Towards Formally Verified Compilation of Tag-Based Policy Enforcement

CHR Chhak
Portland State University

Andrew Tolmach
Portland State University

Sean Anderson
Portland State University

Citation Details

Chhak, C. H. R., Tolmach, A., & Anderson, S. (2021, January). Towards formally verified compilation of tag-based policy enforcement. In Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (pp. 137-151).

Towards Formally Verified Compilation of Tag-Based Policy Enforcement

CHR Chhak
Andrew Tolmach
Sean Anderson
Portland State University
Portland, OR, USA
{chr.chhak,tolmach,ander28}@pdx.edu

Abstract

Hardware-assisted reference monitoring is receiving increasing attention as a way to improve the security of existing software. One example is the PIPE architecture extension, which attaches metadata tags to register and memory values and executes tag-based rules at each machine instruction to enforce a software-defined security policy. To use PIPE effectively, engineers should be able to write security policies in terms of source-level concepts like functions, local variables, and structured control operators, which are not visible at machine level. It is the job of the compiler to generate PIPE-aware machine code that enforces these source-level policies. The compiler thus becomes part of the monitored system's trusted computing base—and hence a prime candidate for verification.

To formalize compiler correctness in this setting, we extend the source language semantics with its own form of user-specified tag-based monitoring, and show that the compiler preserves that monitoring behavior. The challenges of compilation include mapping source-level monitoring policies to instruction-level tag rules, preserving fail-stop behaviors, and satisfying the surprisingly complex preconditions for conventional optimizations. In this paper, we describe the design and verification of Tagine, a small prototype compiler that translates a simple tagged WHILE language to a tagged register transfer language and performs simple optimizations. Tagine is based on the RTLgen and Deadcode phases of the CompCert compiler, and hence is written and verified in Coq. This work is a first step toward verification of a full-scale compiler for a realistic tagged source language.

ACM ISBN 978-1-4503-8299-1/21/01…$15.00
https://doi.org/10.1145/3437992.3439929

CCS Concepts: • **Security and privacy** → **Logic and verification**; **Hardware security implementation**; **Software security engineering**.

Keywords: verified compilers, reference monitors, tag-based secure hardware, Coq proof assistant

ACM Reference Format:
CHR Chhak, Andrew Tolmach, and Sean Anderson. 2021. Towards Formally Verified Compilation of Tag-Based Policy Enforcement. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '21), January 18–19, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3437992.3439929

1 Introduction

Reference monitors [2, 21] are a powerful mechanism for dynamic enforcement of software security policies such as access control, memory safety [18], and information-flow control (IFC). Monitors interpose a validation test at each security-relevant program point and cause the program to fail-stop in the event of a security violation. They are used in settings where the underlying software cannot easily be modified or perhaps even inspected. This makes them an important tool of the *security engineer*—somebody tasked with improving system security, often not the original programmer. However, monitoring is expensive to implement in software, even when applied only at coarse granularity, e.g. only at function calls.

Recent work has shown that *hardware-assisted* monitoring approaches can enforce fine-grained security policies while still providing good performance. For example, PIPE [7, 19] (Processor Interlocks for Policy Enforcement)[1] is a programmable hardware mechanism for supporting reference monitors at the granularity of individual instructions. In a processor architecture extended with PIPE, metadata tags are associated with each value in memory and registers. Just before each instruction executes, PIPE checks its opcode and the tags on its operands to see if the operation should be permitted, and if so, what tags should be assigned to the instruction's results. These *tag rules* collectively form

[1]In previous and contemporaneous work, variants of PIPE are also called PUMP, SDMP, or CoreGuard.

a *micro-policy* [4] (hereinafter simply *policy*). Tag rules are implemented in software (running in a privileged supervisor context or on a dedicated co-processor), so policies are completely flexible in how they interpret tags and gate machine operations. Adding software checks at per-instruction granularity would be far too expensive, so the results of tag rules are stored in a fast hardware cache. In a well-designed policy, the cache hit rate will be high, so most instructions will execute at full speed. Experiments have shown reasonable performance on a range of useful policies [7, 8, 19].

However, the PIPE approach does have some limitations. Defining tag rules at the level of the machine ISA is a difficult task for the security engineer, much as writing machine code is harder than working in a high-level language. In principle, working at the instruction level minimizes the trusted computing base (TCB) of the monitoring system; in particular, security properties are enforced independently of how the machine code was produced. In practice, however, writing useful policies often requires understanding the output of a particular compiler. For example, a policy intended to guarantee integrity of the stack [19] must have at least partial knowledge of how the compiler lays out stack frames and which generated instructions are performing stack manipulation. This kind of reverse engineering is both tedious and error-prone.

More fundamentally, some policies can only be expressed in terms of high-level code features that are not preserved at machine level. For example, an access control policy might wish to gate entry to a function by inspecting the tags on its arguments, but it may not be clear at machine level where those arguments live. A memory safety policy may want to distinguish accesses to local variables from accesses to the heap, even though both are compiled into the same machine-level load and store instructions. Or an IFC policy may want to delimit the scope of implicit flows [6] based on knowledge of the structured control flow (e.g. `if-then-else` constructs) in the source program, which is not explicitly visible in machine code.

We therefore propose defining policies in a high-level source language, compiling to PIPE-compatible code, and including the compiler within the TCB. We extend a high-level language with a tag-based reference monitoring semantics, and implement this extended language by compilation to machine code for a PIPE-equipped processor. In the source language, tag rules are triggered at meaningful *control points* in the dynamic semantics, such as evaluation of arithmetic operators, reading or writing variables, function entry and return, and split and join points in the control-flow graph. We use hardware-level tags on the generated instructions to trace their provenance back to the source-level construct (and associated control point and tag rule) that produced them.

Since the compiler is now in the TCB, it is essential that it correctly implements the intended monitoring semantics, in particular the fail-stop behavior. So we verify it. In this paper, we present Tagine, a verified compiler that includes a translator from a simple WHILE language (with expressions, statements, and functions) to an instruction-level language of control flow graphs, and a simple dead-code removal optimization for the instruction-level language. Tagine is based on the RTLgen and Deadcode passes of the CompCert C compiler [15]; consequently, it is written in Gallina and verified in Coq. We have also implemented (though not verified) a tagged common-subexpression elimination (CSE) optimization based on CompCert's CSE pass, and designed (though not implemented) a tagged version of CompCert's ConstProp pass.

Our initial work focuses on these compiler passes in order to study the most novel aspects of tagged compilation: moving from source-level control points to per-instruction rules, and performing optimizations in the presence of tag rules. Our key verification result is *policy preservation*: Tagine correctly preserves fail-stop behavior as well as standard semantics in the target code. Although Tagine is currently lacking many important high-level language features, notably memory and pointers, we believe it can be scaled up to a full compiler for Tagged C, a version of C extended with control points and tagging that we are currently designing.

This paper makes the following contributions:

- We describe a general scheme for implementing tag-based fine-grained reference monitoring in high-level language programs by compilation to PIPE-equipped hardware.
- We instantiate this scheme on simple source and target languages equipped with tag-based monitoring and implement the translation from source to target.
- We verify in Coq that the translation preserves monitoring semantics.
- We analyze the requirements for performing standard optimizations, including dead-code elimination, common-subexpression elimination, and constant propagation, in the tagged setting.
- We implement and verify in Coq the dead-code elimination optimization, and implement the CSE optimization.

The remainder of this paper is organized as follows. §2 gives background on the underlying PIPE tagged hardware architecture. §3 shows how the idea of tagged monitoring can be extended to a high-level language. §4 outlines our general approach to compiling a tagged high-level language to PIPE. §5 formalizes Tagine's key pass, RTLgen^T, and describes its verification. §6 discusses optimizations. §7 gives a brief overview of our Coq development. §8 describes related work. §9 describes future work and concludes. The complete Coq sources for Tagine may be found at https://github.com/hope-pdx/Tagine-public.

2 PIPE

PIPE is a collection of architectural features that extend a standard ISA (such as X86, ARM, or RISC-V) with support for tag-based, per-instruction monitoring. The design has been developed over the past eight years by a collaboration of industrial and academic researchers, partly under the aegis of several DARPA programs. Open-source hardware simulators and simple OS ports are available [11], and IP incorporating the designs is currently marketed commercially by Draper Labs and Dover Microsystems [10].

PIPE augments architectural state by associating a *metadata tag* with each value in a register or memory location. Since instructions live in memory, each instruction has a tag. In addition, the processor maintains a *PC tag* conceptually associated with the program counter value; this tag holds metadata characterizing the current control state of the program. Tags are intended to be large—roughly the size of pointers in the underlying architecture. PIPE hardware makes no assumptions about the structure or meaning of tags, which are completely configurable in software.

On each instruction, a PIPE-equipped processor evaluates a *tag rule* to determine whether the instruction should be permitted to execute, and if so, what tags to put on its result values. A distinct tag rule can be associated with each instruction op code; the inputs and outputs of the tag rule are op-code specific. We write $\overline{\texttt{op}}$ for the tag rule associated with instruction op. For example, the RISC-V instruction add r_d,rs_1,rs_2, which adds the contents of registers rs_1 and rs_2 and stores the result in register r_d, has a tag rule with signature

$$\overline{\texttt{add}}\colon (ti, tpc, ts_1, ts_2) \rightarrow \texttt{OK}(tpc', td) + \texttt{Error}$$

Here ti is the tag on the ADD instruction, ts_1 and ts_2 are the tags on source registers rs_1 and rs_2, and tpc is the current PC tag before the instruction executes. The rule result is either OK or Error. In the Error case, the rule has decided that the instruction should not be permitted to execute, and the processor halts or raises a software interrupt to terminate the process. In the OK case, execution continues, after setting two result tags: td, the tag on the value written to destination register rd, and tpc', the new PC tag after the instruction executes.

As another example, the conditional branch instruction beq rs_1,rs_2,*offset* has the slightly simpler rule signature

$$\overline{\texttt{beq}}\colon (ti, tpc, ts_1, ts_2) \rightarrow \texttt{OK}(tpc') + \texttt{Error}$$

because there is no result value to tag. The rule for the store instruction stw rs_2,*offset*(rs_1) takes as an additional input the tag tm of the old contents of the target memory location and generates an additonal output tag tm' for the new contents:

$$\overline{\texttt{stw}}\colon (ti, tpc, ts_1, ts_2, tm) \rightarrow \texttt{OK}(tpc', tm') + \texttt{Error}$$

The tag rules for other instructions follow similar patterns.

A *policy* is a complete collection of tag rules covering all the ISA's opcodes. As a very simple example, we sketch an IFC policy intended to enforce confidentiality. Suppose we wish to distinguish public and secret values and prevent the program from writing secret values to certain memory locations $\mathcal{L}$ representing public channels. To implement this scheme, we can use single boolean values for both value and PC tags, where `true` means secret and `false` means public. We ignore instruction tags in this policy; their utility is explained in §4. We assume that values in memory have been pre-tagged appropriately; in particular, the values in $\mathcal{L}$ are tagged `false`. New values computed from secrets should also be secret. Also, to detect implicit flows, we maintain a "security context level" in the PC tag; initially set to false, it is raised to true if we test a secret value, since this can be used to expose the secret. Here are some of the rules for this policy (the other rules are similar):

$$\begin{aligned}
&\overline{\texttt{add}}(ti, tpc, ts_1, ts_2) \triangleq \\
&\qquad \texttt{OK}(tpc' = tpc, td = ts_1 \lor ts_2 \lor tpc) \\
&\overline{\texttt{beq}}(ti, tpc, ts_1, ts_2) \triangleq \\
&\qquad \texttt{OK}(tpc' = ts_1 \lor ts_2 \lor tpc) \\
&\overline{\texttt{stw}}(ti, tpc, ts_1, ts_2, tm) \triangleq \\
&\qquad \texttt{if } (ts_2 \lor tpc) \rightarrow tm \\
&\qquad \texttt{then OK}(tpc' = tpc, tm' = tm) \\
&\qquad \texttt{else Error}
\end{aligned}$$

One unfortunate feature of this policy is that once the PC tag has been raised by $\overline{\texttt{beq}}$, it remains secret indefinitely; this is a form of "label creep" [20]. While it would be sound to lower the PC tag back to public when control reaches a join point following both branches of the conditional, this is hard to do in a machine-level policy because such join points are not explicit in machine code. We return to this issue in §3.

Software-defined policies are extremely flexible. The policy code can manage its own data structures, even treating tags as pointers into its own (protected) memory space. This is useful for combining policies by treating tags as (pointers to) data structures containing the product of each policy's metadata. Policies can also maintain internal *state* that persists between rule invocations. For example, a memory safety policy might maintain a counter to generate a fresh tag identifier for each object allocated in memory.

If every instruction of the PIPE-enhanced machine had to evaluate a tag rule in software before executing, the system would be ridiculously slow. So PIPE relies on a *rule cache* which contains the results of recent rule evaluations, indexed by a tuple of instruction opcode and input tags. The expectation is that in normal steady-state operation, most instructions will find their tag rule result in the cache. The rule evaluation software is invoked only in case of a cache miss. When designing policies, care must be taken to avoid writing rules that inhibit effective caching.

3 High-Level Language Tag Policies

We next consider how to lift the idea of tag-based policies from machine code to a higher-level language with features such as expressions, structured control flow, and functions. The key idea is to attach tag rules to *control points* in the language's execution semantics. Control points are placed everywhere that a policy might want to inspect tags and possibly halt execution. Tags themselves have arbitrary structure and significance, just as at the PIPE hardware level, and we continue to assume that rule evaluation is implemented in arbitrary software (not necessarily coded in the high-level language being monitored). The tag rule for a control point is passed the tags of relevant values in the environment and, in some cases, returns tags for result values. Also, even though there is no program counter in a high-level language, we retain the idea of a "PC tag" that holds metadata associated with the current control state of the program; it is passed to, and possibly updated by, each tag rule.

For example, an assignment statement of the form $x := e$ has a control point with a tag rule of the form

$$\overline{:=}: (tpc, tx, te) \rightarrow \mathsf{OK}(tpc', tx') + \mathsf{Error}$$

where te is the tag on the value computed for e, tx and tx' are the tags on the contents of x before and after the assignment, and tpc and tpc' are the PC tags before and after the assignment. Note that this rule closely resembles the machine-level rule we saw above for STW, which is not surprising given that an assignment might well be compiled into a store. Similarly, each binary arithmetic expression $l \oplus r$ has a control point that triggers a tag rule

$$\overline{\oplus}: (tpc, tl, tr) \rightarrow \mathsf{OK}(tpc', t') + \mathsf{Error}$$

where tl and tr are the tags of l and r, tpc and tpc' are the PC tags before and after evaluation, and t' is the tag to be associated with the result of the operation. (For a language in which expression evaluation cannot change program state, it might make sense to prevent expression tag rules from changing the PC tag, in which case tpc' would not be included as part of the rule result.) This rule is similar to the machine-level ADD rule, which again is unsurprising.

The control points for structured control statements are more novel. The basic idea is to place a control point wherever the control flow graph splits or joins. For example, an `if-then-else` statement has two control points, one at the conditional test point and another at the join point following the statement:

$$\begin{array}{l} \mathtt{if}\ e_l \bowtie e_r \ \longleftarrow \overline{\mathtt{IfSplit}} \\ \mathtt{then}\ s_1 \\ \mathtt{else}\ s_2 \\ \mathtt{endif} \longleftarrow \overline{\mathtt{IfJoin}} \end{array}$$

The associated tag rule forms are:

$$\begin{array}{l} \overline{\mathtt{IfSplit}}(tpc, \bowtie, t_l, t_r) \rightarrow \mathsf{OK}(tpc') + \mathsf{Error} \\ \overline{\mathtt{IfJoin}}(tpc, tpc_0) \rightarrow \mathsf{OK}(tpc') + \mathsf{Error} \end{array}$$

where $\bowtie$ is an arbitrary binary comparison, t_l and t_r are the tags of the compared values, tpc and tpc' are the PC tags before and after rule execution, and tpc_0 is the original PC tag *at the split point* corresponding to the join point being executed. To show the motivation for this rule signature, consider again the IFC secrecy policy from §2, this time expressed using high-level language tag rules.

$$\begin{array}{ll} \overline{\oplus}(tpc, tl, tr) & \triangleq \mathsf{OK}(tpc' = tpc, t' = tl \vee tr \vee tpc) \\ \overline{:=}(tpc, tx, te) & \triangleq \mathtt{if}\ (te \vee tpc) \rightarrow tx\ \mathtt{then} \\ & \quad \mathsf{OK}(tpc' = tpc, tx' = tx) \\ & \quad \mathtt{else\ Error} \\ \overline{\mathtt{IfSplit}}(tpc, te) & \triangleq \mathsf{OK}(tpc' = te \vee tpc) \\ \overline{\mathtt{IfJoin}}(tpc, tpc_0) & \triangleq \mathsf{OK}(tpc' = tpc_0) \end{array}$$

Recall that we use the PC tag to track the "security context level," which needs to be raised to secret (`true`) when we are executing conditionally under control of a secret, in order to detect implicit flows. A key benefit of using high-level language tag rules here is that the $\overline{\mathtt{IfJoin}}$ control point rule can reset the PC tag to its original value when control leaves the `if` statement, thus potentially allowing subsequent statements to execute at lower secrecy. This rule is only sound because, unlike the machine-level PIPE, the high-level language monitoring framework understands the semantics of structured control flow operators. Other structured statements like `while` and `case` need similar control points.

Finally, control points are also placed before and after each function call site and at each function entry and exit. Rules executed at these points can inspect the tags on function parameter values as well as on the function itself. Again, this is also information that would be difficult or impossible to collect at machine level.

Note that the set of control points and tag rule signatures will typically be fixed once and for all when designing monitoring for the high-level language. They should therefore be designed to be sufficiently general to implement any policy of interest. Our control point design is based primarily on consideration of IFC, memory safety, and compartmentalization policies. Of course, adequacy of the control point design cannot be absolutely guaranteed, as new kinds of policies may be invented later.

4 Compilation Approach

Tag-based high-level language policies could be monitored in software, e.g. by generating code to evaluate the rule functions and interleaving it with normal execution code in the spirit of aspect-oriented weaving [13]. But given the density of control points, the overhead of this approach would probably be very high. We instead opt to *compile* the tagged high-level language to machine code for a PIPE-equipped processor, in such a way that the reference monitoring behavior of the source is preserved in the target.

Such a *policy-preserving* compiler can be built by modifying a standard compiler from (untagged) source to (untagged) target. The task is simplified by the fact that the structure and meaning of tags is largely the same at both levels, and invocations of the source-level tag rule evaluation code can be embedded directly in the target-level rules.

The main challenge is that the high-level monitor associates tag rules with (language-dependent) control points, whereas the PIPE framework associates them with each individual machine instruction. While some control points, such as those at arithmetic operations, correspond naturally to single instructions, others will correspond to multiple instructions. Moreover, many different high-level features will compile to instructions that use the same opcodes. For example, an add instruction in the target code might be implementing an explicit addition expression in the source code, but it might equally well have been generated by the compiler as part of array addressing or stack frame management. Clearly the opcode alone is not sufficient to determine which source tag rule should be executed at a given target instruction.

To solve this problem, we rely on the fact that PIPE associates a separate tag *ti* with each instruction in memory, and feeds it as one of the inputs to the rule evaluated at each execution step. Instruction tags (I-tags) effectively let us design a customized instruction set that *refines* the hardware ISA by providing different variants of some opcodes based on the instruction's semantic role in the policy being enforced. Here, we use I-tags to specify *provenance*, i.e. the source code construct from which the instruction was generated. The opcode's tag rule can dispatch on the I-tag to evaluate the relevant source tag rule, if any, for each possible provenance. For example, if an add instruction is generated from an explicit + expression, it might be tagged IT+, whereas otherwise it might be tagged ITdc (for "don't care"). The tag rule for add could then be:

```
$\overline{\texttt{add}}(ti, tpc, ts_1, ts_2) \triangleq$
  match $ti$ with
    IT+ $\Rightarrow \overline{+}(tpc, ts_1, ts_2)$
    ITdc $\Rightarrow$ OK$(tpc' = tpc, td = ts_1)$
```

where in the "don't care" case we arbitrarily choose to propagate the left operand's tag to the result tag.

This "piggybacking" technique, in which we trigger the source control point rule check by attaching it to the PIPE-rule for an instruction that is already being generated, will work for most source language constructs. But sometimes a tag-aware compiler must generate additional instructions into the target code just to manage tags. One example of this is the if-then-else statement. As described in §3, the PC tag at the control split point must be saved so it can be passed to the $\overline{\texttt{IfJoin}}$ at the join point. To do this, we can generate a target instruction at the split point (for example a mov with a particular I-tag) whose only purpose is to copy the PC tag into (the tag portion of) a register or onto the stack until it is needed at the join point.[2] Similar dummy instructions may be needed to track the PC tag at other structured control statements, or when marshalling the tags of function arguments to feed them to a tag rule at a function entry control point, etc.

To formalize correctness of a tag-aware compiler, we start by defining semantics for source and target languages that incorporate tag-based monitoring by construction. Both semantics are parameterized by tag policies; the source tag rules are arbitrary, and the target tag rules embed the source rules. Tag policy violations lead to fail-stop states, which are distinct from stuck states or other kinds of errors. Then a policy-preserving compiler is one that preserves both ordinary computation behaviors and fail-stop behaviors. In particular, the compiler must *not* treat source policy errors as undefined behaviors that can be refined into arbitrary valid executions in the target.

In principle, a policy-preserving compiler can be completely ignorant of the actual source language policy, so that a single version of the generated code can be used to enforce arbitrary source policies just by changing the tag rule evaluation code. To achieve this, the source and target must invoke the same sequence of source rules, with the same arguments; since the rules are arbitrary, any change in a rule invocation might change the fail-stop behavior of the overall rule sequence. In addition to maximizing runtime flexibility, this approach also keeps the compiler and its verification relatively simple.

However, maintaining this *policy-independence* property may slow down target code unnecessarily. For example, the compilation scheme for saving and restoring PC tags described above introduces extra instructions and adds extra pressure on register use, which may degrade performance: if the policy being run doesn't actually make use of the PC tag, adding this overhead is pointless. More subtly, the need to preserve arbitrary tag rule semantics inhibits the applicability of many simple code optimizations such as dead code elimination, common subexpression elimination, or constant folding and propagation. For example, a conventional optimizer might use a standard liveness analysis to eliminate an add instruction if its result register value is never used. However, in general it is not sound to skip evaluation of the instruction's associated tag rule. Although calculating the tag on the instruction result is not important—since that result is never used, its tag is not read either—the rule might fail-stop, change the PC tag, or change internal policy state.

In general, determining statically whether a tag rule execution can be skipped is clearly uncomputable. However, our

[2]An alternative approach would be for the policy to maintain a stack of split point PC tags in the current PC tag itself, avoiding the need to plumb them through the rules. Although this would be simpler than our scheme, its worst-case rule cache efficiency would be poorer.

analysis of the existing CompCert optimizations on scalar locals has identified several simple and intuitive conditions on tag rules that, in various combinations, suffice to keep these optimizations sound. These conditions might include not fail-stopping, not altering the PC tag, or being insensitive to the input PC tag. Thus, to enable optimizations, Tagine must know at least something about the rules, but not necessarily have their full definitions. Adopting this condition-based approach helps decouple the compiler from the details of the rules, and will allow the same compiled code to run with multiple sets of rules as long as they obey the conditions.

5 The RTLgen$^{\mathrm{T}}$ Compiler Pass

To study the verification of policy preservation in detail, we have designed and verified RTLgen$^{\mathrm{T}}$, a small prototype compiler pass that translates HLL, a simple tagged WHILE language (with expressions, statements, and functions) to RTL$^{\mathrm{T}}$, an instruction-level language represented in an explicit control-flow graph (CFG). We focus on this compiler pass because it is here that high-level program structures (statements and expressions) are transformed into instructions, and hence where the tag rules for control points must be attached to appropriate instruction positions.

HLL, RTL$^{\mathrm{T}}$ and RTLgen$^{\mathrm{T}}$ are closely based on the CminorSel and RTL languages and the RTLgen pass of the CompCert compiler [15], and our proof of policy preservation is structured similarly to Leroy's correctness proof. We prove a forward simulation result which lifts into a refinement result thanks to the determinism of the target language [15]. As usual, this proof involves establishing and maintaining a matching relation between corresponding source and target states. We believe this is one of the more challenging parts of producing a full-scale CompCert variant for Tagged C, which is our long-term goal.

5.1 The Source Language : HLL

HLL is a simple, untyped, deterministic, imperative language with expressions, structured statements, and functions with local variables. For simplicity, we assume that the language has no I/O facilities, but the final value returned by the `main` function is observable.

HLL's semantics is implicitly parameterized by a high-level rule policy $\mathcal{P}$, which consists of a set of value tags $\mathbf{T}$, ranged over by t; a set of PC tags $\mathbf{P}$, ranged over by p; a set of tag rules covering all possible control points; and a set of of possible tag errors carried by `ERROR` returns (which we elided for simplicity in §2 and §3), ranged over by *err*.

Figure 1 describes the syntax of HLL expressions and statements, which are largely standard, except that literal constant expressions are *atoms* [5], consisting of a natural number v paired with a value tag t, written $v@t$ and ranged over by a. HLL has explicit WHILE loops in place of CminorSel's

$$
\begin{array}{lll}
e := & v@t & \text{literal constants} \\
\mid & x & \text{local variables} \\
\mid & e \oplus e & \oplus := + \mid - \\
\\
s := & \texttt{Skip} & \text{empty/no-op} \\
\mid & s\,;\,s & \text{sequencing} \\
\mid & x := e & \text{assignment (to local)} \\
\mid & \texttt{If}\ (e \bowtie e)\ \texttt{Then}\ s\ \texttt{Else}\ s & \text{ifs/conditionals} \\
\mid & \texttt{While}\ (e \bowtie e)\ s & \text{while loops} \\
\mid & x = f(\vec{e}) & \text{function calls} \\
\mid & \texttt{Return}(e) & \text{function returns} \\
\\
\bowtie := & \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{<} \mid \texttt{>=} \mid \texttt{>} & \text{relational operator}
\end{array}
$$

Figure 1. Syntax of HLL expressions and statements

$$
\frac{\overline{\texttt{Const}}\ p\ t \Rightarrow t'}{E, p \vdash v@t \Downarrow v@t'}\ [_{eval}\mathsf{Const}]
$$

$$
\frac{E(x) = v@t \qquad \overline{\texttt{Var}}\ p\ t \Rightarrow t'}{E, p \vdash x \Downarrow v@t'}\ [_{eval}\mathsf{Var}]
$$

$$
\frac{\begin{array}{l} E, p \vdash e_1 \Downarrow v_1@t_1 \\ E, p \vdash e_2 \Downarrow v_2@t_2 \qquad \overline{\oplus}\ p\ t_1\ t_2 \Rightarrow t' \end{array}}{E, p \vdash e_1 \oplus e_2 \Downarrow (v_1 \oplus v_2)@t'}\ [_{eval}\mathsf{Op}]
$$

Figure 2. Evaluation of HLL expressions (excluding errors)

general-purpose LOOP, BLOCK, and EXIT statements, because it is difficult to design sensible IFC policies for the latter. A function definition consists of a parameter list, local variable declarations, body (a statement), and a function tag $p \in \mathbf{P}$ which will be made available to the tag rule executed at function entry. A program is just a collection of functions, with a distinguished `main` function.

Following CminorSel, we give a relational natural semantics for expressions, and a transition system for statements and functions. For expressions, we define the judgement

$$E, p \vdash e \Downarrow (a + err)$$

where e is the expression being evaluated, E is the current environment, mapping variable to atoms, and p is the current PC tag. The result of evaluation is either an atom a or a tag error *err*. Figure 2 gives the non-error cases for this judgement, which are standard except for the tags and tag rule invocations. We write $\Rightarrow$ to indicate tag rule evaluation. Henceforth we use metavariables (without explicit `OK` and `ERROR` constructors) to indicate the type of rule results, and adopt a juxtaposition-as-application style for rule arguments.

Since expressions are pure, we make the assumption that no policy will ever need expression evaluation to change the PC tag. We omit the error cases induced when a tag rules returns an *err*; as usual, propagation of errors leads to an

$$\frac{}{\mathcal{S}(F,(s_1;s_2),k,p,c,E) \rightarrow \mathcal{S}(F,s_1,(s_2;k),p,c,E)}\ [\textit{Seq}]$$

$$\frac{}{\mathcal{S}(F,\texttt{Skip},(s;k),p,c,E) \rightarrow \mathcal{S}(F,s,k,p,c,E)}\ [\textit{SkipSeq}]$$

$$\frac{\overline{\textit{join}}\ p\ p_s \Rightarrow p'}{\mathcal{S}(F,\texttt{Skip},(\overline{\textit{join}},p_s);k,p,c,E) \rightarrow \mathcal{S}(F,\texttt{Skip},k,p',c,E)}\ [\textit{SkipJoin}]$$

$$\frac{E,p \vdash e \Downarrow v@t \qquad E(x) = \_@t_{old} \qquad \overline{\coloneqq}\ p\ t_{old}\ t \Rightarrow (p',t')}{\mathcal{S}(F,(x \coloneqq e),k,p,c,E) \rightarrow \mathcal{S}(F,\texttt{Skip},k,p',c,E[x \mapsto v@t'])}\ [\textit{Assign}]$$

$$\frac{E,p \vdash e_1 \Downarrow v_1@t_1 \qquad E,p \vdash e_2 \Downarrow v_2@t_2 \qquad \overline{\texttt{IfSplit}}\ p \bowtie t_1\ t_2 \Rightarrow p' \qquad s = \text{if } v_1 \bowtie v_2 \text{ then } s_t \text{ else } s_f}{\mathcal{S}(F,\texttt{If }(e_1 \bowtie e_2)\texttt{ Then } s_t \texttt{ Else } s_f,k,p,c,E) \rightarrow \mathcal{S}(F,s,(\overline{\texttt{IfJoin}},p);k,p',c,E)}\ [\textit{Cond}]$$

Figure 3. Selected HLL statement transitions.

explosion of judgements. The error rules fix the order of sub-expression evaluation, since an error in the left operand is propagated in favor of one in the right operand.

The semantics for HLL statements is given by a transition system between *program states* $\mathcal{S}t$. The transition relation, written $\mathcal{S}t \longrightarrow \mathcal{S}t'$, describes a single execution step.

Borrowing from CompCert, we distinguish function internal, entry, and exit states, and add a new state $\mathcal{E}(err)$ representing a fail-stop due to tag error *err*.

$\mathcal{S}t \coloneqq \mathcal{S}(F,s,k,p,c,E)$	regular states
$\mid \mathcal{C}(F,\vec{a},p,c)$	call states
$\mid \mathcal{R}(a,p,c)$	return states
$\mid \mathcal{E}(err)$	fail-stop states

Regular states correspond to execution within a function and carry the current function F; the current program point, represented by a statement-under-focus s and a local continuation k; the current PC tag p; a call continuation c, representing the call stack; and a local environment E that maps variables to atoms. In call states, F is the callee, and $\vec{a}$ its parameters. Return states carry a returned atom, a.

CompCert, following Appel and Blazy [3], combines local and callstack continuations into one continuation, but we find it simpler and clearer to separate them. Local continuations obey the following grammar:

$k \coloneqq \texttt{emp}$	return
$\mid s;\ k$	continue with s, then do k
$\mid (\overline{\textit{join}}, p_s); k$	update PC tag; then do k

The novelty here is the PC tag update, which is explained below. To save space and remain focused on the key ideas of tag-based compilation, we will not discuss function calls, function entry and exit states, or call continuations further in this paper; for details, see the full Coq development.

An *initial state* (for a program) is the call state where F is `main()` (which takes no parameters) and c is empty. Program execution is described by the transitive closure of the steps taken in the semantics from the initial state. *Final states* are return states with an empty call stack, and are those in which a program is considered to have terminated normally. There are no transitions out of fail-stop states or final states.

A program may exhibit one of the following *behaviors*:

- **Terminate** with result a, when program execution reaches a final state carrying a.
- **Fail-stop** with tag error err, when program execution reaches an error state carrying err.
- **Diverge**, when program execution may always take another step in the transition semantics.
- **Go Wrong** (or, "get stuck"), when program execution cannot take a step in the transition semantics (but is not in a fail-stop, or final state).

We write $S_{\mathcal{P}} \Downarrow B$ to mean that a program S, executing under policy $\mathcal{P}$, exhibits behavior B. We use behaviors to help formalize a notion of semantic preservation (§5.4).

Figure 3 gives transition judgements for a small selection of statements. The non-tag aspects of these are standard. Note that the local continuation grows under sequencing, and is consumed when the statement under focus is `Skip`. The semantics precisely specifies the position and signature of each control point. For example, *Assign* evaluates the right-hand side to an atom with tag t, fetches the tag t_{old} of the existing value in x, and then passes them to $\overline{\coloneqq}$ together with the PC tag p. If the rule does not fail, it returns the new PC tag p' and a tag t' to associate with the new value in x; we step to a new regular state with the new PC tag and the environment entry for x suitably updated. *AssignRuleErr* (not shown) applies when $\overline{\coloneqq}$ returns a tag error *err*, in which case we step to $\mathcal{E}(err)$.

The control point semantics for `if-then-else` are more complicated. As discussed in §3, we want one control point where the conditional is evaluated and another at the implicit join point following the statement. The first of these is specified by the invocation of $\overline{\texttt{IfSplit}}$ in the premises of

$$\frac{F_g(n) = \mathsf{nop}\ n'\ @\ \mathit{itag} \qquad \overline{\mathsf{nop}}\ \mathit{itag}\ p \Rightarrow p'}{\mathcal{S}(F, n, p, c, B) \longrightarrow \mathcal{S}(F, n', p', c, B)}\ [{}_{\mathit{exec}}\mathit{Nop}]$$

$$\frac{F_g(n) = \mathsf{mov}\ r_s\ r_d\ n'\ @\ \mathit{itag} \qquad \begin{array}{c} B(r_s) = v_s@t_s \\ B(r_d) = v_d@t_d \end{array} \qquad \overline{\mathsf{mov}}\ \mathit{itag}\ p\ t_s\ t_d \Rightarrow (p', t')}{\mathcal{S}(F, n, p, c, B) \longrightarrow \mathcal{S}(F, n', p', c, B[r_d \mapsto v_s@t'])}\ [{}_{\mathit{exec}}\mathit{Mov}]$$

$$\frac{F_g(n) = \mathsf{movi}\ v@t\ r_d\ n'\ @\ \mathit{itag} \qquad \overline{\mathsf{movi}}\ \mathit{itag}\ p\ t \Rightarrow (p', t')}{\mathcal{S}(F, n, p, c, B) \longrightarrow \mathcal{S}(F, n', p', c, B[r_d \mapsto v@t'])}\ [{}_{\mathit{exec}}\mathit{MovI}]$$

$$\frac{F_g(n) = \mathsf{op}_\oplus\ \ r_1\ r_2\ r_d\ n'\ @\ \mathit{itag} \qquad B(r_1) = v_1@t_1 \qquad B(r_2) = v_2@t_2 \qquad \overline{\mathsf{op}_\oplus}\ \mathit{itag}\ p\ t_1\ t_2 \Rightarrow (p', t')}{\mathcal{S}(F, n, p, c, B) \longrightarrow \mathcal{S}(F, n', p', c, B[r_d \mapsto (v_1 \oplus v_2)@t'])}\ [{}_{\mathit{exec}}\mathit{Op}]$$

$$\frac{F_g(n) = \mathsf{cond}_{\bowtie}\ r_1\ r_2\ n_t\ n_f\ @\ \mathit{itag} \qquad \begin{array}{c} B(r_1) = v_1@t_1 \\ B(r_2) = v_2@t_2 \end{array} \qquad \begin{array}{c} \overline{\mathsf{cond}_{\bowtie}}\ \mathit{itag}\ p\ t_1\ t_2 \Rightarrow p' \\ n' = \text{if } v_1 \bowtie v_2 \text{ then } n_t \text{ else } n_f \end{array}}{\mathcal{S}(F, n, p, c, B) \longrightarrow \mathcal{S}(F, n', p', c, B)}\ [{}_{\mathit{exec}}\mathit{Cond}]$$

Figure 4. Selected RTL$^\text{T}$ instruction transitions

$i := \mathsf{nop}\ n$	no-op (go to n)
$\mid \mathsf{op}_\oplus\ r_{s1}\ r_{s2}\ r_d\ n$	binary operation
$\mid \mathsf{mov}\ r_s\ r_d\ n$	move
$\mid \mathsf{movi}\ a\ r_d\ n$	move immediate
$\mid \mathsf{cond}_{\bowtie}\ r_1\ r_2\ n_t\ n_f$	branch
$\mid \mathsf{call}\ f\ \vec{r}_s\ r_d\ n$	call
$\mid \mathsf{ret}\ r$	return

Figure 5. Syntax of RTL$^\text{T}$ instructions

Cond. But the second join point needs to be associated with the *continuation* of the statement, and it needs to be given the PC tag from the split point as one of its arguments. To do this, we generate a continuation of the form $(\overline{join}, p_s); k$. ($\overline{join}$ is a metavariable.) This continuation indicates that the program passed through a split point earlier (in this case a conditional) and has now reached the corresponding join point. It is processed by judgement *SkipJoin*, which invokes $\overline{join}\ p\ p_s$ (p and p_s are the current and split-point PC tags, respectively), updates the PC tag to this result, and proceeds with continuation k. For the `if-then-else` join we specify $\overline{join}$ to be $\overline{\texttt{IfJoin}}$ and p_s to be the split-point PC tag. The same continuation mechanism is used for `while` statements (specifying, say, $\overline{\texttt{WhileExit}}$ for $\overline{join}$). A similar technique is used to specify the control points associated with calls and returns.

5.2 The Target Language: RTL$^\text{T}$

RTL$^\text{T}$ is a deterministic, 3-address code, register transfer abstract machine language based on CFG's: it represents functions as graphs, where each node is an individual instruction. Like HLL, its semantics are parameterized by an instruction-level rule policy, i.e. sets of value tags **T** and PC tags **P**, a set of tag rules covering each possible instruction, and a set of possible error tags ranged over by *err*. In addition, RTL$^\text{T}$ is parameterized by an arbitrary set of I-tags, ranged over by *itag*; each instruction is labeled with an I-tag.

Figure 5 describes the syntax of RTL$^\text{T}$ instructions. We use n to range over labels for graph nodes and r to range over pseudo-registers. Each instruction carries the label of the next node(s) to be executed. We reuse the arithmetic and relational operators found in HLL.

An RTL$^\text{T}$ function F is described by a graph F_g, which is a finite partial mapping from nodes to instructions; an entry-point node; registers containing parameters; and a function tag. Each function has an infinite bank of registers.

Program states and behaviors are similar to those in HLL. A difference is in regular states $\mathcal{S}(F, n, p, c, B)$, where program points are now tracked via node labels n (there are no local continuations) and variable environments are replaced by register banks B, which map registers to atoms. Figure 4 gives transition judgements for RTL$^\text{T}$ instructions, omitting calls, returns, and error rules. Each instruction invokes a tag rule, which is passed the instruction's I-tag in addition to value and PC tags.

5.3 Compilation

We now describe compilation of HLL into RTL$^\text{T}$ by example. Compilation involves the translation of statements and expressions into instructions, but also the injection of an HLL tag policy $\mathcal{P}$ into an equivalent RTL$^\text{T}$ policy, $\mathcal{I}(\mathcal{P})$. The per-opcode rules in $\mathcal{I}(\mathcal{P})$ begin by dispatching on I-tags: an instruction's tag rule effectively depends on both its opcode and its I-tag. For compactness, we write rules in the form *opcode I-tag parameters* $\triangleq$ *rule-body*. Many I-tags correspond directly to source constructs, and their rule bodies simply invoke the corresponding HLL tag rule. Other I-tags signal administrative tag operations generated by the compiler. We give some of the rules for $\mathcal{I}(\mathcal{P})$ below, interleaved with the discussion of the relevant compilation cases.

The translation from HLL to RTL$^{\mathrm{T}}$ decomposes expressions into linear sequences of RTL$^{\mathrm{T}}$ instructions, and recursively translates statements into CFG's. Functions and programs in RTL$^{\mathrm{T}}$ are extremely similar to those in HLL, so the compiler has little to do for these, and we omit further discussion of them.

Since most RTL$^{\mathrm{T}}$ instructions incorporate an explicit *successor node*, the compiler builds CFG's in reverse execution order. Each translation function takes a source fragment, a variable map (holding registers for parameters and locals), and a target successor node; it returns an entry node n_e into the modified CFG.

Translation of expressions We write the expression translation function as

$$n_e : r = [\![e]\!]_{n_s}$$

where e is the expression being translated, n_s is the successor node, n_e is the entry node returned, and r is a fresh register generated to hold the result of e. The output of translation is a CFG, rooted at n_e and exiting to n_s, whose execution will have the effect of evaluating e and (if this is successful) placing its atom in register r before continuing to n_s. If evaluation of e leads to a tag error err, execution of the subgraph will halt in state $\mathcal{E}(err)$.

The translation function is defined by cases on syntax constructors. We show the result of each translation as a CFG diagram. Rounded white boxes represent single graph nodes showing an instruction, its I-tag and node label. Recursive calls to translation functions generate subgraphs, represented by shaded rectangular boxes.

Here are the cases for constants, variables and operations, where r_x is the register mapped to hold the HLL variable x:

$$\begin{tikzcd} n_e : r = [\![a]\!]_{n_s} \arrow[d, Rightarrow] \\ \boxed{n_e : \texttt{movi}\ a\ r\ n_s\ @\ \texttt{ITconst}} \end{tikzcd} \qquad \begin{tikzcd} n_e : r = [\![x]\!]_{n_s} \arrow[d, Rightarrow] \\ \boxed{n_e : \texttt{mov}\ r_x\ r\ n_s\ @\ \texttt{ITvar}} \end{tikzcd}$$

$$\begin{tikzcd} n_e : r = [\![e_1 \oplus e_2]\!]_{n_s} \arrow[d, Rightarrow] \\ \boxed{n_e : r_1 = [\![e_1]\!]_{n_1}} \arrow[d] \\ \boxed{n_1 : r_2 = [\![e_2]\!]_{n_2}} \arrow[d] \\ \boxed{n_2 : \texttt{op}_{\oplus}\ r_1\ r_2\ r\ n_s\ @\ \texttt{ITop}\ \oplus} \end{tikzcd}$$

To understand the tag-related behavior of the generated instructions, we must also examine the behavior of the RTL$^{\mathrm{T}}$ tag policy $\mathcal{I}(\mathcal{P})$, under which this code will execute. These rules essentially replicate HLL's tag processing in RTL$^{\mathrm{T}}$. The definitions for expressions are:

$$\begin{array}{llll} \overline{\texttt{movi}} & \texttt{ITconst} & p\ t & \triangleq \overline{\texttt{Const}}\ p\ t \\ \overline{\texttt{mov}} & \texttt{ITvar} & p\ t_s\ t_d & \triangleq \overline{\texttt{Var}}\ p\ t_s \\ \overline{\texttt{op}_{\oplus}} & \texttt{IT}\oplus & p\ ts_1\ ts_2 & \triangleq \begin{cases} (p,t) & \text{if } \overline{\oplus}\ p\ ts_1\ ts_2 \Rightarrow t \\ err & \text{if } \overline{\oplus}\ p\ ts_1\ ts_2 \Rightarrow err \end{cases} \end{array}$$

Translation of statements and functions The statement translation function is written $n_e : [\![s]\!]_{n_s}$ where s is a source statement, n_s a successor node parameter and n_e the generated entry node. Executing the resulting CFG, rooted at n_e and exiting to n_s, will have the same effect in RTL$^{\mathrm{T}}$ as statement s does in HLL. As with expressions, if evaluation of s leads to a tag error, execution halts in $\mathcal{E}(err)$. Assignment and sequencing CFG's and their rule definitions in $\mathcal{I}(\mathcal{P})$:

$$\begin{tikzcd} n_e : [\![s_1;\ s_2]\!]_{n_s} \arrow[d, Rightarrow] \\ \boxed{n_e : [\![s_1]\!]_{n_1}} \arrow[d] \\ \boxed{n_1 : [\![s_2]\!]_{n_s}} \end{tikzcd} \qquad \begin{tikzcd} n_e : [\![x := e]\!]_{n_s} \arrow[d, Rightarrow] \\ \boxed{n_e : r = [\![e]\!]_{n_1}} \arrow[d] \\ \boxed{n_1 : \texttt{mov}\ r\ r_x\ n_s\ @\ \texttt{ITassign}} \end{tikzcd}$$

$$\begin{array}{llll} \overline{\texttt{mov}} & \texttt{ITassign} & p\ t_s\ t_d & \triangleq \overline{:=}\ p\ t_d\ t_s \end{array}$$

Note that assignments and variable expressions both compile to a mov, but the I-tags encode enough information about the source provenance of the instruction to reproduce the correct rule processing in the target.

Pseudo-instructions and join points The most interesting cases for statement compilation in our tagged world are conditionals and while loops; for brevity, we focus on the former.

$$\begin{tikzcd} & n_e : [\![\texttt{If}\ (e_1 \bowtie e_2)\ \texttt{Then}\ s_t\ \texttt{Else}\ s_f]\!]_{n_s} \arrow[d, Rightarrow] & \\ & \boxed{n_e : \texttt{mov}\ r_{pc}\ r_{pc}\ n_1\ @\ \texttt{ITsavePC}} \arrow[d] & \\ & \boxed{n_1 : r_1 = [\![e_1]\!]_{n_2}} \arrow[d] & \\ & \boxed{n_2 : r_2 = [\![e_2]\!]_{n_3}} \arrow[d] & \\ & \boxed{n_3 : \texttt{cond}_{\bowtie}\ r_1\ r_2\ n_t\ n_f\ @\ \texttt{ITifSplit}} \arrow[dl] \arrow[dr] & \\ \boxed{n_t : [\![s_t]\!]_{n_4}} \arrow[dr] & & \boxed{n_f : [\![s_f]\!]_{n_4}} \arrow[dl] \\ & \boxed{n_4 : \texttt{mov}\ r_{pc}\ r_{pc}\ n_s\ @\ \texttt{ITifJoin}} & \end{tikzcd}$$

$$\begin{array}{llll} \overline{\texttt{mov}} & \texttt{ITsavePC} & p\ \_\ \_ & \triangleq (p,p) \\ \overline{\texttt{cond}_{\bowtie}} & \texttt{ITifSplit} & p \bowtie t_1\ t_2 & \triangleq \overline{\texttt{IfSplit}}\ p \bowtie t_1\ t_2 \\ \overline{\texttt{mov}} & \texttt{ITifJoin} & p\ t_s\ t_d & \triangleq \overline{\texttt{IfJoin}}\ p\ t_s \end{array}$$

Recall that these statements have multiple control points, corresponding to splits and joins in program control flow, and that the PC tag at the split point needs to be passed as a parameter to the rule at the matching join point. The compiled versions of these statements use pseudo-instructions to save the split point PC tag (and a dummy value) and recover this PC tag to use in join point rules. These instructions are implemented as `mov`s and distinguished by their I-tags.

When used to save PC tags at split points, `mov` ignores the tag of the source register, and moves the current PC tag into the tag portion of the destination register. When used to recover the PC tag, it just passes the tag portion of the register (t_s) to the join point rule . The pseudo-instructions for split and join points are always generated in pairs, and for each instruction, the same "save" register r_{pc} is used for both source and destination, leaving the value part of r_{pc} unchanged. Since split-join pairs can be arbitrarily nested, the set of "save" registers that are live at any given program point form a stack.

5.4 Verification Approach

The verification of RTLgen$^{\mathrm{T}}$ and Deadcode$^{\mathrm{T}}$ (§6.1) follow the general framework laid out in CompCert [15]. Our notion of semantics preservation is *refinement-for-safe-programs*: Any behavior exhibited by the target program must be one exhibited by the source program. In other words, *compilation should not introduce new behaviors in the target.* Behavioral equality is defined up to equality of results and errors, i.e., **Terminate** a = **Terminate** a' iff $a = a'$ and **Fail-stop** err = **Fail-stop** err' iff $err = err'$. Further, we only want to consider *safe* source programs, i.e. those that do not **Go Wrong**. This is to allow the compiler the flexibility of, for example, removing a division by zero whose result is unused.

Each Tagine pass is proved via a *forward* simulation

$$\begin{tikzcd}
\sigma_1 \arrow[r, "\sigma_1 \rightarrow \sigma_2"] \arrow[d, dash, "\sigma_1 \sim \varsigma_1"'] & \sigma_2 \arrow[d, dash, dashed, "\sigma_2 \sim \varsigma_2"] \\
\varsigma_1 \arrow[r, dashed, "\varsigma_1 \rightarrow^* \varsigma_2"'] & \varsigma_2
\end{tikzcd}$$

which, coupled with the determinacy of the target's semantics, implies refinement (see Leroy [15] for more details). In the commuting diagram, which shows a general forward simulation, σ ranges over source states and ς over target states. Solid lines represent premises, dashed lines proof obligations, $\rightharpoonup^*$ represents zero-or-more steps and $\sim$ is a *matching relation* between source and target program states.

Matching relations—which describe things such as the environments, what computations are to be performed next, or the functions on the call stack, in each language—are exactly the context that provide the formal meaning of what it means for the executions to be equivalent. Defining the matching relation is usually the most intricate part of a simulation argument. Intuitively, matching relates equivalent points in the source and target programs' execution. For simple passes, e.g., an optimization pass whose compilation scheme replaces one instruction with another (as opposed to translating one source construct to multiple target constructs), the matching relation is easy to define, as the execution moves in lockstep. RTLgen$^{\mathrm{T}}$ is, of course, more complicated, further underscoring that it was a key pass to study.

5.5 RTLgen$^{\mathrm{T}}$ Theorem and Matching Relation

Theorem 5.1. ***(Semantic Preservation RTLgen$^{\mathrm{T}}$)*** *Let S and $\mathcal{P}$ be a HLL program and policy respectively. Let C be the RTL$^{\mathrm{T}}$ result of compiling S and $\mathcal{I}(\mathcal{P})$ the result of compiling $\mathcal{P}$. Under their respective semantics and policies: If S does not* ***Go Wrong****, then the behavior displayed by C is displayed by S. Formally: $\forall B.safe(S) \rightarrow C_{\mathcal{I}(\mathcal{P})} \Downarrow B \rightarrow S_{\mathcal{P}} \Downarrow B$.*

The proof of theorem 5.1 is mostly straightforward; we focus discussion on subtle or novel clauses of the matching relation.

The matching relation is hierarchical. It is defined on states, which in turn requires the definition of matching on functions, atoms, tag errors, call stacks, environments etc. Many of these constituent matchings are straightforward, because the matched structures are either very similar in both languages (e.g., call, return, and fail-stop states) or because they are relatively simple (e.g., atoms, tag errors, or environments). We focus our discussion on regular states, which describe computation internal to a function. Recall that HLL regular states $\mathcal{S}(F, s, k, p, c, E)$ describe the current computation with a statement-under-focus, s, and a (local) continuation, k, which describes the rest of the function body. (All HLL expressions are embedded in statements.) RTL$^{\mathrm{T}}$ regular states $\mathcal{S}(F, n, p, c, B)$ describe the current computation with a node label n pointing at an instruction in the CFG (contained in the function F). To preserve semantics, we just need to ensure that HLL and RTL$^{\mathrm{T}}$ functions (in regular states) step to a return state at the same time, carrying matching atoms. However, the only way to guarantee this is to make sure that the computation in s and k match the computation starting at n, i.e., that they update environment E and register bank B in parallel, that they make the same intermediate computations, etc. Thus, we need to define matching for statements, continuations and expressions. Each of these individual relations carries pertinent information, such as the state of the register bank or specially pre-defined return registers, which we elide until we discuss particular clauses of the relation.

We match each statement s to a CFG *interval* $[n_e,n_s)$, written $s \sim [n_e, n_s)$. An interval is a contiguous chain of instructions (similar to the diagrams in §5.3) that starts at the instruction labeled n_e and ends at an instruction whose successor is n_s. (The instruction at n_s is not part of the interval.) Intervals need not be *linear*; they may branch so long as the last instruction in each branch has n_s as a sucessor. Similarly, $e \sim_r [n_e, n_s)$ denotes that expression e matches an interval, where r is the register in which the expression's atom will be stored by the code in the interval. Expression and statement matching are naturally closely related to their compilation schemes (cf §5.3). Here are some simple cases:

Skip `Skip` $\sim [n_s, n_s)$. (Here the interval $[n_s, n_s)$ is empty.)

Seq $s_1; s_2 \sim [n_e, n_s)$ if $\exists n_1$ s.t. $s_1 \sim [n_e, n_1) \ \wedge \ s_2 \sim [n_1, n_s)$.

Add $e_1 + e_2 \sim_r [n_e, n_s)$ if $\exists n_1, n_2, r_1, r_2$ s.t.
- $e_1 \sim_{r_1} [n_e, n_1)$ and $e_2 \sim_{r_2} [n_1, n_2)$
- n_2: $\mathtt{op}_+ \ r_1 \ r_2 \ r \ n_s$
- sundry side conditions, e.g.: $[n_1, n_2)$ does not overwrite r_1, $[n_e, n_s)$ does not overwrite registers holding variables of the HLL environment, etc.

Function termination provides some more complex examples of matching relations. In HLL, a function terminates when a `Return` statement is under focus, or when a function "falls through" by reaching the end of its body (`Skip` under focus and empty k). When falling through, functions return a default atom a_d. RTL$^\mathrm{T}$ functions have a single exit point, so function compilation predefines a return value register r_{ret}, and two instructions:
- n_{def} : `movi` $a_d \ r_{ret} \ n_{ret}$
- n_{ret} : `ret` r_{ret}

The matching relation for `Return` statements may look disconcerting upon first examination: label n_s is free because the `mov` jumps to the pre-defined exit n_{ret}.

Return `Return` $e \sim [n_e, n_s)$ if $\exists n_1, r_1$ s.t.
- $e \sim_{r_1} [n_e, n_1)$
- n_1: `mov` $r_1 \ r_{ret} \ n_{ret}$

Continuation matching is defined in terms of a *single* CFG node, as illustrated by the fall-through continuation.

Fall through `emp` $\sim n_{def}$ if
- n_{def} : `movi` $a_d \ r_{ret} \ n_{ret}$
- n_{ret} : `ret` r_{ret}

n_{ret} is effectively the end of the interval corresponding to every continuation.

The matching clauses we have seen so far are only slightly modified from CompCert's RTLgen. The next clause is novel, however.

k-Join $(\overline{join}, p_s); k \sim n_e$ if $\exists n_1$ s.t.
- n_e: `mov` $r_{pc} \ r_{pc} \ n_1$ @ *itag*
- $k \sim n_1$
- $\forall q, q_{s \in \mathbf{P}}$. $\overline{\mathtt{mov}}$ *itag* $q \ q_s \ \_ = \overline{join} \ q \ q_s$
- r_{pc} contains p_s.

The first two conditions shows that an HLL join rule on top of the local continuation matches to the instruction at n_e. Recall that this is a pseudo-instruction that will invoke a RTL$^\mathrm{T}$ level join rule (cf §5.3). Consider *SkipJoin* (Fig. 3): in order to process join points uniformly, it abstracts over the rule $\overline{join}$. Without this abstraction, we would need an additional transition rule for each type of join point (there are three), in both HLL and RTL$^\mathrm{T}$ semantics. As shown in the `If-Then-Else` compilation diagram, split and join point instructions are generated at the same time; hence, it is easy to ensure they get corresponding split and join rule I-tags. During execution, however, arbitrary code runs between the split and join points; hence, the third condition (relating $\overline{\mathtt{mov}}$ and $\overline{join}$) insists that the `mov` carries the corresponding (and therefore correct) I-tag, i.e., that ($\overline{\mathtt{mov}}$ *itag*) invokes $\overline{join}$. The last condition, ensuring that $\overline{\mathtt{mov}}$ is actually invoked on the split point PC, is really an invariant—one whose maintenance is quite involved. We have to parameterize all compilation functions with stacks (to account for nesting of split/join points) of these "save" registers, and augment all matching relations with the invariant that none of the described computations trample these "save" registers.

To summarize matching over regular states:

Regular states $\mathcal{S}(F, s, k, p, c, E) \sim \mathcal{S}(F', n, p', c', B)$ if $\exists n_1$ s.t.
- $s \sim [n, n_1) \ \wedge \ k \sim n_1$
- $F \sim F' \ \wedge \ p \sim p' \ \wedge \ c \sim c' \ \wedge \ E \sim B$

As previously mentioned, we elide the details of these straightforward matchings

6 Optimizations

We have analyzed the CompCert RTL-improving passes Deadcode, CSE (common sub-expression elimination), and ConstProp to determine what information about policies is needed to adapt these optimizations to a tagged setting. All these passes have the effect of removing instructions, so the key concern is whether it is valid to skip the corresponding tag rule executions as well. As mentioned in §4, we have identified several simple and intuitive conditions on tag rules that, in various combinations, are sufficient to keep these optimizations sound. These conditions are dynamic, and not decidable in general at compile time, but there are simple conservative static approximations for each of them. We now consider the optimizations in turn, defining the conditions as they become relevant.

6.1 The Deadcode$^\mathrm{T}$ Pass

CompCert's Deadcode removes reachable but redundant instructions. In ordinary RTL, an instruction is dead if its destination register is dead, and a register is dead if it is not passed as an operand to any following instruction (before being re-defined). In RTL$^\mathrm{T}$, since instructions and their tag rules get their operands from the same registers, the standard notion of register liveness still holds, as whenever a value is passed to an instruction, its tags are passed to that instruction's tag rule.

However the standard notion of instruction deadness is *not* sufficient in RTL$^\mathrm{T}$, because even if an instruction's result value is not used, its tag rule might still fail-stop, change the PC tag, or change internal tag policy state. The latter two forms of state behave very similarly; since our prototype compiler does not support internal tag policy state, we consider only PC tag changes in the remainder of the paper.

In our adaptation Deadcode^T, we found that the conjunction of two conditions on rule evaluation were sufficient for a tag rule to be treated as dead:

- The rule never fail-stops.

  $$\mathcal{DFS}(\overline{rule}) \triangleq \forall \vec{x}.\ \overline{rule}\,\vec{x} \not\Rightarrow err$$

  This condition is in fact *necessary* to allow rule execution to be skipped: if there is a chance a rule might fail-stop, then skipping it might not preserve fail-stop behavior of the program.
- The rule outputs the same PC tag that it received, if it does not fail-stop, i.e. the rule exhibits "PC-purity".

  $$\mathcal{PCP}(\overline{rule}) \triangleq \forall p\ \vec{x}.\ \overline{rule}\,p\,\vec{x} \Rightarrow (p',\ \vec{x'}) \rightarrow p = p'$$

  Since the PC tag is threaded throughout the program's execution, it can effectively be used to pass state, which could affect a fail-stop decision in a later rule. $\mathcal{PCP}$ simply says that a rule is side-effect-free with respect to the PC tag.

$(\mathcal{DFS} \land \mathcal{PCP})$ is used as an additional guard to the standard notion of instruction deadness, both in the liveness analysis and the code transformation. Although these conditions are not statically computable, they have simple conservative approximations: A rule must be $\mathcal{DFS}$ if it never returns a tag error, and must be $\mathcal{PCP}$ if the output PC tag is always syntactically equal to the input PC tag.

6.2 Deadcode^T Theorem and Verification

Theorem 6.1. ***(Semantic Preservation Deadcode^T)*** *Let S and $\mathcal{P}$ be a RTL^T program and policy respectively, and C be the RTL^T result of performing the deadcode optimization on S. Under RTL^T semantics and the policy $\mathcal{P}$: If S does not* ***Go Wrong****, then the behavior displayed by C is displayed by S. Formally:* $\forall B.safe(S) \rightarrow C_{\mathcal{P}} \Downarrow B \rightarrow S_{\mathcal{P}} \Downarrow B$.

Having covered the generalities in §5.4, we discuss here how the analysis and proof of theorem 6.1 are driven by a set of *flags* that indicate which properties hold on the HLL rules. As optimizations work over RTL^T, it is the properties of RTL^T rules that we are interested in. Just as we define RTL^T policies out of HLL ones, we define RTL^T flags out of HLL ones and prove that whenever a RTL^T rule has a property, so does the corresponding HLL rule. e.g., $\mathcal{PCP}$ ($\overline{\texttt{mov}}$ `ITassign`) only if $\mathcal{PCP}$ ($\overline{:=}$).

In the correctness proof, we would like these flags to have type `Prop`, but in the compiler, we need them to be computable. So, we encode them as `option Props`. For example, `Some` (H:$\mathcal{DFS}$ ($\overline{\texttt{op}_\oplus}$)) tells the compiler that $\overline{\texttt{op}_\oplus}$ does not fail-stop and provides the `Prop` H:$\mathcal{DFS}$ ($\overline{\texttt{op}_\oplus}$) for use in the proof. The `None` case tells the compiler the property does not hold.

This dependent type does double duty: It helps us cleanly define RTL^T flags out of HLL ones while simultaneously verifing that if a RTL^T rule has property X, so does its related HLL rule. We define the RTL^T flags in Coq proof mode by providing a `Some` (requiring a proof of the property it carries) or `None` witness. Eliminating cases on HLL flags in order to provide such witnesses defines RTL^T flags from HLL ones. This approach helps us *validate* our policy compilation. As an example, if we wanted to show a witness for `Some` (H:$\mathcal{DFS}$ ($\overline{\texttt{op}_\oplus}$)), not having to case analyze the HLL flag (`option` $\mathcal{DFS}$ ($\overline{\oplus}$)) would be a hint that the definition of $\overline{\texttt{op}_\oplus}$ (via compilation of the HLL policy) is wrong. This validation mechanism caught several bugs in our initial compiler code.

There is a caveat attached to our verification. In Tagine's current implementation, we only model HLL policy signatures, not actual rule definitions. Therefore, the compiler cannot derive the HLL flag settings by inspecting the policy, but instead relies on external specification of the flag settings as axioms. As future work, we envision modelling the policy rule language in detail, so that properties of HLL policies can be extracted by a provably-correct static analysis.

6.3 CSE (Common-Subexpression Elimination)

This pass replaces repetitions of an $\texttt{op}_\oplus$ instruction (the common sub-expression) with a `mov` instruction that writes the previously computed value into the $\texttt{op}_\oplus$ destination register.

The variety of CSE implemented by CompCert is local value numbering (LVN). LVN works by maintaining a bijective mapping between symbolic identifiers (the value numbers) and expressions (i.e., variables or operations). It operates as a forward dataflow analysis over extended basic blocks. When encountering a (syntactic) expression, LVN checks the map to see if it already has a value number; if not, a fresh value number is assigned to the expression, and the map is updated accordingly. In standard LVN, an expression's value number is cleared if any of its constituent variables are redefined.

```
1: z = x + y          1: z = x + y
2: c = a + b          2: c = a + b
3: w = x + y          3: w = z
4: x = 5              4: x = 5
5: v = x + y          5: v = x + y
   pre-CSE               post-CSE
```

In the pre-CSE pseudocode, while lines 1, 3 and 5 contain a syntactically equivalent expression (`x + y`), only lines 1 and 3 have a *common* sub-expression, as they perform the same computation, while line 5 does not, due to the redefinition of `x` on line 4. LVN determines this by assigning the first repetition (line 3) the same value number as the original (line 1), because nothing causes it to be cleared from the map. However, the repetition on line 5 gets a new value number because `x` is redefined on line 4. LVN then replaces repeated sub-expressions whose value number is associated with a variable by an `mov` from that variable, as illustrated in the post-CSE pseudocode.

The standard notion of LVN guarantees that two expressions with the same value number are equivalent computations. In the Tagine setting, this is enough to guarantee that the rules of two expressions with the same value number will receive the same value tag inputs, but we still need to account for the PC tag input. The intuition is that LVN is sound in Tagine whenever $\overline{\texttt{op}_\oplus}$ is insensitive to the PC tag input or the repeated rules receive the same PC tag input as the original. We present two cases where this holds.

(a) When the `op`'s rule is (weakly) insensitive to the PC tag input, meaning that it is $\mathcal{PCP}$, and its PC tag input influences neither its output value tags nor whether it fail-stops.

$$\begin{aligned}
&\mathcal{WPCI}(\mathit{rule}) \triangleq \mathcal{PCP}(\mathit{rule}) \land (\forall p_1,\ p_2,\ \vec{v}.\\
&(\forall p_1'\ p_2'\ \vec{v_1'}\ \vec{v_2'}.\quad \overline{\mathit{rule}}\, p_1\ \vec{v} \Rightarrow (p_1', \vec{v_1'}) \land\\
&\qquad\qquad\qquad\qquad \overline{\mathit{rule}}\, p_2\ \vec{v} \Rightarrow (p_2', \vec{v_2'}) \to \vec{v_1'} = \vec{v_2'})\\
&\land\ \overline{\mathit{rule}}\, p_1\ \vec{v} \Rightarrow \mathit{err} \leftrightarrow \overline{\mathit{rule}}\, p_2 \vec{v} \Rightarrow \mathit{err})
\end{aligned}$$

The intuition for PC insensitivity is that a rule should "do nothing" with the PC tag, and in the case of weak PC insensitivity, propagating the input PC tag is the most innocuous choice for the output PC tag. In this case, the standard definition of LVN is already sound in Tagine.

(b) When a repeated `op`'s rule can be guaranteed to receive the same PC tag input as the original because all intervening instructions between the original sub-expression (including that sub-expression itself) and a candidate repetition are $\mathcal{PCP}$.

In summary, LVN in Tagine must modify standard LVN to clear out the value numbers of all non-$\mathcal{WPCI}$ instructions upon encountering a non-$\mathcal{PCP}$ instruction.

We have implemented this revised version of CSE in Tagine, but have not completed its verification.

6.4 ConstProp

ConstProp folds constants (concrete values known at compile time) by turning $\texttt{op}_\oplus$s whose results can be computed at compile time into `movi`s. It also performs constant propagation by a dataflow analysis over the contents of registers to compute their abstract values.

As a running example, consider a register bank with $r_1 \coloneqq 3, r_2 \coloneqq 4$. Standard constant folding makes the change:

$$\texttt{op}_+\ r_1\ r_2\ r_d \Rightarrow \texttt{movi}\ 7\ r_d$$

In Tagine we need to compute a tag to write into r_d as well. We outline two approaches to making folding sound in Tagine.

The first approach applies when the $\texttt{op}_\oplus$ to be folded has constant operand tags. This approach permits folding by (ultimately) invoking $\overline{\texttt{op}_\oplus}$ despite replacing the $\texttt{op}_\oplus$ with a `movi`. It does so with a special I-tag that can take parameters, e.g., $(\texttt{ITp} \oplus t_1\ t_2)$, where everything enclosed by the parentheses is one I-tag. In our running example, with $r_1 \coloneqq 3@c_1, r_2 \coloneqq 4@c_2$, folding makes this change:

$$\texttt{op}_+\ r_1\ r_2\ r_d\ @\ \texttt{IT+} \Rightarrow \texttt{movi}\ (7@\_)\ r_d\ @\ (\texttt{ITp}+\ c_1\ c_2)$$

$\overline{\texttt{movi}}$ is defined to invoke $\overline{\texttt{op}_+}$ on c_1 and c_2 when given I-tag $(\texttt{ITp}+\ c_1\ c_2)$, and r_d is tagged with the result.

The second approach applies when we can statically compute a concrete output value tag for $\overline{\texttt{op}_\oplus}$, which occurs in the following cases: **(1)** If $\overline{\texttt{op}_\oplus}$ is $\mathcal{PCP}$[3] and produces a constant value tag, implying insensitivity to all its inputs. In this case we do not require concrete inputs. **(2)** If $\overline{\texttt{op}_\oplus}$ is $\mathcal{PCP}$, its value tag output is simply propagated from one of its inputs, and that input is known at compile time. **(3)** If $\overline{\texttt{op}_\oplus}$ is strongly PC insensitive (i.e., the PC tag input does not influence the rule's output), the input value tags are known, and we can evaluate the rule at compile time. If the computed result is a fail-stop, the pass does not replace the $\texttt{op}_\oplus$, preserving fail-stop behavior.

Neither of these approaches is strictly more useful or applicable than the other. The first approach requires concrete tag values but can deal with dynamic PC tag input. This approach however, also generates more I-tags, which will cause more compulsory misses. The second approach does not always require concrete tag values, but is only applicable in rather ad hoc circumstances.

7 Coq Development

The proof of RTLgen^T is ~1700 lines, while Deadcode^T has a proof of ~650 lines. These numbers reflect formalization specific to the passes. From CompCert, we also used some proofs on the general metatheory of simulations.

The goal of keeping the components of Tagine as decoupled as possible led us to adopt a highly modularized and functorized architecture in Coq. In particular, IRs and compiler passes do not depend on semantics or proofs. However, to implement monitoring at RTL^T-level, Tagine must invent new RTL^T tags and policies. Moreover, while the abstract definition of RTL^T is independent of HLL, all Tagine-RTL^T notions (tags, language, policies, flags, semantics) must be parameterized by HLL ones. This means that optimization passes must be functors over Tagine-RTL^T and therefore parameterized by HLL tags, policies and flags as well. As examples, the proofs of RTLgen^T and Deadcode^T are functors over eleven and eight other modules, respectively.

8 Related Work

Hardware reference monitors and other secure hardware platforms have been the focus of much recent attention as a potential foundation for secure systems. Some target a specific security policy; for example, CHERI [23] implements compartmentalization using capabilities. PIPE instead aims

[3]In HLL, expression rules are designed to be implicitly $\mathcal{PCP}$: they do not even produce an output PC tag. Therefore RTL^T rules defined from HLL expression rules will always be $\mathcal{PCP}$.

to be general [7, 8]. It has been used to enforce information flow control [5], stack safety [19], and capability-based heap-safety, among other micropolicies [4]. Abate et. al. use PIPE as an example enforcement mechanism for their Secure Compartmentalized Compilation property [1].

Aspect-Oriented Programming bears a structural similarity to the reference monitor approach; when used for security it also entails interleaving policy validation with application code [13]. Advice points are akin to our control points. But AOP's advice code is normally written in the same language as the underlying program and can operate on the full program environment, which naturally suggests a semantics and implementation based on weaving together the program and advice. While parts of AOP semantics have been formalized [9, 22], we are not aware of any attempts to prove correctness of AOP tool implementations.

Like our work, much of the literature in compiler verification focuses on toy compilers that illustrate a key challenge [17], or verifies compilation of a specific, small part of a language [16]. The VLISP [12] project is notable in that it has a correctness proof for an implementation of LISP. But while rigorous, it is not machine checked. CompCert [15] and CakeML [14] stand alone as industrial strength, machine-checked, verified compilers; the former has been used to explore verifying optimizations, while CakeML has focused on reducing the trusted computing base, and verifies other parts of the run time, such as the garbage collector.

9 Conclusions and Future Work

We have demonstrated a plausible design for high-level tag-based monitoring and its compilation to PIPE-equipped hardware, formalized a prototype compiler, and verified that it preserves monitoring semantics. Although our formal development covers only a toy source language, it has allowed us to confirm the feasibility of the most novel aspects of the compilation approach.

There are numerous ways to extend this work to handle more realistic source languages and compilation mechanisms, in particular towards our goal of a fully verified compiler for a tagged version of C. Our first priority is to add addressable memory and pointers. In particular, we are interested in using policies to enforce memory safety and compartmentalization properties on top of a memory-unsafe but control-safe language called Concrete C, with the goal of giving security engineers a flexible tool for trading different levels of memory safety against performance. Adding support for this feature should be largely orthogonal to the existing Tagine development, although we will need to extend the optimization passes to handle pointer operations.

We also plan to extend HLL and RTLgen^{T} to handle the full set of C control flow operators. We expect this task to be straightforward, although we are still exploring whether useful IFC policies can be defined for this richer language.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. Greg Sullivan and Cătălin Hrițcu provided useful comments on earlier drafts. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-18-C-0011. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

References

[1] Carmine Abate, Arthur de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C Pierce, Marco Stronati, and Andrew Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1351–1368, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3243734.3243745, doi:10.1145/3243734.3243745.

[2] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Air Force Systems Command Electronic Systems Division, October 1972.

[3] Andrew W Appel and Sandrine Blazy. Separation logic for small-step Cminor. In *International Conference on Theorem Proving in Higher Order Logics*, pages 5–21. Springer, 2007.

[4] A. A. d. Amorim, M. Dénès, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy*, pages 813–830, May 2015. doi:10.1109/SP.2015.55.

[5] Arthur de Amorim, Nathan Collins, André DeHon, Delphine Demange, Cătălin Hrițcu, David Pichardie, Benjamin C Pierce, Randy Pollack, and Andrew Tolmach. A verified information-flow architecture. *SIGPLAN Not.*, 49(1):165–178, 1 2014. URL: http://doi.acm.org/10.1145/2578855.2535839, doi:10.1145/2578855.2535839.

[6] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 2002. doi:10.1145/359636.359712.

[7] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Jr., Benjamin C. Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 487–502, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2694344.2694383, doi:10.1145/2694344.2694383.

[8] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr., Benjamin C Pierce, and André DeHon. PUMP: A programmable unit for metadata processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '14, page 8:1–8:8, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2611765.2611773, doi:10.1145/2611765.2611773.

[9] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, REFLECTION '01, page 170–186, Berlin, Heidelberg, 2001. Springer-Verlag.

[10] Dover Microsystems. Coreguard overview. URL: https://www.dovermicrosystems.com/solutions/coreguard/.

[11] Draper Laboratories. Hope software toolchain. URL: https://github.com/draperlaboratory/hope-src/.

[12] Joshua D Guttman, John D Ramsdell, and Mitchell Wand. VLISP: A verified implementation of Scheme. *Lisp and Symbolic Computation*, 8(1-2):5–32, 1995.

[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP 97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997. doi:10.1007/BFb0053381.

[14] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2535838.2535841, doi:10.1145/2535838.2535841.

[15] Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009. URL: http://dx.doi.org/10.1007/s10817-009-9155-4, doi:10.1007/s10817-009-9155-4.

[16] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical aspects of computer science*, 1, 1967.

[17] Robin Milner and Richard Weyhrauch. Proving compiler correctness in a mechanized logic. *Machine Intelligence*, 7(3):51–70, 1972.

[18] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler enforced temporal safety for C. *SIGPLAN Not.*, 45(8):31–40, June 2010. URL: http://doi.acm.org/10.1145/1837855.1806657, doi:10.1145/1837855.1806657.

[19] Nick Roessler and Andre Dehon. Protecting the stack with metadata policies and tagged hardware. In *Proceedings - IEEE Symposium on Security and Privacy*, 2018. doi:10.1109/SP.2018.00066.

[20] A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003. URL: http://www.cs.cornell.edu/andru/papers/jsac/sm-jsac03.pdf, doi:10.1109/JSAC.2002.806121.

[21] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, February 2000. URL: http://doi.acm.org/10.1145/353323.353382, doi:10.1145/353323.353382.

[22] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, September 2004. URL: http://doi.acm.org/10.1145/1018203.1018208, doi:10.1145/1018203.1018208.

[23] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings - IEEE Symposium on Security and Privacy*, 2015. doi:10.1109/SP.2015.9.