Blame-Preserving Secure Compilation

Marco Patrignani
University of Trento

Matthis Kruse
CISPA Helmholz 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 $[\![\mathsf{C}]\!]$ [4]. That is: a compiler is secure for P, if for any source component C, if C robustly preserves property P, then $[\![\mathsf{C}]\!]$ 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	$\leftrightarrow$	typed programs
program contexts K	$\leftrightarrow$	untyped programs
property to be upheld P	$\leftrightarrow$	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 $K[C] \rightsquigarrow \overline{\alpha}$ to indicate that running the whole program $K[C]$ according to the language semantics generates trace $\overline{\alpha}$. 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 $\overline{\alpha}$ are finite sequences of actions α (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 α done by the program context as $\alpha \in K$ and an action α done by the component as $\alpha \in C$. Finally, we write $\texttt{blame}(\overline{\alpha}, P) = \overline{\alpha}_1 \mid \overline{\alpha}_2$ to indicate that trace $\overline{\alpha}$ can be split into two traces $\overline{\alpha}_1$ and $\overline{\alpha}_2$ such that $\overline{\alpha}_1$ upholds P and the violating action (at the beginning of $\overline{\alpha}_2$) is done by K. Formally:

$$
\begin{aligned}
\texttt{blame}(\varnothing, P) &= \varnothing \mid \varnothing \\
\texttt{blame}(\overline{\alpha}\alpha, P) &= \overline{\alpha}_1 \mid \overline{\alpha}_2\alpha && \text{if } \texttt{blame}(\overline{\alpha}, P) = \overline{\alpha}_1 \mid \overline{\alpha}_2 \\
& && \text{and } \overline{\alpha}_2 \neq \varnothing \\
\texttt{blame}(\overline{\alpha}\alpha, P) &= \overline{\alpha}_1 \mid \alpha && \text{if } \texttt{blame}(\overline{\alpha}, P) = \overline{\alpha}_1 \mid \varnothing \\
& && \text{and } \alpha \in K \text{ and } \overline{\alpha}_1\alpha \notin P \\
\texttt{blame}(\overline{\alpha}\alpha, P) &= \overline{\alpha}_1\alpha \mid \varnothing && \text{if } \texttt{blame}(\overline{\alpha}, P) = \overline{\alpha}_1 \mid \varnothing \\
& && \text{and } \overline{\alpha}_1\alpha \in P
\end{aligned}
$$

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

Definition 1.1 (Blame-Preserving Compilation)**.**

$$
\begin{aligned}
\vdash [\![\cdot]\!] : \mathsf{BPC} \stackrel{\text{def}}{=} \forall P \in \mathit{Safety}.\ \forall \mathsf{C}.\\
\text{if } \forall \mathsf{K}.\ \forall \overline{\alpha}, \overline{\alpha}_1, \overline{\alpha}_2.\ \text{if } \mathsf{K}[\mathsf{C}] \rightsquigarrow \overline{\alpha}\\
\text{then } \texttt{blame}(\overline{\alpha}, P) = \overline{\alpha}_1 \mid \overline{\alpha}_2\\
\text{then } \forall K.\ \forall \overline{\alpha}, \overline{\alpha}_1, \overline{\alpha}_2.\ \text{if } K[[\![\mathsf{C}]\!]] \rightsquigarrow \overline{\alpha}_1\\
\text{then } \texttt{blame}(\overline{\alpha}, P) = \overline{\alpha}_1 \mid \overline{\alpha}_2
\end{aligned}
$$

When looking at the premise of BPC, note that in case $\overline{\alpha}$ does not violate P, $\texttt{blame}(\overline{\alpha}, P)$ returns $\overline{\alpha} \mid \varnothing$. 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 $\overline{\alpha}_1$ 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 $\vdash [\![\cdot]\!] : \mathsf{PF\text{-}BPC}$).

Definition 1.2 (Sketch: Desired Criterion)**.**

$$
\begin{aligned}
\vdash [\![\cdot]\!] : \mathsf{PF\text{-}BPC} \stackrel{\text{def}}{=} \forall \mathsf{C}.\ \forall K.\ \forall \overline{\alpha}_1, \overline{\alpha}_2.\\
\text{if } K[[\![\mathsf{C}]\!]] \rightsquigarrow \overline{\alpha}_1 \text{ and } \overline{\alpha}_1 \sim \overline{\alpha}_2\\
\text{then } \exists \mathsf{K}.\ \mathsf{K}[\mathsf{C}] \rightsquigarrow \overline{\alpha}_2
\end{aligned}
$$

We want a criterion with this kind of shape in order to reuse existing proof techniques called trace-based backtranslation that let us build the existentially-quantified source context K from the target trace $\overline{\alpha}_1$ (and $\overline{\alpha}_2$ 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 $\vdash [\![\cdot]\!] : \mathsf{PF\text{-}BPC}$ implies $\vdash [\![\cdot]\!] : \mathsf{BPC}$.

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

[1] Martín Abadi and Gordon D. Plotkin. On protection by layout randomization. *ACM Transactions on Information and System Security*, 15: 8:1–8:29, July 2012. ISSN 1094-9224.

[2] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *IEEE Symposium on Logic in Computer Science*, pages 105–116, 1998.

[3] Carmine Abate, Arthur Azevedo 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. CCS '18, 2018.

[4] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *2019 IEEE 32th Computer Security Foundations Symposium*, CSF 2019, June 2019.

[5] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. An extended account of trace-relating compiler correctness and secure compilation. *ACM Trans. Program. Lang. Syst.*, 43 (4), nov 2021. ISSN 0164-0925. doi: 10.1145/3460860. URL https://doi.org/10.1145/3460860.

[6] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. *SIGPLAN Not.*, 46(1):201–214, jan 2011. ISSN 0362-1340. doi: 10.1145/1925844.1926409. URL https://doi.org/10.1145/1925844.1926409.

[7] Arthur Azevedo de Amorim, Cătălin Hriţcu, and Benjamin C. Pierce. The meaning of memory safety. In Lujo Bauer and Ralf Küsters, editors, *Principles of Security and Trust*, pages 79–105, Cham, 2018. Springer International Publishing. ISBN 978-3-319-89722-6.

[8] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343, 2018. doi: 10.1109/CSF.2018.00031.

[9] William J. Bowman and Amal Ahmed. Noninterference for free. *SIGPLAN Not.*, 50(9):101–113, aug 2015. ISSN 0362-1340. doi: 10.1145/2858949.2784733. URL https://doi.org/10.1145/2858949.2784733.

[10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. doi: 10.3233/JCS-2009-0393. URL https://www.cs.cornell.edu/~clarkson/papers/clarkson_hyperproperties_journal.pdf.

[11] Akram El-Korashy, Stelios Tsampas, Marco Patrignani, Dominique Devriese, Deepak Garg, and Frank Piessens. Capableptrs: Securely compiling partial programs using the pointers-as-capabilities principle. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*, pages 1–16, 2021. doi: 10.1109/CSF51468.2021.00036. URL https://doi.org/10.1109/CSF51468.2021.00036.

[12] Akram El-Korashy, Roberto Blanco, Jérémy Thibault, Adrien Durier, Deepak Garg, and Catalin Hritcu. Secureptrs: Proving secure compilation with data-flow back-translation and turn-taking simulation, 2022.

[13] Matthis Kruse and Marco Patrignani. Composing secure compilers. January 2022. in ACM SIGPLAN Workshop on Principles of Secure Compilation (Prisc'22).

[14] Alexandra Michael, Anitha Gollamudi, Jay Bosamiya, Craig Disselkoen, Aidan Denlinger, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. Mswasm: Soundly enforcing memory-safe execution of unsafe code. *Proc. ACM Program. Lang.*, (POPL), jan 2023.

[15] 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. ISSN 0362-1340. doi: 10.1145/1837855.1806657. URL http://doi.acm.org/10.1145/1837855.1806657.

[16] Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 103–116. ACM, 2016. doi: 10.1145/2951913.2951941. URL https://doi.org/10.1145/2951913.2951941.

[17] Marco Patrignani and Deepak Garg. Robustly safe compilation, an efficient form of secure compilation. *ACM Trans. Program. Lang. Syst.*, 43(1), feb 2021. ISSN 0164-0925. doi: 10.1145/3436809. URL https://doi.org/10.1145/3436809.

[18] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. *ACM Trans. Program. Lang. Syst.*, 37:6:1–6:50, April 2015. ISSN 0164-0925.

[19] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation a survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6):125:1–125:36, January 2019.

[20] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Stktokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *J. Funct. Program.*, 31:e9, 2021. doi: 10.1017/S095679682100006X. URL https://doi.org/10.1017/S095679682100006X.

[21] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. Linear capabilities for fully abstract compilation of separation-logic-verified code. *J. Funct. Program.*, 31:e6, 2021. doi: 10.1017/S0956796821000022. URL https://doi.org/10.1017/S0956796821000022.

[22] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, page 1–16, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 9783642005893. doi: 10.1007/978-3-642-00590-9_1. URL https://doi.org/10.1007/978-3-642-00590-9_1.