Blame-Preserving Secure Compilation

Marco Patrignani
University of Trento

Matthias Kruse
CISPA Helmholtz Center for Information Security

1 Introduction

Let $C$ be a component (i.e., a partial program), $P$ be a (security) property, and $K$ be a program (often called program context) that $C$ can be linked against. We say that $C$ respects $P$ robustly if $C$ linked with any $K$ upholds $P$. Secure compilation can be stated as the robust preservation of a security property from a source component $C$ into its compiled counterpart $[C]$ [4]. That is: a compiler is secure for $P$, if for any source component $C$, if $C$ robustly preserves property $P$, then $[C]$ also robustly preserves $P$.

To allow the development of such secure compilers, it is often the case that the language targeted by the compiler must be enriched with security primitives [19]. A number of existing works define such target-level security primitives (e.g., coarse- [1, 18] and fine-grained [11, 12, 17, 20, 21] memory isolation, cryptographic constant-time [8], cryptography [2], types [9, 16], control-flow integrity [3] and more).

Sometimes, however, we may be interested in determining whether a compiler is secure but the problem does not lie in the target language (which we assume to be one such suitably-secure one), but in the source. Let us now consider $C$ as the source language, Rust as the target language, and a compiler between the two that aims at preserving temporal memory safety ($TMS$ [7]), as done by Nagarakatte et al. [15].

Unfortunately, in $C$, the proposition ‘any source component $C$ robustly preserves $TMS$’ is trivially false. This is because to uphold a property robustly, one must prove that a component $C$ has that property when interoperating with any larger program context $K$ (which is still a $C$ program in this case). Alas, this proposition is false, because in $C$ several of those larger program simply do pointer arithmetic and violate $TMS$ of any $C$. Thus, any compiler from $C$ to Rust can be proven to be secure according to the definition of secure compilation above: the premise of the implication is false!

One may be tempted to say that the problem lies with the definition of robustness, which forces us to consider any larger program $K$, and if we considered a subset of all $K$ we may be able to prove that a $C$ program satisfies $TMS$ almost-robustly. But we disagree. The strong security benefits of using such secure compilation statements come precisely from considering any $K$, i.e., any possible attacker to the program, not just a subset of them.

Then, we may be looking for an alternative criterion to indicate security of our compiler, and thus we may want to show that the compiler robustly enforces $TMS$. To prove a compiler enforces a property $P$ robustly, one must prove that any code produced by the compiler respects $P$ even when linked with any arbitrary target context $K$. Unfortunately, such a definition is also problematic. Consider the same C-to-Rust setting as before, the compiler that produces a random well-typed safe Rust program trivially respects it: any well-typed safe Rust program has $TMS$ by virtue of the Rust type system. Alas, such a compiler has completely messed up the behaviour of the original source component.

Thus, both with secure compilation stated as the robust preservation of a property, and with secure compilation stated as a robust enforcement property, we are left with an unsatisfactory statement about the security of our compiler.

In this paper we investigate a novel secure compilation criterion to be used in case one wants to prove that the compiled code has some property $P$ robustly, but the source language does not uphold $P$ robustly. We call this criterion Blame-Preserving Compilation (BPC), since it builds on the existing notion of blame [6, 22].

Blame is a notion that arises in the context of mixing typed and untyped programs and being able to show that if the execution ‘goes wrong’, then it went wrong in the untyped program, i.e., the untyped program must be blamed. In this work, we rely on a parallelism between secure compilation and blame work:

### Secure compilation has: Blame calculi have:

- component of interest $C$ ↔ typed programs
- program contexts $K$ ↔ untyped programs
- property to be upheld $P$ ↔ execution going wrong

So we leverage the notion of blame in order to relax the notion of robust property satisfaction into robust blame property satisfaction. A component (not a compiler) $C$ satisfies a property $P$ in the robust-blame fashion if either it robustly preserves $P$, or if the behaviour of $C$ when linked with an arbitrary program context $K$ violates $P$, then the violation of the property lies in $K$. With this intuition in mind, we informally say that a compiler attains Blame-Preserving Compilation if it preserves a property in the robust-blame fashion. That is:

- either the compiler robustly preserves property $P$;
- or, if the source execution violated $P$,
  - the violation of $P$ was done by the source program context $K$ (and not by the source component $C$),
  - and the compilation of $C$ still upholds $P$ robustly (i.e., against any target $K$) until the violation point.

With BPC, the only case where secure compilation cannot be applied is when a violation of $P$ happens in the source component being compiled. We believe it would be the duty of a (static) analysis tool to prevent that component from running. A compiler should not change the behaviour of
components through compilation, even if it means turning an insecure component into a secure one, lest source-level reasoning be lost, and source programmers are left confused.

1.1 Blame-Preserving Compilation

We now give the formal preliminaries that lead to the definition of BPC (Definition 1.1).

In the following we write \(K[C]\) to indicate the whole program resulting from the linking of component \(C\) with program context \(K\). Then, we write \(K[C] \rightsquigarrow \overline{a}\) to indicate that running the whole program \(K[C]\) according to the language semantics generates trace \(\overline{a}\). We assume both source and target languages of the compiler have the same trace model, as commonly done in compilation work [4], and leave lifting this limitation for future work [5]. Since many security properties (namely all safety and all hypersafety properties such as non-interference [10]) can be stated with finite traces only, we consider a trace model where traces \(\overline{a}\) are finite sequences of actions \(\alpha\) (the empty trace being \(\varnothing\)). In common trace jargon, all traces in our model really are prefixes, so we focus on properties that belong to the safety class (though the hypersafety class could also work). Additionally, we assume that any action that appears on a trace can be unequivocally identified as being done by either the program context or by the component; this is a feature that is common in all secure compilation works using traces [3, 11, 12, 17, 18].

We indicate an action \(\alpha\) done by the program context as \(\alpha \in K\) and an action \(\alpha\) done by the component as \(\alpha \in C\). Finally, we write \(\text{blame}(\overline{a}, P) = \overline{a}_1 | \overline{a}_2\) to indicate that trace \(\overline{a}\) can be split into two traces \(\overline{a}_1\) and \(\overline{a}_2\) such that \(\overline{a}_1\) upholds \(P\) and the violating action (at the beginning of \(\overline{a}_2\)) is done by \(K\).

Formally:

\[
\begin{align*}
\text{blame}(\varnothing, P) &= \varnothing | \varnothing \\
\text{blame}(\overline{a}\alpha, P) &= \overline{a}_1 | \overline{a}_2\alpha & \text{if } \text{blame}(\overline{a}, P) = \overline{a}_1 | \overline{a}_2 \\
& & \text{and } \overline{a}_2 \neq \varnothing \\
\text{blame}(\overline{a}\alpha, P) &= \overline{a}_1 | \alpha & \text{if } \text{blame}(\overline{a}, P) = \overline{a}_1 | \varnothing \\
& & \text{and } \alpha \in K \text{ and } \overline{a}_2\alpha \notin P \\
\text{blame}(\overline{a}\alpha, P) &= \overline{a}_1\alpha | \varnothing & \text{if } \text{blame}(\overline{a}, P) = \overline{a}_1 | \varnothing \\
& & \text{and } \overline{a}_2\alpha \in P
\end{align*}
\]

We now have all the ingredients to formally define when a compiler attains Blame-Preserving Compilation of \(P\) (denoted as \(\vdash [] : \text{BPC}(P)\)).

Definition 1.1 (Blame-Preserving Compilation).

\[
\vdash [] : \text{BPC} \overset{\text{def}}{=} \forall P \in \text{Safety. } \forall C.
\]

\[
\begin{align*}
&\text{if } \forall K. \forall \overline{a}, \overline{a}_1, \overline{a}_2, \text{ if } K[C] \rightsquigarrow \overline{a}\text{ then blame}(\overline{a}, P) = \overline{a}_1 | \overline{a}_2 \\
&\text{then } \forall K. \forall \overline{a}, \overline{a}_1, \overline{a}_2, \text{ if } K[[C]] \rightsquigarrow \overline{a}_1 \\
&\text{then blame}(\overline{a}, P) = \overline{a}_1 | \overline{a}_2
\end{align*}
\]

When looking at the premise of BPC, note that in case \(\overline{a}\) does not violate \(P\), \(\text{blame}(\overline{a}, P)\) returns \(\overline{a} | \varnothing\). So, for traces where a program robustly satisfies \(P\), BPC is equivalent to the robust preservation of \(P\).

The conclusion of BPC is equivalent to the secure compilation criterion for the preservation of any safety property, since the target code must emit the \(\overline{a}_1\) trace. This ensures that we can compose any BPC compiler with other secure compilers that preserve safety properties and still prove security of the composed compiler [13].

1.2 Proving BPC

Proving that a compiler attains BPC in the sense of Definition 1.1 may be overly complicated, as it requires reasoning about arbitrary target contexts \(K\). For this, modern secure compilation criteria often come with an equivalent statement that is simpler to prove. We do not have one such criterion, but we know what “shape” it should have, so we conjecture it below (and we indicate it as \(\vdash [] : \text{PF-BPC}\)).

Definition 1.2 (Sketch: Desired Criterion).

\[
\vdash [] : \text{PF-BPC} \overset{\text{def}}{=} \forall C. \forall K. \forall \overline{a}_1, \overline{a}_2.
\]

\[
\begin{align*}
&\text{if } K[[C]] \rightsquigarrow \overline{a}_1 \text{ and } \overline{a}_1 \sim \overline{a}_2 \\
&\text{then } \exists K. K[C] \rightsquigarrow \overline{a}_2
\end{align*}
\]

We want a criterion with this kind of shape in order to reuse existing proof techniques called trace-based backtransformation that let us build the existentially-quantified source context \(K\) from the target trace \(\overline{a}_1\) (and \(\overline{a}_2\) in this case).

Ideally, we need to find a trace relation \(\sim\) (or any property linking the two traces) that lets us prove at least that \(\vdash [] : \text{PF-BPC}\) implies \(\vdash [] : \text{BPC}\).

1.3 Applying BPC

We believe BPC would be best applied to settings with a weak source language, such as C. Recently, Michael et al. [14] have developed a compiler for C to MSWasm, an extension of WebAssembly with Cheri-like capabilities. MSWasm is proven to have language-level robust spatial and temporal memory safety (MS). Alas, the compiler of Michael et al. [14] is only proven correct, and its correctness entails some form of security, but not in the presence of any active adversary.

We believe we can phrase BPC for memory safety (which can be phrased as a safety property) and prove that the compiler from C to MSWasm upholds BPC, giving a stronger and more precise characterisation of the security guarantees provided by that compiler.

Similarly, we can apply BPC to reason about a compiler from Rust to MSWasm, where the source component being compiled is safe, but it may link against unsafe Rust. Proving the compiler attains BPC for temporal memory safety would let us prove that the compiled Rust code is protected from MSWasm-level attackers, and violations of TMS always belong to the attacker context.
Acknowledgements: This work was partially supported by: the Italian Ministry of Education through funding for the Rita Levi Montalcini grant (call of 2019); the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

References


