiled code.

More specifically, the contributions of this paper are:

- We provide a simple formal model of a target platform that supports both memory and object capabilities.
- We formally define a compiler from a simple C-like language to that target platform that uses object capabilities to place each C module in a separate protection domain.
- We implement our compiler on the CHERI platform as a source-to-source compiler that injects the necessary calls to CHERI's object capability API.
- We show by means of examples that our compiler protects against additional attacks that the standard CHERI compiler (even in its most secure mode) does not address.
- We discuss conjectures about the formal security properties that our compiler satisfies.

This paper is a work-in-progress paper, and we do not yet have full proofs for our conjectures, nor do we have an experimental evaluation of our implementation. However, the work has progressed sufficiently to have some confidence in the results: our implementation can compile example C programs that illustrate the additional security properties offered by the compiler.

II. The target language

Our target language models a platform that supports memory and object capabilities, and is strongly inspired by the CHERI system [1], [14], a MIPS-based capability-machine architecture. CHERI offers fine-grained memory capabilities through hardware support, and it offers object capabilities through a combination of hardware support, kernel support and a user-space library (libcheri).

Accordingly, we model in this section a low-level target language, which we call **LLibcheri**. This language includes abstractions that mimic the interfaces offered by libcheri as well as CHERI's capabilities. Our model of capabilities draws heavily from a prior model of a capability machine [16].

A. A target language with capabilities and libcheri

Our target language, **LLibcheri**, uses unstructured control flow constructs. Programs in **LLibcheri** own unforgeable memory capabilities that mediate memory operations. They also own object capabilities which are used to invoke functions. A trusted call stack manages such function invocation. For the sake of readability, we typeset target language terms in *orange*, source terms will be typeset in *blue*.

B. Values, expressions, and commands

Values in **LLibcheri** are denoted by $\mathcal{V} = \mathbb{Z} \cup Cap$ and range over integers $\mathbb{Z}$ and memory capabilities $Cap = \{\kappa, \delta\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}$. Memory capabilities are code or data capabilities, denoted by κ and δ respectively, where the κ-labeled elements describe a range of the code memory $\mathcal{M}_c$ and an offset within this range, and the δ-labeled elements describe the same for the data memory $\mathcal{M}_d$. We separate capabilities from integers to model unforgeability of capabilities, which is a key design feature in CHERI [1], [14]. Formal arguments of how this unforgeability is guaranteed by the CHERI architecture are beyond the scope of this paper, but can be found in [16].

Definition 1 (Valid code/data capability)**.** *We use the judgment* $\vdash_x (\sigma, s, e, off)$ *to mean that* $\sigma = x$ *and that* $off \in [0, e - s)$

Validity of a code/data capability (σ, s, e, off) ensures that it is of the intended capability type x, and that its offset lies within the legal range that it prescribes.

Definition 2 (Subset relation and disjoint capabilities)**.** *We also use the judgment* $(\sigma, s_1, e_1, \_) \subseteq (\sigma, s_2, e_2, \_)$ *to mean* $[s_1, e_1) \subseteq [s_2, e_2)$ *and similarly* $(\sigma, s_1, e_1, \_) \cap (\sigma, s_2, e_2, \_) = \emptyset$ *to mean that* $[s_1, e_1) \cap [s_2, e_2) = \emptyset$.

And we define the function `inc`: $Cap \times \mathbb{Z} \to Cap$ as $\texttt{inc}((\sigma, s, e, off), z) \stackrel{def}{=} (\sigma, s, e, off + z)$ which increments the offset of a capability by z.

Memory notation: Code and data memories ($\mathcal{M}_c : \mathbb{N} \to Cmd$ and $\mathcal{M}_d : \mathbb{N} \to \mathcal{V}$) are maps from addresses –that are natural numbers– to commands and values respectively. Memory values have been described above. Below we describe expressions and commands. But we first fix some notation regarding code and data memories:

- We refer to the type $\mathbb{N} \to Cmd$ as $CodeMemory$ and to the type $\mathbb{N} \to \mathcal{V}$ as $DataMemory$.
- The operator $\uplus$ is used throughout the paper to refer to disjoint union of sets or functions. For functions f and g with $\texttt{dom}(f) \cap \texttt{dom}(g) = \emptyset$, the function $(f \uplus g)$ has domain $\texttt{dom}(f) \cup \texttt{dom}(g)$ and is defined as $(f \uplus g)(x) \stackrel{def}{=} f(x)$ if $x \in \texttt{dom}(f)$, and $g(x)$ otherwise. We use the notation $\mathcal{M}_c = \biguplus_i \mathcal{M}_{c_i}$ to mean the linking of several code memories $\mathcal{M}_{c_i}$ with disjoint mapped addresses into one code memory $\mathcal{M}_c$, and similarly for other constructs that are maps or functions.

*Commands in **LLibcheri**:* Commands Cmd in **LLibcheri** are the following. Figure 1 shows the semantics of these commands.

- Assign $\mathcal{E}_L$ $\mathcal{E}_R$ which evaluates the expression $\mathcal{E}_R$ to a value $v \in \mathcal{V}$, evaluates the expression $\mathcal{E}_L$ to a data capability value $c \in \{\delta\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}$, and stores in the data

Fig. 1. Evaluation of commands Cmd in **LLibcheri**

(assign)

$$\frac{\vdash_\kappa \mathsf{pcc} \quad \mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, 1) \quad \mathcal{M}_c(\mathsf{pcc}) = \mathsf{Assign}\ \mathcal{E}_L\ \mathcal{E}_R \quad \mathcal{E}_R, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \quad \mathcal{E}_L, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow c \quad \vdash_\delta v \Longrightarrow v \cap \mathsf{stc} = \emptyset \quad \vdash_\delta c \quad \mathcal{M}'_d = \mathcal{M}_d[c \mapsto v]}{\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{nfree}\rangle \rightarrow \langle \mathcal{M}_c, \mathcal{M}'_d, stk, imp, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}', \mathsf{nfree}\rangle}$$

(jump1)

$$\frac{\vdash_\kappa \mathsf{pcc} \quad \mathcal{M}_c(\mathsf{pcc}) = \mathsf{Jump}\ \mathcal{E}_{cond}\ \mathcal{E}_{cap} \quad \mathcal{E}_{cond}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \quad v \in \mathbb{Z} \setminus \{0\} \quad \mathcal{E}_{cap}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow c \quad c \in \{\kappa\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z} \quad \mathsf{pcc}' = c}{\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{nfree}\rangle \rightarrow \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}', \mathsf{nfree}\rangle}$$

(jump0)

$$\frac{\vdash_\kappa \mathsf{pcc} \quad \mathcal{M}_c(\mathsf{pcc}) = \mathsf{Jump}\ \mathcal{E}_{cond}\ \mathcal{E}_{cap} \quad \mathcal{E}_{cond}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \quad v = 0 \quad \mathsf{pcc}' = \mathtt{inc}(\mathsf{pcc}, 1)}{\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{nfree}\rangle \rightarrow \langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}', \mathsf{nfree}\rangle}$$

(cinvoke)

$$\frac{\vdash_\kappa \mathsf{pcc} \quad \mathcal{M}_c(\mathsf{pcc}) = \mathsf{Cinvoke}\ mid\ fid\ \overline{e} \quad stk' = \mathtt{push}(stk, (\mathsf{ddc}, \mathsf{stc}, \mathsf{pcc})) \quad \mathsf{nfree}' = \mathsf{nfree} + \phi \quad \overline{e}(i), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_i\ \forall i \in [0, nArgs) \quad \mathcal{M}'_d = \mathcal{M}_d[\mathsf{nfree} - nArgs + i \mapsto v_i\ \forall i \in [0, nArgs)] \quad \mathsf{stc}' = (\delta, \mathsf{nfree} - nArgs, \mathsf{nfree}', \mathsf{nfree}) \quad (c, d, offs) = imp(mid) \quad \mathsf{ddc}' = d \quad \mathsf{pcc}' = \mathtt{inc}(c, offs(fid))}{\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{nfree}\rangle \rightarrow \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nfree}'\rangle}$$

(creturn)

$$\frac{\vdash_\kappa \mathsf{pcc} \quad \mathcal{M}_c(\mathsf{pcc}) = \mathsf{Creturn} \quad stk', (\mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}') = \mathtt{pop}(stk) \quad \mathsf{nfree}' = \mathsf{nfree} - \phi \quad \mathcal{M}'_d = \mathcal{M}_d[n \mapsto 0\ \forall n \in [\mathsf{nfree}', \mathsf{nfree})]}{\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{nfree}\rangle \rightarrow \langle \mathcal{M}_c, \mathcal{M}'_d, stk', imp, \mathsf{ddc}', \mathsf{stc}', \mathsf{pcc}', \mathsf{nfree}'\rangle}$$

memory $\mathcal{M}_d$ the value v at the address indicated by c (the address $(s + o)$ for $c = (\delta, s, e, o)$).

- Jump $\mathcal{E}_{cond}$ $\mathcal{E}_{cap}$ is a conditional jump which evaluates the expression $\mathcal{E}_{cond}$ to a value $v \in \mathbb{N}$, and if $v \neq 0$, then it evaluates $\mathcal{E}_{cap}$ to a code capability value $c \in \{\kappa\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}$, and sets pcc to c. Otherwise ($v = 0$), nothing is done.
- Cinvoke mid fid $\overline{e}$ [1] , which is used to invoke an object capability. Our target platform is configured (in the imp component of the initial machine state, see below) with a fixed number of object capabilities identified by module identifiers mid, and each object capability supports invocation of a finite number of functions specified by function identifiers fid. Each secure call gets access via stc to a new data stack frame of a fixed constant size ϕ (for local use), in addition to a memory region of size $nArgs$, which should contain the parameter values that the caller passes.
- CReturn, which is used to return from a call that has been performed using Cinvoke. The rules cinvoke and creturn in fig. 1 specify the exact operations performed to push and pop the necessary capabilities to/from the trusted stack.

A state $\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{nfree}\rangle$ of a program in **LLibcheri** consists of:

- code and data memories, $\mathcal{M}_c$ and $\mathcal{M}_d$ as defined earlier (We define $\mathcal{M}_d((\delta, s, e, o)) \stackrel{def}{=} \mathcal{M}_d(s + o)$, and similarly for update expressions and for $\mathcal{M}_c$ with κ-labeled values.),
- a trusted call stack $stk : \overline{Cap}$[3], which is a list of triples of capabilities that stores the history of the values of ddc, stc, pcc at the call locations.
- a map of imports $imp : \mathbb{N} \rightarrow CapObj$ that for each module identifier, keeps an object capability ($CapObj = (\{\kappa\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}) \times (\{\delta\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}) \times (\mathbb{N} \rightarrow \mathbb{N})$). An object capability consists of
  - a code capability that grants access to the module's code region in $\mathcal{M}_c$,
  - a data capability that grants access to the module's data region in $\mathcal{M}_d$,
  - and an offsets map, that for each function identifier in the module, specifies the offset within the module's code memory at which the function's code starts (i.e., this map of offsets describes the legitimate entry points to the module).
- three capability registers/variables:
  - ddc : $\{\delta\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}$, the data capability (which specifies the region in the data memory $\mathcal{M}_d$ that is private to the active module),
  - stc : $\{\delta\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}$, the stack-data capability (which specifies the region in the data memory $\mathcal{M}_d$ that corresponds to the current activation record),
  - and pcc : $\{\kappa\} \times \mathbb{N} \times \mathbb{N} \times \mathbb{Z}$, the program counter capability (which specifies the region in the code memory $\mathcal{M}_c$ in which the currently-executing module is defined);

[1] We use the notation $\overline{x}$ to denote that x has a list type. And we also use the same notation for types (i.e., as a type constructor). For instance, we write $\overline{\mathbb{N}}$ to denote the type of lists of natural numbers.

- and a marker $\mathsf{nfree} : \mathbb{N}$ that holds the first non-allocated address in $\mathcal{M}_d$.

It is worth noting that the map of imports imp is fixed at load time, and its contents are not modified by any instruction. But one could imagine an extension to the language enabling private memory to be allocated at run-time and to be shared at run-time as well. We leave this for future work.

The syntax of the language enables the use of capabilities that are expressible in terms of three distinguished names, "ddc", "stc", and "pcc" denoting *data capability*, *stack capability*, and *program counter capability*, respectively.

Expressions in **LLibcheri** are denoted by the grammar $\mathcal{E} ::= \mathbb{Z} \mid \mathtt{ddc} \mid \mathtt{stc} \mid \mathtt{pcc} \mid \mathtt{inc}(\mathcal{E}, \mathbb{Z}) \mid \mathtt{deref}(\mathcal{E}) \mid \mathcal{E} \oplus \mathcal{E}$ where $\oplus ::= + \mid - \mid *$, and $\mathbb{Z}$ is the set of integers. $\mathtt{ddc}, \mathtt{stc}$ and $\mathtt{pcc}$ are the distinguished names for the corresponding capabilities. $\mathtt{inc}(\mathcal{E}, \mathbb{Z})$ increments the offset of a capability value. $\mathtt{deref}(\mathcal{E})$ evaluates to the value at the memory address pointed to by a capability only if it is a valid capability according to Definition 1. The evaluation of expressions $\mathcal{E}$ to values $\mathcal{V}$ is given by rules of the form $\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow \mathcal{V}$ listed in fig. 2.

C. Target setup, and initial and terminal states

Having defined the program state, we now define a target setup $TargetSetup = CodeMemory \times DataMemory \times (\mathbb{N} \to CapObj)$ as a triple of code memory, data memory, and imports map. The target setup can be seen as a target module except that we do not require any well-formedness conditions on a target setup or on the linking of setups because we want them to model low-level attackers as well as legitimate modules. We also define the linking $t_1 \uplus t_2$ of two target setups $t_1 = (\mathcal{M}_{c1}, \mathcal{M}_{d1}, imp_1)$, $t_2 = (\mathcal{M}_{c2}, \mathcal{M}_{d2}, imp_2) \in TargetSetup$ to be the component-wise linking $(\mathcal{M}_{c1} \uplus \mathcal{M}_{c2}, \mathcal{M}_{d1} \uplus \mathcal{M}_{d2}, imp_1 \uplus imp_2)$, which is defined only if:

1) the component-wise linking is defined for all three components,
2) and $\forall c_1 \in \mathtt{range}(imp_1), c_2 \in \mathtt{range}(imp_2).\, c_1 \cap c_2 = \emptyset$ where for $c, c' \in CapObj$, $c \cap c' = \emptyset \stackrel{def}{=} c.1 \cap c'.1 = \emptyset \wedge c.2 \cap c'.2 = \emptyset$ and disjointness of capabilities is as in Definition 2.

A program state $\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{nfree} \rangle$ is **initial** for target setup $(\mathcal{M}_c, \mathcal{M}_d, imp)$ if all of the following hold:

1) $stk = \mathtt{nil}$
2) $\mathsf{nfree} >$ the maximum address protected by the data capabilities of objects $c \in \mathtt{range}(imp)$
3) $\mathsf{stc} \cap \mathsf{ddc} = \emptyset$
4) **either** $\forall c \in \mathtt{range}(imp).\, (\mathsf{pcc}, \mathsf{ddc}, \_) \cap c = \emptyset$ **or** $\exists (cc, dc, \_) \in \mathtt{range}(imp).\, \mathsf{pcc} \subseteq cc \wedge \mathsf{ddc} \subseteq dc$

An **initial** state is one where the stack is empty, the free memory marker captures the correct amount of allocated memory, the stack data region is disjoint from the module's private data region, and the current code and data capabilities give privilege to at most one imported module (the one that supposedly started execution). We refer to a state s that is **initial** for setup t as $t \vdash_i s$.

A program state $\langle \mathcal{M}_c, \mathcal{M}_d, stk, imp, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc}, \mathsf{nfree} \rangle$ is **terminal** if

1) $\vdash_\kappa \mathsf{pcc}$
2) $\mathcal{M}_c(\mathsf{pcc}) = \mathsf{Creturn}$
3) $stk = \mathtt{nil}$

We refer to a state s that is terminal as $\vdash_t s$.

Given two target setups $t_1, t_2 \in TargetSetup$, we write $t_1[t_2] \Downarrow$ (convergence) to mean that $t_1 \uplus t_2$ is defined, and that $\forall\, s.\, t_1 \uplus t_2 \vdash_i s \implies \exists s_t.\, s \to^* s_t \wedge \vdash_t s_t$ where $\to^*$ is the reflexive transitive closure of the evaluation relation defined in fig. 1. Conversely, we write $t_1[t_2] \Uparrow$ (divergence or getting stuck) to mean that $t_1 \uplus t_2$ is defined, and that $\exists\, s.\, t_1 \uplus t_2 \vdash_i s \wedge \nexists s_t.\, s \to^* s_t \wedge \vdash_t s_t$.

D. Summary of target language features

Our model, **LLibcheri**, aims to model the essential security features provided by the CHERI hardware architecture and its runtime library, libcheri. In particular, call invocations between mutually distrustful components is a core feature of CHERI, which can be used to attain compartmentalized execution [16]. Passing parameters of function calls while ensuring non-retention of access to the stack frame of the callee after the call has returned is also a core feature of CHERI that we model in our language using the stack capability, and a restriction on storing the stack capability in memory (note that the rule assign categorically prohibits storing the stack capability in memory). In the actual CHERI architecture, these restrictions can be implemented using what is called the "permissions field" on capabilities. Here, we abstract a bit by modeling specific uses of this field rather than the field itself. Formal arguments showing that the permissions field can actually be used to attain our abstractions already exist in prior work [16], [17].

III. Problem Statement and Overview of our Solution

A. Standard compilation and its security issues

Consider the following C program with two modules.

```c
// module 1
int f2(void);
int g2(void);

static int gv1 = 0;

int f1(void)
{
  [..]
  f2();
  [..]
}
```

Listing 1. module1.c

```c
// module 2
int f1(void);

static int gv2 = 0;

int f2(void)
{
  [..]
}
int g2(void)
{
  [..]
}
```

Listing 2. module2.c

The run-time state of the machine at a point where it is executing within function $f1()$ when this program is compiled with a standard compiler will look as shown in Figure 3.

Fig. 2. Evaluation of expressions $\mathcal{E}$ in **LLibcheri**

$$\frac{n \in \mathbb{Z}}{n, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow n}\ (\text{evalconst}) \qquad \frac{}{\mathsf{ddc}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow \mathsf{ddc}}\ (\text{evalddc}) \qquad \frac{}{\mathsf{stc}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow \mathsf{stc}}\ (\text{evalstc})$$

$$\frac{}{\mathsf{pcc}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow \mathsf{pcc}}\ (\text{evalpcc})$$

$$\frac{\mathcal{E}_1, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_1 \qquad v_1 \in \mathbb{Z} \qquad \mathcal{E}_2, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v_2 \qquad v_2 \in \mathbb{Z} \qquad v' = v_1 [\oplus] v_2}{\mathcal{E}_1 \oplus \mathcal{E}_2, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'}\ (\text{evalBinOp})$$

$$\frac{\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \qquad v = (\sigma, s, e, \mathit{off}) \in \mathit{Cap} \qquad v' = (\sigma, s, e, \mathit{off} + z)}{\mathtt{inc}(\mathcal{E}, z), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'}\ (\text{evalIncCap})$$

$$\frac{\mathcal{E}, \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v \qquad v = (\sigma, s, e, \mathit{off}) \in \mathit{Cap} \qquad \vdash_\delta v \qquad v' = \mathcal{M}_d(s + \mathit{off})}{\mathtt{deref}(\mathcal{E}), \mathcal{M}_d, \mathsf{ddc}, \mathsf{stc}, \mathsf{pcc} \Downarrow v'}\ (\text{evalDeref})$$

The bounds of the program counter capability (pcc) span the entire code segment, and the offset is pointing to the currently executing instruction. The default data capability (ddc) spans the entire data segment, and the stack capability (stc) spans the entire stack segment, with the offset pointing to the current top of the stack where the top record is an activation record for $f1()$.

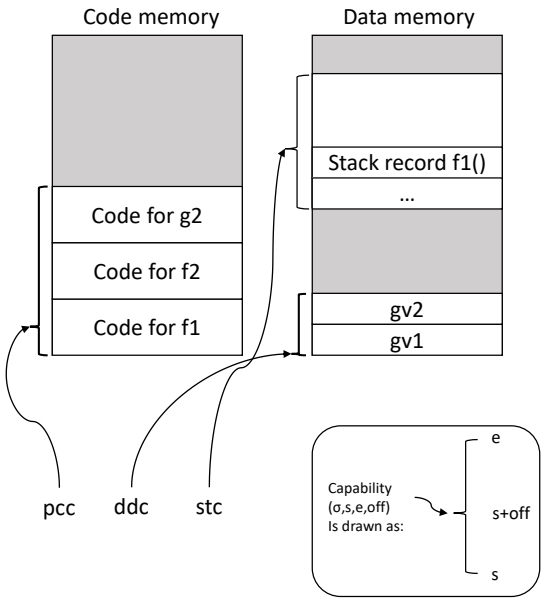

Fig. 3. Machine state with standard compilation

The compiled code could benefit from the memory capability support of the platform to do array bounds checking—for instance if global variable $gv1$ would point to an array, then at run-time the global variable would be represented as a memory capability whose bounds correspond to those of the array, thus providing spatial safety. But the compiled code does *not* make any use of the object capability mechanism provided by the platform. The entire program runs in a single protection domain. The use of memory capabilities can securely isolate this program from other programs that might be running in the same address space, but it does not isolate one part of a program from another part of the same program.

This compilation scheme provides little to no security against an attacker that can provide binary code for one of the translation units (for instance, the attacker can compromise a library that the program links to).

We discuss some examples.

```
1  // Provided by external library
2  extern void untrusted_function(void);
3
4  static int secret = 0xf100f;
5
6  void fun(void)
7  {
8    untrusted_function();
9  }
```

Listing 3. ex1.c

The first example (in Listing 3) is about confidentiality. A global variable qualified as ***static*** is not visible outside its translation unit or module. However, a malicious library providing *untrusted_function()* may peek into the process' data segment to access the value of *secret* when linked against ex1.c.

```
1  // Provided by external library
2  extern void untrusted_function(void);
3
4  static int ton = 1000;
5
6  int ton_mult(int a)
7  {
8    return a *ton;
9  }
10
11 int fun(void)
12 {
13   untrusted_function();
14
15   return ton_mult(10);
16 }
```

Listing 4. ex2.c

The second example shown in Listing 4 deals with integrity. In a similar fashion to the first example, internal variable `ton` should retain its value after calling `untrusted_function()` because C code cannot modify it outside of module/translation unit ex2.c. Much like in the previous example, malicious low-level code in `untrusted_function()` can freely modify the internal variable `ton`.

```
// Provided by external library
extern void untrusted_function(void);

void fun(void)
{
  untrusted_function();

  if (get_level() <= ACCESS_LEVEL)
    printf("Low access level");
  else
    //Critical code
}
```

Listing 5. ex3.c

Finally, the third example (Listing 5) involves control flow. Function `fun` will branch to the critical code only if the return value of `get_level()` is higher than a predetermined value. If compiled in an unsafe manner, a low-level attacker may jump from `untrusted_function()` to the critical part of the program as all executable code is accessible by every instruction, regardless of previous module boundaries in C.

B. Overview of our solution

The key idea of our proposal is that the compiler should map translation units of the source program to target platform *objects*. Consider the program formed by linking the modules in Listings 1 and 2. The run-time state of the machine at a point where it is executing within function $f1()$ when this program is compiled with our proposed compiler will look as shown in Figure 4 on the left. Since execution is now in translation unit 1, the pcc spans only the code of $f1()$, and the dcc spans only the global variable $gv1$. Moreover the stc provides only access to activation record of this invocation of $f1()$. The machine state includes two objects, the green object for translation unit 1 and the red object for translation unit 2. When $f1()$ calls $f2()$, this is compiled to a CInvoke instruction on the red object, leading to a run-time state shown in Figure 4 on the right. With this compilation scheme, each translation unit is running in a separate protection domain, and even if an attacker can provide malicious binary code for one of the translation units, the attacker's power is limited to calling exported functions from other translation units. In particular, all the example attacks discussed in the previous subsection are now prevented, as the attacker cannot directly access global variables of other modules and cannot jump into the middle of a function provided by another module.

IV. The Source Language

We formalize a compiler of a simple **imp**erative language **LImpMod** that features **mod**ules and functions with conditional goto statements. The goal of formalizing this compiler from **LImpMod** to **LLibcheri** is to show that the features of **LLibcheri** can be used to design a fully-abstract (source-to-source) compilation scheme for a C-like imperative language that features protection for module-private state (i.e., for translation-unit-static variables, in C terminology).

A. Program and module representation, and well-formedness

A program in the source language **LImpMod** consists of a list of modules. Each module consists of a list of function definitions, and a list of module-private variables. We skip the syntax of module and function definitions, and we directly represent them as structures (tuples of lists) that are output by the parser. We refer to the set of module identifiers as $ModID$, function identifiers as $FunID$, variable identifiers as $VarID$, and commands as Cmd. We give the syntax for commands and expressions later. We define the set of functions as $FunDef = ModID \times FunID \times \overline{VarID} \times \overline{VarID} \times \overline{Cmd}$ where a function specifies argument names `args`, local variable names `localIDs`, and a body (list of `commands`). Modules $Mod = ModID \times \overline{VarID} \times \overline{FunDef}$ where a module specifies a list of module-private variable names, and a list of function definitions. Programs $Prog = \overline{Mod}$ are lists of modules subject to the following **well-formedness conditions**:

1) Module identifiers are unique across the program, and modules are sorted by lexicographical order of their identifiers.
2) Function identifiers are unique across the program, and function definitions within a module are sorted by lexicographical order of their identifiers.
3) Programs are closed (i.e., the set of all function identifiers existing in a program contains all the function identifiers that are called by any command in the program).
4) The last command of every function is a Return and all jump statements go to destinations inside the same function.

We refer to the operation of linking two lists of modules $\overline{mods_1}$ and $\overline{mods_2}$ into one well-formed program P as $P = \overline{mods_1} \uplus \overline{mods_2}$ where $\uplus$ reorders and concatenates the two lists of modules only if they form a well-formed program P, and is not defined otherwise.

B. Commands and expressions

The syntax of commands is given by the grammar $Cmd ::= \mathsf{Assign}\ \mathcal{E}_l\ \mathcal{E}_r \mid \mathsf{Call}\ FunID\ \overline{\mathcal{E}} \mid \mathsf{Return} \mid \mathsf{Jump}\ \mathcal{E}_c\ n$.

Expressions $\mathcal{E} ::= \mathsf{addr}(VarID) \mid \mathsf{deref}(\mathcal{E}) \mid \mathcal{E} \oplus \mathcal{E} \mid \mathbb{Z} \mid VarID$ in **LImpMod** model a simple notion of C pointers which admits only storing a reference to or, equivalently, the name of a variable but does not support pointer arithmetic. Values $\mathcal{V} = \mathbb{Z} \cup (VarID \times T)$ are integers and pairs of variable identifiers and allocation tokens (allocation tokens are described later). Evaluation of expressions is given by the rules of the form $\mathcal{E}, MVar, VEnv, pc, Fd \Downarrow \mathcal{V}$.

Even though CHERI supports spatial memory safety [14], modeling that is not our goal. Consequently, we do not yet include arrays or pointer arithmetic in **LImpMod**.

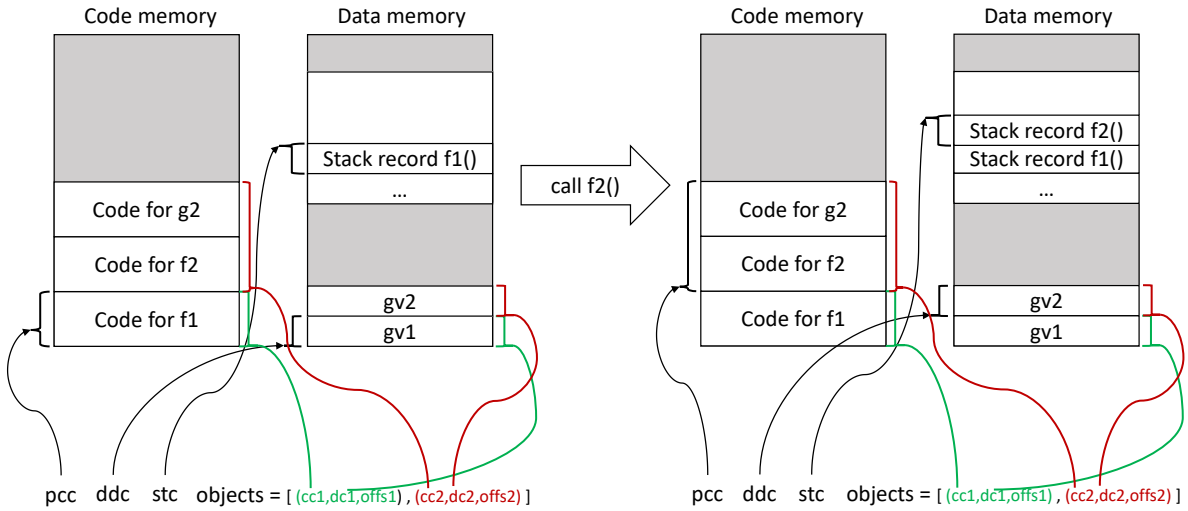

Fig. 4. Machine state with secure compilation

Fig. 5. Evaluation of expressions $\mathcal{E}$ in **LImpMod**

(Evaluate-expr-const)

$$\frac{z \in \mathbb{Z}}{z, MVar, VEnv, pc, Fd \Downarrow z}$$

(Evaluate-expr-binop)

$$\frac{e_1, MVar, VEnv, pc, Fd \Downarrow z_1 \quad z_1 \in \mathbb{Z} \quad e_2, MVar, VEnv, pc, Fd \Downarrow z_2 \quad z_2 \in \mathbb{Z} \quad z_r = z_1 [\oplus] z_2}{e_1 \oplus e_2, MVar, VEnv, pc, Fd \Downarrow z_r}$$

(Evaluate-expr-addr-local)

$$\frac{(fid, \_, t) = pc \quad vid \in \texttt{localIDs}(Fd(fid))}{\mathsf{addr}(vid), MVar, VEnv, pc, Fd \Downarrow (vid, t)}$$

(Evaluate-expr-addr-module)

$$\frac{(fid, \_, t) = pc \quad vid \notin \texttt{localIDs}(Fd(fid)) \quad vid \in MVar(\texttt{moduleID}(Fd(fid)))}{\mathsf{addr}(vid), MVar, VEnv, pc, Fd \Downarrow (vid, 0)}$$

(Evaluate-expr-var)

$$\frac{\mathsf{addr}(id), MVar, VEnv, pc, Fd \Downarrow a \quad VEnv(a) = v}{id, MVar, VEnv, pc, Fd \Downarrow v}$$

(Evaluate-expr-deref)

$$\frac{e, MVar, VEnv, pc, Fd \Downarrow (vid, t') \quad (vid, t') \in VarID \times T \quad VEnv((vid, t')) = v}{\mathsf{deref}(e), MVar, VEnv, pc, Fd \Downarrow v}$$

C. Program state

A program state $\langle MVar, VEnv, stk, Fd, pc, t \rangle$ consists of:

- an immutable map $MVar : ModID \to \overline{VarID}$ of module IDs to module-private variable identifiers,
- an environment $VEnv : (VarID \times T) \to \mathcal{V}$ representing the memory, where T is a set of symbols/tokens that guarantees freshness of allocation across activation records (this is simply a technical alternative to using a stack of activation records, but that additionally captures freshness. Note that all variables in the same activation record have the same token.),
- a call stack $stk : \overline{FunID \times \mathbb{N} \times T}$ which is a list of program counters that record the function calls history (see pc below),
- an immutable map $Fd : FunID \to FunDef$ of function identifiers to function definitions,
- a program counter $pc : FunID \times \mathbb{N} \times T$ keeping track of the current activation record's allocation token, and the index of the next-to-execute command within the list of commands of the current function. We define $\texttt{inc}((funId, n, t)) \stackrel{def}{=} (funId, n + 1, t)$,
- a token $t : T$ that represents the next free allocation token. On every allocation of an activation record, t is incremented. This distinguishes variables in one activation record from those in another.

Note that knowing the currently executing function and the token for its activation record's allocation gives the correct values of the function-local variables from the environment

Fig. 6. Evaluation of commands Cmd in **LImpMod**

(Assign)

$$\frac{(fid, n, \_) = pc \quad \texttt{commands}(Fd(fid))(n) = \mathsf{Assign}\ e_l\ e_r \quad e_l, MVar, VEnv, pc, Fd \Downarrow (vid, t_{e_l}) \quad e_r, MVar, VEnv, pc, Fd \Downarrow v \quad v \in \mathbb{Z} \vee v \in VarID \times \{0\} \quad VEnv' = VEnv[(vid, t_{e_l}) \mapsto v]}{\langle MVar, VEnv, stk, Fd, pc, t\rangle \to \langle MVar, VEnv', stk, Fd, \texttt{inc}(pc), t\rangle}$$

(Call)

$$\frac{(fid, n, \_) = pc \quad \texttt{commands}(Fd(fid))(n) = \mathsf{Call}\ fid_{call}\ \overline{e} \quad argNames = \texttt{args}(Fd(fid_{call})) \quad nArgs = \texttt{length}(argNames) = \texttt{length}(\overline{e}) \quad \overline{e}(i), MVar, VEnv, pc, Fd \Downarrow v_i\ \forall i \in [0, nArgs) \quad v_i \in \mathbb{Z} \vee v_i \in VarID \times \{0\}\ \forall i \in [0, nArgs) \quad stk' = \texttt{push}(stk, pc) \quad pc' = (fid_{call}, 0, t) \quad VEnv' = VEnv[(argNames(i), t) \mapsto v_i\ \forall i \in [0, nArgs)]}{\langle MVar, VEnv, stk, Fd, pc, t\rangle \to \langle MVar, VEnv', stk', Fd, pc', t+1\rangle}$$

(Return)

$$\frac{(fid, n, \_) = pc \quad \texttt{commands}(Fd(fid))(n) = \mathsf{Return} \quad (pc', stk') = \texttt{pop}(stk)}{\langle MVar, VEnv, stk, Fd, pc, t\rangle \to \langle MVar, VEnv, stk', Fd, \texttt{inc}(pc'), t\rangle}$$

(Jump-non-zero)

$$\frac{(fid, n, \_) = pc \quad \texttt{commands}(Fd(fid))(n) = \mathsf{Jump}\ e_c\ n_{dest} \quad e_c, MVar, VEnv, pc, Fd \Downarrow v \quad v \neq 0}{\langle MVar, VEnv, stk, Fd, pc, t\rangle \to \langle MVar, VEnv, stk, Fd, \texttt{inc}(pc), t\rangle}$$

(Jump-zero)

$$\frac{(fid, n, t_{cur}) = pc \quad \texttt{commands}(Fd(fid))(n) = \mathsf{Jump}\ e_c\ n_{dest} \quad e_c, MVar, VEnv, pc, Fd \Downarrow v \quad v = 0}{\langle MVar, VEnv, stk, Fd, pc, t\rangle \to \langle MVar, VEnv, stk, Fd, (fid, n_{dest}, t_{cur}), t\rangle}$$

$VEnv$. All module-private variables are associated with a token value 0 denoting static allocation. The semantics of expressions and commands are given in fig. 5 and fig. 6. The necessary condition $v \in \mathbb{Z} \vee v \in VarID \times \{0\}$ in Assign ensures that assignable values are either integers or addresses of module-private variables, but not local variables of functions.

D. Initial and terminal states

The **initial state** $\langle MVar, VEnv, \texttt{nil}, Fd, (0, 0, 1), 2\rangle$ of a program $p : Prog$, denoted $\texttt{init}(p)$, contains the following components. The static map $MVar$ of module to module-private variables is populated in the obvious way from p. The environment $VEnv$ maps all module-private variable identifiers v to 0 (i.e., $(v, 0) \mapsto 0$). The call stack is empty ($\texttt{nil}$). The function definitions map Fd is populated from p in the obvious way. The program counter points to the 0th command of the 0th function (assuming that `main` will always have the identifier 0), and the allocation token for the activation record of `main` is 1.

A **terminal state** is any state satisfying the judgment $\vdash_t \langle \_, \_, \texttt{nil}, Fd, (fid, n, \_), \_\rangle \stackrel{def}{=}$ $\texttt{commands}(Fd(fid))(n) = \mathsf{Return}$. For two lists of modules $m_1, m_2 : \overline{Mod}$, we use the notation $m_1[m_2] \Downarrow$ to mean that $m_1 \uplus m_2$ is defined (according to the conditions in section IV-A), and $\exists s_t.\ \texttt{init}(m_1 \uplus m_2) \to^* s_t \wedge \vdash_t s_t$, where $\to^*$ is the evalution relation defined in fig. 6. We write $m_1[m_2] \Uparrow$ to mean that $m_1 \uplus m_2$ is defined but $\nexists s_t.\ \texttt{init}(m_1 \uplus m_2) \to^* s_t \wedge \vdash_t s_t$. The judgment $m_1[m_2] \Downarrow$ denotes (proper) convergence of the program $m_1 \uplus m_2$, while $m_1[m_2] \Uparrow$ denotes divergence and "getting stuck".

V. The Compiler

In this section, we give a formal specification of the essential features of our source-to-source compiler, and state the conjectured security properties, whose proofs we leave for future work. The compilation scheme is explained bottom-up, starting from the expression translation all the way up to program translation.

A. Expression and command translation

The expression translation function $\wr\cdot\int_\mu : \mathcal{E} \to \mathcal{E}$ is indexed by a map $\mu : VarID \to \mathcal{E}$ that gives for each variable name in $VarID$ of **LImpMod** the corresponding target expression $\mathcal{E}$ from **LLibcheri** that would evaluate to the address (more precisely, the capability on the address in $\mathcal{M}_d$) in which the variable lives. Construction of μ is explained in section V-B.

For simplicity, we assume from now on that $\mathbb{Z} = \mathbb{Z} = \mathbb{Z}$ and $\mathbb{N} = \mathbb{N} = \mathbb{N}$. Thus, expression translation $\wr\cdot\int_\mu$ is defined as follows:

- $\wr z\int_\mu \stackrel{def}{=} z$ for $z \in \mathbb{Z}$
- $\wr vid\int_\mu \stackrel{def}{=} \texttt{deref}(\mu(vid))$ for $vid \in VarID$
- $\wr e_1 \oplus e_2\int_\mu \stackrel{def}{=} \wr e_1\int_\mu \oplus \wr e_2\int_\mu$
- $\wr \texttt{deref}(e)\int_\mu \stackrel{def}{=} \texttt{deref}(\wr e\int_\mu)$
- $\wr \texttt{addr}(vid)\int_\mu \stackrel{def}{=} \mu(vid)$ for $vid \in VarID$

We also define expression translation for a list of expressions as $\wr\overline{e}\int_\mu \stackrel{def}{=} \wr e_0\int_\mu ... \wr e_{n-1}\int_\mu$ where $\overline{e} \equiv e_0 ... e_{n-1}$.

The command translation function $(\!|\cdot|\!)_{\mu,i,\rho} : Cmd \to Cmd$ is indexed by the map $\mu : VarID \to \mathcal{E}$ described above, an index $i \in \mathbb{N}$ of the command being translated within the function body $\overline{Cmd}$ in which the command appears, and a requirements map $\rho : FunID \to \mathbb{N}^2$, which for each function identifier gives the corresponding module and function identifiers of the compiled program.

The command translation $(\!|\cdot|\!)_{\mu,i,\rho}$ is thus defined as follows:

- $(\!|\mathsf{Assign}\ e_l\ e_r|\!)_{\mu,i,\rho} \stackrel{def}{=} \mathsf{Assign}\ \wr e_l \int_\mu\ \wr e_r \int_\mu$
- $(\!|\mathsf{Return}|\!)_{\mu,i,\rho} \stackrel{def}{=} \mathsf{Return}$
- $(\!|\mathsf{Jump}\ e\ n|\!)_{\mu,i,\rho} \stackrel{def}{=} \mathsf{Jump}\ \wr e \int_\mu\ \mathtt{inc(pcc, j)}$ where $\mathtt{j} = \wr n \int_\mu - i$
- $(\!|\mathsf{Call}\ fid\ \overline{e}|\!)_{\mu,i,\rho} \stackrel{def}{=} \mathsf{Cinvoke}\ \rho(fid).1\ \rho(fid).2\ \wr \overline{e} \int_\mu$

We note that this simplicity of the expression and command translation is the consequence of deliberately designing the expressions and commands of **LImpMod** and **LLibcheri** to be similar. This allows us to focus on security issues, rather than on translating expressions and commands.

B. Function and module translation

The function translation algorithm $\llbracket\cdot\rrbracket_{\mu,\rho,i_s} : \mathit{FunDef} \to \mathit{CodeMemory}$ takes a parsed function and returns a code memory in which the translation of the function is given in successive addresses starting at the address i_s. (The construction of map μ is explained in rules Module-translation and Function-translation and ρ is as explained in section V-A.) The function translation algorithm is specified by the inference rule Function-translation in fig. 7. We assume for simplicity that the number of arguments to each function is fixed to nArgs which is known to the Cinvoke command semantics.

Module translation $\lceil\!\lceil\cdot\rceil\!\rceil_{\rho,i_c,i_d} : \mathit{Mod} \to (\mathit{CodeMemory} \times \mathit{DataMemory} \times \mathit{ObjCap})$ produces the translated module's code and data memories (in successive addresses starting at i_c and i_d respectively), along with the object capability protecting them. Module translation is specified by the inference rule Module-translation in fig. 7. We note that modules are assumed to adhere to the well-formedness conditions in section IV-A that are applicable to individual modules (e.g., the list of functions fundef is sorted alphabetically by the function identifiers, which is a step that can be performed by a compiler pass or the parser [5], [6].

C. A compiler from *LImpMod* to *LLibcheri*

Our compilation scheme $[\![\cdot]\!]_\rho : \overline{\mathit{Mod}} \to \mathit{TargetSetup}$ translates a list of **LImpMod** modules into a **LLibcheri** setup in a way that ensures compartmentalization with respect to the source modules. The inference rule Module-list-translation completes the definition of our compilation scheme. The requirements map $\rho : \mathit{FunID} \to \mathbb{N}^2$ is assumed to be input to the compiler for the open parts of the program. (This models the mapping of headers of functions to symbols that are resolved at load-time. Here, the symbols are the module and function identifier pairs.) For the available function definitions in a module with identifier mid, function identifiers FunID are mapped to $\mathbb{N}$ sequentially starting from 0 in the order of appearance of the function definitions in the source module that has been mapped to the identifier mid. The map $\varrho : \mathit{ModID} \to \mathbb{N}$ gives the target module identifiers.

D. Security Properties

To be convinced about the securty of the compiler, we need:

1) properties for compilers that capture security,
2) a statement that our compiler $[\![\cdot]\!]_{\rho,\varrho}$ has those properties,
3) and a proof of said statement.

In this section we provide the first two and only an informal proof that the proof of the statement holds; a complete formal proof is left for future work.

To express compiler security, one de-facto standard exists: compiler full abstraction [18]. Informally, a compiler is fully abstract if the compilation from source programs to target programs preserves and reflects behavioural equivalence. In other words, a compiler is fully-abstract if for any two source programs $\overline{m}_1$ and $\overline{m}_2$, we have that they are behaviourally equivalent ($\overline{m}_1 \simeq_{ctx} \overline{m}_2$) if and only if their compiled counterparts are behaviourally equivalent ($[\![\overline{m}_1]\!] \simeq_{ctx} [\![\overline{m}_2]\!]$). The notion of behavioural equivalence used here is the canonical notion of contextual equivalence: two terms are equivalent if they behave the same when plugged into any valid context.

In this setting, a source context $\mathbb{C}$ for an open program $\overline{m}$ is a list of modules $\overline{c}$ such that $\overline{c} \uplus \overline{m}$ is defined. A target context $\mathbb{C} : \mathit{TargetSetup}$ for a compiled program $p : \mathit{TargetSetup}$ is one for which $\mathbb{C} \uplus p$ is defined.

Source and target contextual equivalence can be stated as follows (we use black to avoid repeating the definition in both colours), where $\Uparrow$ means divergence:

$$\overline{m} \simeq_{ctx} \overline{m}' \stackrel{\text{def}}{=} \forall \mathbb{C}.\ \mathbb{C}[\overline{m}] \Uparrow \iff \mathbb{C}[\overline{m}'] \Uparrow$$

This definition is standard and used by most papers in the literature on secure compilation [3]–[7], [19]–[22].

Compiler full abstraction can be stated as follows:

$$\forall \overline{m}_1, \overline{m}_2.\ \overline{m}_1 \simeq_{ctx} \overline{m}_2 \iff [\![\overline{m}_1]\!] \simeq_{ctx} [\![\overline{m}_2]\!]$$

We denote a compiler $[\![\cdot]\!]$ being fully-abstract as $[\![\cdot]\!] \in \mathit{FA}$.

Another crucial property that compilers must have is modularity. A compiler is modular when it operates on components and compiled modules can be linked together into larger components (and possibly into whole programs). Supporting modular compilation and linking of modules is a de-facto requirement of modern compilers, as it is easier to write and compile code in separate components.

Modularity is formalised as follows:

$$\forall \overline{m}_1, \overline{m}_2.\ [\![\overline{m}_1 \uplus \overline{m}_2]\!]_{\rho,\varrho} \simeq_{ctx} [\![\overline{m}_1]\!]_{\rho,\varrho} \uplus [\![\overline{m}_2]\!]_{\rho,\varrho}$$

We denote a compiler $[\![\cdot]\!]$ being modular as $[\![\cdot]\!] \in \mathit{MO}$.

The combination of full abstraction and modularity yields *modular full abstraction* [7]. Formally, a compiler is modularly fully abstract if:

$$\begin{aligned}&\forall \overline{m}_1, \overline{m}_2, \overline{m}_3, \overline{m}_4.\\ &\quad \overline{m}_1 \uplus \overline{m}_2 \simeq_{ctx} \overline{m}_3 \uplus \overline{m}_4 \iff\\ &\quad [\![\overline{m}_1]\!]_{\rho,\varrho} \uplus [\![\overline{m}_2]\!]_{\rho,\varrho} \simeq_{ctx} [\![\overline{m}_3]\!]_{\rho,\varrho} \uplus [\![\overline{m}_4]\!]_{\rho,\varrho}\end{aligned}$$

We denote a compiler $[\![\cdot]\!]$ being modular as $[\![\cdot]\!] \in \mathit{MFA}$.

Finally, a compiler should be *functionally* correct, i.e., it should preserve the meaning of the program. Here, the compilation of expressions and commands is straightforward, so we expect that proving functional correctness will be

Fig. 7. Compilation of functions, modules and module lists

$$\frac{\begin{array}{c}\text{(Function-translation)}\\ |\overline{args}| = nArgs \quad |\overline{localvars}| = \phi \quad \mu_a = \biguplus_{i\in[0,nArgs)} \overline{args}(i) \mapsto \texttt{inc(stc,i)} \\ \mu_l = \biguplus_{i\in[0,|\overline{localvars}|)} \overline{localvars}(i) \mapsto \texttt{inc(stc,i} + nArgs\texttt{)} \\ \mu' = \mu \uplus \mu_a \uplus \mu_l \qquad \mathcal{M}_c = \biguplus_{i\in[0,|\overline{cmd}|)} i_s + i \mapsto (\!|\overline{cmd}(i)|\!)_{\mu',i,\rho}\end{array}}{[\![(\_,\_,\overline{args},\ \overline{localvars},\ \overline{cmd})]\!]_{\ \mu,\rho,i_s} = \mathcal{M}_c}$$

$$\frac{\begin{array}{c}\text{(Module-translation)}\\ \mathcal{M}_d = \biguplus_{i\in[0,|\overline{privvars}|)} i_d + i \mapsto 0 \\ \mu = \biguplus_{i\in[0,|\overline{privvars}|)} \overline{privvars}(i) \mapsto \texttt{inc(ddc,i)} \\ (\mathcal{M}_c, offs) = (\biguplus_{j\in[0,|\overline{fundef}|)} \mathcal{M}_{c_j}, \biguplus_{j\in[0,|\overline{fundef}|)} j \mapsto i_j) \ with\ \mathcal{M}_{c_j} = [\![\overline{fundef}(j)]\!]_{\mu,\rho,i_j} \ with\ i_j = i_c + \sum_{k\in[0,j)} |\texttt{dom}(\mathcal{M}_{c_k})| \\ objcap = ((\kappa, i_c, i_c + |\texttt{dom}(\mathcal{M}_c)|, 0), (\delta, i_d, i_d + |\texttt{dom}(\mathcal{M}_d)|, 0), offs)\end{array}}{[\![(\_,\overline{privvars},\overline{fundef})]\!]_{\rho,i_c,i_d} = (\mathcal{M}_c, \mathcal{M}_d, objcap)}$$

$$\frac{\begin{array}{c}\text{(Module-list-translation)}\\ (\mathcal{M}_c, \mathcal{M}_d, imp) = \biguplus_{j\in[0,|\overline{m}|)} (\mathcal{M}_{cj}, \mathcal{M}_{dj}, imp_j) \\ with\ m_j = [\![\overline{m}(j)]\!]_{\rho,i_{cj},i_{dj}}, \mathcal{M}_{cj} = m_j.1, \mathcal{M}_{dj} = m_j.2, imp_j = \varrho(\texttt{modID}(\overline{m}(j))) \mapsto m_j.3, \\ with\ i_{cj} = \sum_{k\in[0,j)} |\texttt{dom}(\mathcal{M}_{c_k})|, i_{dj} = \sum_{k\in[0,j)} |\texttt{dom}(\mathcal{M}_{d_k})|\end{array}}{[\![\overline{m}]\!]_{\rho,\varrho} = (\mathcal{M}_c, \mathcal{M}_d, imp)}$$

straightforward. Hence, we do not consider this requirement any further.

We believe that our compiler has the following three properties:

Theorem 1 ($[\![\cdot]\!]$ is fully abstract)**.** $[\![\cdot]\!] \in FA$

Theorem 2 ($[\![\cdot]\!]$ is modular)**.** $[\![\cdot]\!] \in MO$

Theorem 3 ($[\![\cdot]\!]$ is modularly fully abstract)**.** $[\![\cdot]\!] \in MFA$

The hard theorem is Theorem 1. In fact, formal techniques for proving compiler full abstraction are an active topic of research [22], [23]. We leave a proof of this theorem to future work.

VI. Implementation

In this section we introduce our implementation of the compilation scheme that was formalized previously.

The compiler is built upon CheriBSD, a port of FreeBSD for the CHERI processor, and CHERI's Clang/LLVM compiler [2]. In order to describe the details of the source-to-source compiler, it is necessary to briefly explain CheriBSD's programmer-friendly interface to CHERI's compartmentalization features, namely libcheri [24]. Under libcheri, the isolated, compartmentalized parts of a program are called "sandboxes" and libcheri is the API for loading sandboxes, setting them up and invoking them. The interface consists of a number of functions and macro definitions and is complemented by new compiler attributes and linker scripts.

The main primitives of CHERI's in-process compartmentalization are *classes* and *objects* [24]. Classes represent sandboxes, manifest through statically linked executable images, while objects represent corresponding invokable object capabilities. It is the responsibility of the programmer to group functions into CHERI classes and to create the invokable object capabilities for each class. The programmer may annotate functions as *cheri_ccall* so that conventional function calls are replaced by object capability invocations. Functions that are meant to be exported by the current sandbox should be annotated with the *cheri_ccallee* attribute. At runtime, libcheri acts as a loader that reads the executable images from the filesystem and creates the respective object capabilities. It is critical that when the sandbox loading routines are called the program is in its initial state where it has control over its entire address space as well as file system access.

Our compilation scheme maps each C module to a separate sandbox. This translates to assigning a CHERI class and creating a CHERI object for each module. Our source-to-source compiler achieves this by first performing semantic analysis of all program modules and assembling a mapping of function identifiers to C modules, which helps resolve dependencies in the next step. The compiler then traverses through each module's AST and annotates every external function *declaration* it encounters as either *cheri_ccallee* or *cheri_ccall*, depending on whether the function is defined in the current translation unit or not. In the second case, the source-to-source compiler uses the dependency map to

specify the sandbox each external function belongs to. The compiler will also add libcheri object declarations required by the annotations. As a performance optimization over the formal model, intra-module function calls do not translate to object capability invocations. Instead, they are ordinary MIPS function calls. This change has no security implications.

One hurdle in using libcheri for our compilation scheme is initialization. Each sandbox needs the respective object capabilities to be able to invoke functions exported by other sandboxes. The question is who creates these capabilities and where they are stored. Loading of a new sandbox requires system calls, and CheriBSD prohibits system calls in compartmentalized code. While it is possible to allow sandboxes to invoke system calls by passing the special "system" capability, doing that would violate the principle of least privilege.

Our solution is to extend libcheri with a new load-initialization function `sandbox_chain_load()`. This function is meant to be called only once by an initialization module, which is the only privileged part of the program (and hence can do system calls). `sandbox_chain_load()` loads the "main" sandbox from the filesystem and also any modules that "main" depends on (recursively). It also creates relevant object capabilities for every sandbox and places them at the beginning of the sandbox's data segment. As a result, every sandbox has access to the object capabilities necessary to invoke exported functions from other sandboxes. Extending libcheri required considerable additions to the libcheri code base including the definition of `sandbox_chain_load()`, new versions of sandbox creation routines that support sandbox dependencies and low-level macros that expose relevant sandbox metadata to C.

For reasons of technical convenience, our formal model of Cinvoke differs from its implementation in libcheri. In libcheri, Cinvoke is implemented as a combination of libcheri's `libcheri_cinvoke()` function and CHERI's `ccall` instruction. In our formal model, we push ddc, stc and pcc onto the trusted stack before every call. On the other hand, CHERI/CheriBSD pushes registers `$pcc` and `$idc`, where `$idc` is a capability pointing to a memory region that itself contains four capabilities: the stack pointer `$sp`, the data capability `$ddc`, the stack capability `$stc` and the original `$idc` from the previous ccall instruction [2]. The "bundling" and "unbundling" of `$idc` takes place outside of `ccall` and `creturn` with the help of libcheri.

The compilation procedure is illustrated in Figure 8, Listings 6 to 12. The source program has three C modules: an entry point, main.c, along with libraries lib1.c and lib2.c that implement the functions `f1()` and `f2()`, respectively. The compiler adds a constructor function to each module, whose sole responsibility is to fetch the object capabilities from the module's data segment. Execution begins at the `init()` function of the module init.c, where `sandbox_chain_load()` is called before invoking the actual entry point of the prorgam, `main()`. Since this function is annotated with the `cheri_ccall` attribute, the respective object capability is invoked so that module main.c is executed in its own sandbox as intended.

Observe that the bodies of all function definitions (across all modules) are equal before and after compilation. Hence, our source-to-source compiler only inserts attributes to function declarations; it does not change the functions themselves.

Fig. 8. Function definition mapping after first pass

main → main
f1 → lib1
f2 → lib2

```c
int f1(void);
int f2(void);

int main(void)
{
  f1();
  f2();

  return 0;
}
```

Listing 6. Input to the source-to-source compiler. main.c

```c
int f1(void);
int f2(void);

int f1(void)
{
  f2();
}
```

Listing 7. Input to the source-to-source compiler. lib1.c

```c
int f2(void);

int f2(void)
{
[..]
}
```

Listing 8. Input to the source-to-source compiler. lib2.c

```c
struct cheri_object main_obj;
static struct sandbox_object *main_objectp;

__attribute__((cheri_ccall))
__attribute__((cheri_method_suffix("_cap")))
__attribute__((cheri_method_class(main_obj)))
extern int main(int argc, char *argv[]);

int init(int argc, char *argv[])
{
  sandbox_chain_load("main", &main_objectp);
  main_obj = sandbox_object_getobject(main_objectp);

  main(argc, argv);
}
```

Listing 9. Source-to-source compilation output. Initialization module init.c

```c
extern struct cheri_object main_obj;
struct cheri_object lib1;
struct cheri_object lib2;

__attribute__((cheri_ccall))
__attribute__((cheri_method_suffix("_cap")))
__attribute__((cheri_method_class(lib1)))
int f1(void);

__attribute__((cheri_ccall))
__attribute__((cheri_method_suffix("_cap")))
__attribute__((cheri_method_class(lib2)))
int f2(void);

__attribute__((cheri_ccallee))
__attribute__((cheri_method_class(main_obj)))
int main(void);
```

```
__attribute__ ((constructor)) static void
sandboxes_init(void)
{
  struct sandbox_metadata *mdata;
  struct external_dep *s_lib1_dep,
       *s_lib2_dep;

  mdata = GET_METADATA();

  lib2_dep = sandbox_dep_lookup(mdata→sbm_deps,
          mdata→sbm_depnum, "lib2");
  lib2 = lib2_dep→lib;

  lib1_dep = sandbox_dep_lookup(mdata→sbm_deps,
          mdata→sbm_depnum, "lib1");
  lib1 = lib1_dep→lib;
}

int main(void)
{
  f1();
  f2();

  return 0;
}
```

Listing 10. Source-to-source compilation output. Transformed main.c

```

extern struct cheri_object lib1;
struct cheri_object lib2;

__attribute__((cheri_ccallee))
__attribute__((cheri_method_class(lib1)))
int f1(void);

__attribute__((cheri_ccall))
__attribute__((cheri_method_suffix("_cap")))
__attribute__((cheri_method_class(lib2)))
int f2(void);

__attribute__ ((constructor)) static void
sandboxes_init(void)
{
  struct sandbox_metadata *mdata;
  struct external_dep *s_lib2_dep,

  mdata = GET_METADATA();

  lib2_dep = sandbox_dep_lookup(mdata→sbm_deps,
          mdata→sbm_depnum, "lib2");
  lib2 = lib2_dep→lib;
}

int f1(void)
{
  f2();
}
```

Listing 11. Source-to-source compilation output. Transformed lib1.c

```
extern struct cheri_object lib2;

__attribute__((cheri_ccallee))
__attribute__((cheri_method_class(lib2)))
int f2(void);

int f2(void)
{
[..]
}
```

Listing 12. Source-to-source compilation output. Transformed lib2.c

VII. Related and Future Work

Capabilities are an old notion [10], [25]–[27]. They have been used to add protection to operating systems [10]–[12], [28], [29], programming languages [30], [31] and security architectures [13], [32], [33]. Concerning the latter, few theoretical and practical models exist: the M-machine [32], [34], Capsicum [13] and CHERI [1], [2], [14], [33]. CHERI is not only the most mature capability machine implementation, but it has also recently been formalised by El-Korashy [16]. El-Korashy also proves that a number of security properties such as capability unforgeability, compartmentalisation and control-flow integrity can be realized through careful use of CHERI's ISA. These properties are useful building blocks for the security proof of a compiler. We expect that a formal proof of Theorem 1 will rely on these properties.

Secure compilation has been achieved for different security architectures: SGX-like enclaves [5]–[7], [35], metadata tracking architectures (i.e., the Pump machine) [9], [21] and ASLR [3], [4]. A trivial use of CHERI for secure compilation would be to use capabilities to mimic enclave-like structures and reuse work on securely compiling to enclaves. However, the finer protection granularity in CHERI, as well as its compartmentalisation primitives suggest that compiling directly to CHERI can be more efficient than compiling to enclaves. The Pump machine is an instance of an architecture that allows for efficient secure compilation. However, its extensive hardware-supported metadata tracking seems to be unnecessary for most security applications. As that metadata tracking can cause cache misses and thus performance reduction, we believe CHERI might have better performance. Finally, ASLR only achieves probabilistic security guarantees. CHERI can provide absolute guarantees.

The de-facto formal standard for secure compilation is full abstraction [3]–[7], [18], [19], [22]. Recently, shortcomings of FA have been pointed out and new alternatives have been proposed. For example, modular full abstraction [7] forces a compiler to be both modular and fully abstract, preventing the development of secure compilers for which full abstraction fails under composition. Secure compartmentalised compilation also enforces modularity [21], but it also supports source languages with undefined behaviour. To do so, it requires compiled components to be fully defined, i.e., if undefined behaviour arises, then it does not have effects outside the current component's boundaries. Instead of going for SCC in our full development, we eliminate all C undefined behaviours in the semantics by converting it to an error. While this won't let compilers perform undefined behaviour-based optimisation, it does reduce the surface for attacks and mistakes. Finally, trace preserving compilation or TPC [36] can be seen as a form of full abstraction where all failures and checks are treated uniformly. The main difference with full abstraction is that TPC has been proven to preserve arbitrary safety hyperproperties.

All compiler security properties mentioned above are reminiscent or based directly on full abstraction so we plan to stay with full abstraction in our future development. We will likely consider modularity (which we believe to be fundamental) and treat all failures uniformly (like TPC) since that is both easy and proven to preserve safety hyperproperties. For proving compiler full abstraction, two main approaches exist: equip-

ping the target language with a logical relation [20], [22], [23], [37], or with a labelled transition system that yields a notion of bisimilarity or trace equivalence [4]–[7], [21], [38]. So far, we have not committed to this choice. However, we expect to commit to this choice soon in order to make progress on our proofs.

Finally, another important avenue for future work is to support more features of the C language. Handling dynamic allocation (malloc()), arrays and structs seems reasonably straightforward. They are already supported in our implementation, but not yet included in the formal model. More challenging is handling function pointers, pointers to local variables and, more generally, enforcing temporal memory safety. We plan to address these challenges using CHERI's support for *local capabilities*.

VIII. Conclusion

Capabilities are a powerful fine-grained low-level protection mechanism. Compilers can use this powerful mechanism to enforce properties of the source language at run-time. In this paper we have reported on our work-in-progress of building a compiler that uses the target platform's support for object capabilities to automatically compartmentalize the programs it compiles. Specifically, our compiler creates a separate protection domain for each C translation unit, thus providing protection against malicious libraries that the program links with.

References

[1] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI Capability Model: Revisiting RISC in an Age of Risk," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 457–468, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2678373.2665740

[2] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 20–37.

[3] M. Abadi and G. Plotkin, "On protection by layout randomization," in *CSF '10*. IEEE, 2010, pp. 337–351. [Online]. Available: http://dx.doi.org/10.1109/CSF.2010.30

[4] R. Jagadeesan, C. Pitcher, J. Rathke, and J. Riely, "Local memory via layout randomization," in *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, ser. CSF '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 161–174. [Online]. Available: http://dx.doi.org/10.1109/CSF.2011.18

[5] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens, "Secure compilation to protected module architectures," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 2, p. 6, 2015.

[6] P. Agten, R. Strackx, B. Jacobs, and F. Piessens, "Secure compilation to modern processors," in *CSF '12*. IEEE, 2012, pp. 171 – 185. [Online]. Available: http://dx.doi.org/10.1109/CSF.2012.12

[7] M. Patrignani, D. Devriese, and F. Piessens, "On Modular and Fully-Abstract Compilation," in *Proceedings of the 29th IEEE Computer Security Foundations Symposium CSF 2016, Lisbon, Portugal*, ser. CSF 2016, 2016.

[8] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 487–502, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786763.2694383

[9] Y. Juglaret and C. Hritcu, "Secure compilation using micro-policies," 2015.

[10] H. M. Levy, *Capability-Based Computer Systems*. Newton, MA, USA: Butterworth-Heinemann, 1984.

[11] R. S. Fabry, "Capability-based addressing," *Commun. ACM*, vol. 17, no. 7, pp. 403–412, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361070

[12] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," in *ACM SIGPLAN Notices*, vol. 29, no. 11. ACM, 1994, pp. 319–327.

[13] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for unix."

[14] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, R. Norton, S. Son, and H. Xia, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 5)," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-891, Jun. 2016. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-891.pdf

[15] J. Heinrich, *MIPS R4000 Microprocessor User's manual*, 1994.

[16] A. El-Korashy, "A Formal Model for Capability Machines: An Illustrative Case Study towards Secure Compilation to CHERI," Max-Planck Institute for Software Systems, Saarbrücken, Tech. Rep., 2016. [Online]. Available: https://people.mpi-sws.org/~elkorashy/

[17] "Rigorous Engineering of Mainstream Systems," 2016, [Online; accessed 06-September-2016]. [Online]. Available: https://www.cl.cam.ac.uk/~pes20/rems/

[18] M. Abadi, "Protection in programming-language translations," in *International Colloquium on Automata, Languages, and Programming*. Springer, 1998, pp. 868–883.

[19] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, "Fully abstract compilation to javascript," *SIGPLAN Not.*, vol. 48, no. 1, pp. 371–384, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2480359.2429114

[20] A. Ahmed and M. Blume, "Typed closure conversion preserves observational equivalence," *SIGPLAN Not.*, vol. 43, no. 9, pp. 157–168, Sep. 2008. [Online]. Available: http://doi.acm.org/10.1145/1411203.1411227

[21] Y. Juglaret, C. Hriţcu, A. Azevedo de Amorim, and B. C. Pierce, "Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation," in *29th IEEE Symposium on Computer Security Foundations (CSF)*. IEEE Computer Society Press, Jul. 2016. [Online]. Available: http://arxiv.org/abs/1602.04503

[22] D. Devriese, M. Patrignani, and F. 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, St. Petersburg, FL, USA, January 20 - 22, 2016*, 2016, pp. 164–177. [Online]. Available: http://doi.acm.org/10.1145/2837614.2837618

[23] M. S. New, W. J. Bowman, and A. Ahmed, "Fully abstract compilation via universal embedding," in *International Conference on Functional Programming*. ACM, 2016, pp. 103–116.

[24] R. M. Norton, D. Chisnall, B. Davis, W. Koszek, S. W. Moore, S. J. Murdoch, P. G. Neumann, and J. Woodruff, "Capability Hardware Enhanced RISC Instructions: CHERI Programmer's Guide," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-877, Sep. 2015. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-877.pdf

[25] J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[26] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[27] D. Devriese, L. Birkedal, and F. Piessens, "Reasoning about object capabilities with logical relations and effect parametricity," in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, 2016, pp. 147–162. [Online]. Available: http://dx.doi.org/10.1109/EuroSP.2016.22

[28] J. S. Shapiro, J. M. Smith, and D. J. Farber, "Eros: A fast capability system," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 170–185, Dec. 1999. [Online]. Available: http://doi.acm.org/10.1145/319344.319163

[29] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "Hydra: The kernel of a multiprocessor operating system," *Commun. ACM*, vol. 17, no. 6, pp. 337–345, Jun. 1974. [Online]. Available: http://doi.acm.org/10.1145/355616.364017

[30] A. Mettler, D. Wagner, and T. Close, "Joe-e: A security-oriented subset of java." in *NDSS*. The Internet Society, 2010. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2010.html#MettlerWC10

[31] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Caja - Safe active content in sanitized JavaScript," http://code.google.com/p/google-caja/downloads/detail?name=caja-spec-2008-06-07.pdf, Google Inc., Tech. Rep., Jun. 2008.

[32] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee, "The m-machine multicomputer," *International Journal of Parallel Programming*, vol. 25, no. 3, pp. 183–212, 1997.

[33] J. D. Woodruff, "CHERI: A RISC capability machine for practical memory safety," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-858, Jul. 2014. [Online]. Available: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-858.pdf

[34] N. P. Carter, S. W. Keckler, and W. J. Dally, "Hardware support for fast capability-based addressing," *SIGPLAN Not.*, vol. 29, no. 11, pp. 319–327, 1994. [Online]. Available: http://doi.acm.org/10.1145/195470.195579

[35] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *22nd USENIX Security symposium*. USENIX Association, August 2013, pp. 479–494. [Online]. Available: https://lirias.kuleuven.be/handle/123456789/402673

[36] M. Patrignani and D. Garg, "Secure Compilation and Hyperproperties Preservation," in *Proceedings of the 30th IEEE Computer Security Foundations Symposium CSF 2017, Santa Barbara, USA*, ser. CSF 2017, 2017.

[37] A. Ahmed and M. Blume, "An equivalence-preserving CPS translation via multi-language semantics," *SIGPLAN Not.*, vol. 46, no. 9, pp. 431–444, Sep. 2011. [Online]. Available: http://doi.acm.org/10.1145/2034574.2034830

[38] M. Patrignani and D. Clarke, "Fully abstract trace semantics for protected module architectures," *Computer Languages, Systems & Structures*, vol. 42, pp. 22–45, 2015.