CBMC: Verificación Formal en Software Crítico y Aplicación en Criptoanálisis

Juan Bratti, Emanuel Nicolás Herrador e Ignacio Scavuzzo

Facultad de Matemática, Astronomía, Física y Computación, Universidad Nacional de Córdoba, Argentina
juanbratti@mi.unc.edu.ar, emanuel.nicolas.herrador@unc.edu.ar, iscavuzzo@mi.unc.edu.ar

Abstract CBMC (C Bounded Model Checker) is a formal verification tool for detecting bugs in C and C++ programs using bounded model checking. This report explores its internal mechanisms, verification features, and command-line usage. Through a case study from a Capture The Flag (CTF) cryptographic challenge, we demonstrate how CBMC can model and break a custom XOR cipher. We also compare it with tools like CPA-Seq and ESBMC-incr, and discuss some of its limitations. Our main contribution is the integration of CBMC's symbolic analysis with practical cryptographic vulnerability modeling.

Keywords: CBMC · Formal Verification · Bounded Model Checking · Symbolic Execution · Cryptographic Analysis · SAT/SMT Solving

Resumen CBMC (C Bounded Model Checker) es una herramienta de verificación formal para detectar errores en programas C y C++ mediante técnicas de verificación acotada. Este informe analiza sus mecanismos internos, capacidades de verificación y uso por línea de comandos. A través de un estudio de caso basado en un desafío criptográfico de una competencia CTF, mostramos cómo CBMC puede modelar y romper un cifrado XOR personalizado. También lo comparamos con herramientas como CPA-Seq y ESBMC-incr, y describimos algunas de sus limitaciones. Nuestra principal contribución es la integración del análisis simbólico de CBMC con la modelización práctica de vulnerabilidades criptográficas.

1. Introducción

En el desarrollo de software crítico, especialmente en áreas como la industria aeroespacial, automotriz o médica donde la confiabilidad del software es muy importante, es una prioridad garantizar la corrección funcional del código. Programas escritos en lenguajes como C o C++ pueden contener en sí mismos errores que pueden producir desde fallos de seguridad hasta pérdidas no solo económicas, sino también humanas. En este contexto, las técnicas tradicionales de prueba resultan poco eficientes o costosas para ofrecer una garantía formal sobre el comportamiento de un programa.

La verificación formal de software surge como una respuesta a este problema, permitiendo demostrar mediante métodos formales y matemáticos, que un sistema cumple con especificaciones previamente definidas. Entre las herramientas que implementan este tipo de técnicas se encuentra CBMC. Este informe tiene como objetivo presentar el funcionamiento y alcance de esta herramienta, buscando ilustrar su uso mediante un caso de estudio concreto y su comparación con otras herramientas de verificación formal. La intención es poder destacar la utilidad práctica de la herramienta y su rol a la hora de asegurar corrección y consistencia en el desarrollo de software.

Organización del Contenido La sección 2 introduce la herramienta junto con su contexto histórico. La sección 3 describe su uso desde la línea de comandos y la sección 4 detalla su funcionamiento interno. La sección 5 presenta aplicaciones relevantes en distintos dominios y un caso de estudio específico de criptoanálisis. Luego, en la sección 6 se detallan comparaciones con otras herramientas similares y en la sección 7 se discuten las limitaciones encontradas sobre CBMC. Finalmente, la sección 8 concluye el informe y resume los puntos más importantes tratados.

2. Fundamentos y Origen

CBMC es un verificador de modelos acotados open source[1] para programas en C y C++ que soporta C89, C99, C11, C17 y gran parte de las extensiones de compilador provistas por gcc, clang y Visual Studio. Este forma parte de la familia de herramientas CPROVER destinadas al análisis y verificación automática de software [12]. Fue desarrollada originalmente en el ámbito académico, específicamente en Carnegie Mellon University por Edmund Clarke, Daniel Kroening y Flavio Lerda; y presentada por primera vez en la conferencia TACAS 2004, con el título "A Tool for Checking ANSI-C Programs," publicado bajo la editorial Springer [5]. Desde entonces, tiene una evolución activa y actualmente es mantenida por D. Kroening quien trabaja en colaboración con la Universidad de Oxford [12].

Esta herramienta verifica la seguridad de la memoria (como límites de arreglos y buen uso de punteros), varias variantes de comportamiento indefinido y aserciones especificadas por el usuario. Más aún, puede corroborar equivalencia de entrada y salida de un programa respecto a otros lenguajes como Verilog [11].

Implementa una técnica llamada Bounded Model Checking (BMC) donde la relación de transición de una máquina de estados compleja y su especificación se desarrollan conjuntamente para obtener una fórmula booleana que es satisfactible si existe una traza de error. La fórmula es, luego, verificada usando un procedimiento SAT y si es satisfactible, entonces se presenta el contraejemplo. CBMC corrobora que se haya realizado suficiente desarrollo para asegurar que ya no pueda existir ningún contraejemplo mediante aserciones desarrolladas (unwinding assertions) [5]. Por defecto, incorpora un solucionador integrado basado

[1] CBMC. Disponible en `https://github.com/diffblue/cbmc`

en MiniSat, pero también tiene soporte para SMT externas como Boolector, CVC5 y Z3 [11].

Si bien CBMC es una herramienta que surgió en el ámbito académico, es ampliamente utilizada en la industria, especialmente en contextos donde la confiabilidad del software es crítica [12]. Entre algunas de las aplicaciones de la herramienta, se encuentran: localización y explicación de fallas en programas en C [15], verificación de programas concurrentes [18] y aplicaciones en la seguridad de programas para verificar ausencia de errores como accesos de memoria inválidos y overflows[2]. Además, CBMC tiene múltiples aplicaciones en Criptología, las cuales serán detalladas en secciones futuras. Esta herramienta es especialmente útil en etapas de desarrollo; para encontrar bugs y corroborar propiedades; y por supuesto, en etapas de testing para generación de entrada para tests y detección de vulnerabilidades de seguridad [12].

Dado que CBMC está acotado a C y C++, existen variantes para soportar otros tipos de lenguajes tales como JBMC para Java, Kani para Rust y EBMC para lenguajes de especificación de hardware. La motivación detrás de una herramienta que funcione en un lenguaje como C o C++, se basa en el hecho de que muchos de los *safety-critical software* están escritos en lenguajes de bajo nivel, a veces por cuestiones de mejor desempeño. A diferencia de la verificación de código de alto nivel, la de bajo nivel puede ser más compleja por el uso de punteros u operaciones *bit-wise*, por ejemplo [5]. Se destacan dos objetivos o criterios principales para la creación de la herramienta: garantía de calidad, refiriéndose no a la "correctitud total" sino a la ausencia de fallas especificas; y la automatización, no solo para proveer de simpleza para verificar un programa sino también para ajustar o asegurar la verificación en entornos donde se requiere de entrega rápida de software [5].

3. Interfaz de usuario

La herramienta provee de una interfaz basada en la línea de comandos. Si bien existe la posibilidad de usar Eclipse o VS Code como IDE, en esta sección nos enfocaremos en el uso mediante terminal, ya que permite acceder a la totalidad de las funciones y configuraciones posibles [23]. La sintaxis del comando general para utilizar CBMC es:

```
$ cbmc [options ...] [file.c ...]
```

Entre las opciones más relevantes de opciones se encuentra `--help` para mostrar la lista completa de comandos, y `--function f_name`, que permite definir manualmente la función de entrada para el análisis (en lugar de utilizar una función main por defecto). Esto es muy útil cuando estamos trabajando con módulos individuales o cuando queremos hacer verificaciones de manera parcial sobre un conjunto de módulos.

[2] Desarrollo de ML-KEM como ejemplo. Disponible en `https://github.com/pq-code-package/mlkem-native`

Además, se tienen opciones para ajustar el nivel de información devuelto por terminal. Se usa `--trace` para observar las trazas de ejecución generadas ante violaciones de propiedades; `--verbosity` para cambiar el nivel de detalle incluido en la salida; y `--dimacs` para imprimir la fórmula CNF que se pasa al SAT Solver.

Sumado a lo anterior, también permite ajustar las propiedades específicas a verificar. Para ignorar la verificación de errores tales como buffer overflows, divisiones por cero o desreferencias, están las opciones `--no-bounds-check`, `--no-div-by-zero-check`, `--no-pointer-check` respectivamente. Para ignorar aserciones presentes en el código se usa `--no-assertions-`. En cuanto aspectos más técnicos, se encuentran además `--unwind` y `--unwindset` para establecer configuraciones para el tratamiento de bucles presentes en el código, como la elección de la cota para el desenrollo a realizar. [7]

Además de poder ajustar las propiedades por defecto, una característica importante de la herramienta es la elección de propiedades a verificar. Por un lado, puede verificarse la existencia de problemas comunes como index out of bounds, buffer overflows, memory leaks [12], y por otro lado, el usuario puede definir dentro del código propiedades específicas que desea que se verifiquen. Éstas las llamamos aserciones (en C, generalmente se usa el macro `assert(condition)`). Otra manera de definir aserciones con mensajes específicos de error es usando `__CPROVER_assert(condition, error_message)`. Notar que en ambos casos CBMC verifica las aserciones para cualquier dirección posible en la ejecución. Asimismo, permite acotar el análisis simbólico mediante el uso de `__CPROVER_assume(condition)`, que restringe el espacio de búsqueda a aquellos caminos en los que se cumpla la condición indicada. Este último resulta útil para modelar precondiciones o reducir el dominio de los valores no determinísticos que puede tomar una variable [12].

4. Funcionamiento Interno y Aspectos Técnicos

Para explicar el proceso, consideraremos el ejemplo de la figura 1.

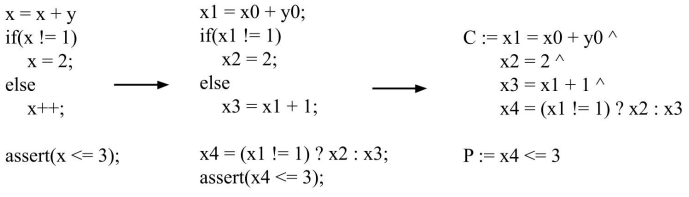

Figura 1. Programa ejemplo transformado a SSA y fórmula

Como primer paso se transforma el programa en un grafo de control de flujo (CFG) [3], la cual es una representación de todos los caminos que pueden ser atravesados por un programa durante su ejecución [10]. En nuestro caso ejemplo se obtendría el grafo de la figura 2. Luego, para corroborar las aserciones o propiedades (como respetar límites de un arreglo), se siguen los caminos del CFG hacia una aserción y luego se construye la fórmula correspondiente [10].

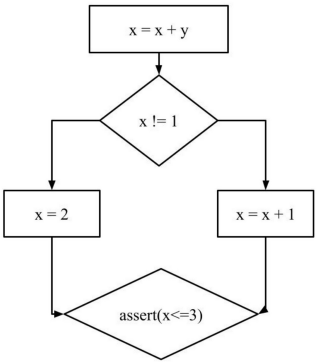

Figura 2. Generación de la CFG

Dado un camino, los ciclos, si los hay, son desenrollados duplicando el cuerpo n veces (para n fijo). Cada copia tiene un condicional que usa la misma condición que el ciclo por si requiere menos de n iteraciones. Luego de la última copia, se agrega una aserción que asegura que el programa nunca requiere más iteraciones. Esta aserción usa la negación de la condición del ciclo (la llamamos aserción desenrollada) y nos permite saber si la cota es suficientemente larga para cubrir todos los ciclos. Un ejemplo del desenrollo realizado puede verse en la figura 3. Respecto a los goto y llamadas a funciones, estos se desenrollan de una forma similar [5].

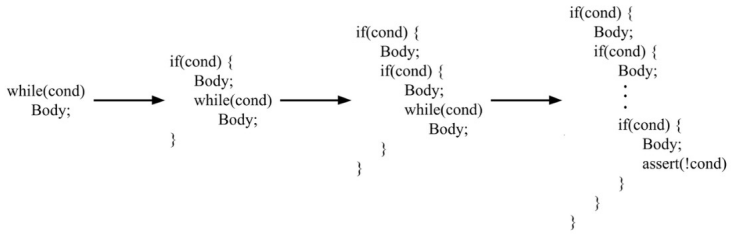

Figura 3. Ejemplo del desenrollo de loops realizado

Posteriormente, este programa es transformado en una forma SSA (static single-assignment) de modo que cada variable sea asignada exactamente una vez, y para que los punteros se simulen como estados abstractos en función de las asignaciones que se le realicen (son tratados como funciones no interpretadas). Se reemplazan las variables originales y construcciones por equivalentes con el mismo efecto pero que facilitarán la verificación [10]. Finalmente, gracias a ello se producen dos ecuaciones de vectores de bits: C para las restricciones y P para la propiedad. Para corroborar esta propiedad, entonces, la fórmula buscada es $C \wedge \neg P$.

[3] Se obtiene luego de las etapas de scanning y parsing usando la gramática de C/C++ y desambiguando las expresiones (igual al funcionamiento inicial de un intérprete o compilador).

Una vez construida esta fórmula, se transforma en CNF usando el método de Tseytin y se pasa a un procedimiento de decisión (SAT/SMT) para obtener una asignación satisfactible que la verifique (es decir, que sea la traza de error). El procedimiento de decisión que se requiere debe poder manejar lógica de vectores de bits, arreglos, listas, conjuntos y diccionarios [10]. Además, se sugiere usar un solucionador incremental SAT para reducir la complejidad de la fórmula a verificar, de modo que se añadan primero las partes fáciles de la fórmula y las complejas solo cuando sea necesario.

5. Aplicaciones y Casos de Estudio

CBMC es herramienta versátil y que provee una funcionalidad de verificación que permite ser utilizada tanto en el sector académico como en el industrial, e incluso en el competitivo (como se podrá ver a continuación en nuestro caso de estudio presentado). Además, el sector o área de estudio en el que se utiliza es de lo más diverso incluyendo sectores como verificación de programas concurrentes [18], hardware [8], sistemas operativos [4], implementación de drivers [16] e incluso de librerías de criptografía [9,6]; corroboración de equivalencia entre programas (o una simulación con un modelo) [21,19], análisis de sistemas físicos y de control [20], generación automática de vectores de prueba [14,1], cálculo del tiempo de ejecución en el peor caso (WCET) [2], entre otros. Otro sector en el que fue aplicado, más ajeno a la computación, es en biología para el estudio de restricciones del tráfico de vesículas en células eucariotas [3].

Como caso de estudio elegido para mostrar y reflejar el potencial y la utilidad de la herramienta, hemos decidido romper un cifrado customizado que consta de una sucesiva aplicación de una encriptación XOR con clave repetida (variante del cifrado de Vernam [22]) como parte del reto "XtraORdinary" [4] propuesto en la competencia Capture The Flag "picoMini" en el año 2021.

Debido a que la importancia de esta sección reside en la utilización de CBMC, los aspectos técnicos y reducciones realizadas como parte del reto no se presentarán; sino que directamente se detallará la solución sobre la versión ya simplificada. La versión completa junto con los archivos tanto del reto como de la solución se encuentra en el repositorio de este informe [5].

En la versión simplificada, el objetivo es quebrar un cifrado obteniendo el mensaje original que corresponde con el mensaje encriptado interceptado (que de ahora en más llamaremos flag) de longitud 76 (en formato hexadecimal):

```
57657535570c1e1c612b3468106a18492140662d2f5-
967442a2960684d28017931617b1f3637
```

El algoritmo de cifrado aplica sucesivamente la encriptación XOR con clave repetida con 6 palabras donde la primera a aplicar es una clave desconocida

[4] "XtraORdinary". Disponible en `https://play.picoctf.org/practice/challenge/208`. Accedido el 21 de mayo de 2025.

[5] CBMC Analysis Report. Disponible en `https://github.com/helcsnewsxd/cbmc-analysis-report`

(privada) mientras que las restantes 5 son claves conocidas (públicas). La encriptación con la clave privada siempre se realiza, pero con las públicas depende de un condicional sujeto a una condición aleatoria (es decir, existen 2^5 opciones de esquemas de cifrados posibles).

En base a ello, el uso de CBMC consta en la implementación del modelo de cifrado descripto con valores desconocidos para la flag (de longitud conocida igual a la mitad del mensaje interceptado: 38), la clave (de longitud desconocida) y los 5 condicionales que corresponden a la aplicación o no del encriptado para cada clave pública. En este modelo se hace uso de la búsqueda de trazas de error generada por CBMC al colocar como aserción final que la bandera encriptada en el cifrado es distinta a la interceptada. De este modo, esta traza muestra los estados correspondientes al par `(flag, key)` que generan el mismo valor interceptado.

Para reducir la búsqueda de posibles entradas que hagan fallar la aserción, se hace uso de `__CPROVER_assume()`. Esto permite asegurar propiedades que tanto el mensaje original como la clave privada deben cumplir. En particular, algunas de estas condiciones implican que ambas contienen solo caracteres imprimibles y que el formato de la bandera es `picoCTF{...}`.

Finalmente, y debido a que el análisis devuelve una sola traza de error, para hacer más eficiente la búsqueda se desarrolló un script que ejecute el análisis por parte de esta herramienta para cada posible tamaño de la clave privada (entre 1 y 38), de modo que podamos obtener todos los pares de mensaje y clave originales posibles. Gracias a esto, en 6,45 segundos [6], se pudo obtener el mensaje original y con ello romper este cifrado. La bandera original es `picoCTF{w41t_s0_1_d1dnt_1nv3nt_x0r???}`.

6. Comparación con Otras Herramientas

La comparación estará basada en el trabajo realizado por Dirk Beyer y Thomas Lemberger [13]. En principio, se compara a CBMC junto con herramientas de testing tradicional (generadores de tests de código automáticos, por ejemplo) y con otras herramientas de verificación formal (*model checkers*). Esta comparación no es trivial, ya que el testing tradicional es una forma de testing muy usada en muchos lenguajes, aunque con algunas desventajas en C. Si bien en los trabajos se menciona que el rendimiento de verificadores formales como CBMC, es mucho mejor en comparación al testeo tradicional; los verificadores entre sí tienen diversas mejoras que hacen que uno sea mejor que otro.

Comenzando por el testing tradicional, el mismo brinda una forma más accesible de testeo para desarrolladores (a veces los verificadores formales pueden parecer complejos si uno no tiene conocimiento en lógica formal). También, permiten testear características específicas de nuestro programa manualmente. En contraste, el testing tradicional no puede garantizar ausencia de errores (pero

[6] Computadora con Sistema Operativo Windows 11 Pro 24H2, 16GB de RAM y procesador AMD Ryzen 7 5700G

sí la presencia). Además, debido a la falta de estandarización para C, la generación de tests automáticos es muy difícil de llevar a cabo [13], por lo que no son fácilmente escalables para programas complejos. Por ambos motivos, aquí CBMC se posiciona como una mejor opción al poder garantizar ausencia de errores (respecto a una cota/límite) y debido a que es escalable a programas complejos.

Respecto a la comparación de CBMC con otros verificadores formales, se destacan que herramientas como CPA-Seq y ESBMC-incr tienen ventajas ante esta. CPA-Seq combina técnicas más avanzadas que le permiten ser mucho más profundo y rápido a la hora de explorar los caminos posibles de un programa. Según la experimentación realizada en los estudios de D. Beyer y T. Lemberger, CPA-Seq encuentra un 59,66 % de errores mientras que CBMC un 55,7 %. [13]

El otro verificador es ESBMC-incr, una variación de CBMC con algunas mejoras. Entre ellas, aplica una variación del bounded model checking en el que se basa CBMC pero incremental (no fijo), lo cual es mucho mas exhaustivo a la hora de buscar errores. En comparación con CBMC y CPA-Seq, ESBMC-incr encuentra 63,69 % de errores [13], consolidándose como la mejor.

7. Limitaciones Encontradas

En el análisis de CBMC realizado, se pudieron evidenciar limitaciones de la herramienta. Algunas de estas son inherentes a la técnica de Bounded Model Checking que utiliza, como por ejemplo la necesidad de definir una cota fija para el análisis y con ello la posibilidad de no encontrar errores o tener falsos positivos si no se hace una elección adecuada. Además, esto le imposibilita realizar análisis exhaustivos en programas con loops infinitos o con estructuras de datos dinámicas complejas, ya que la cota elegida puede no ser suficiente para cubrir todos los caminos posibles. O, si esta es demasiado alta, puede llevar a un tiempo y consumo de CPU y memoria excesivos, lo que la hace inviable para sistemas grandes o complejos.

Una limitante adicional de CBMC es la compatibilidad reducida con C++ 17 y versiones posteriores, lo que puede dificultar su uso en proyectos modernos que utilizan características avanzadas y específicas del lenguaje. En nuestro caso de uso, por ejemplo, durante el desarrollo del modelo tuvimos inconvenientes respecto a la compatibilidad de la herramienta con variables de tipo `std::string` y pasar las mismas como parámetros a funciones. Finalmente, se optó por usar arreglos de caracteres sin signo (`unsigned char[]`) para evitar estos problemas.

Por otro lado, en estudios sobre verificación de equivalencia entre programas, se han identificado limitaciones al comprobar que dos versiones de una función se comportan de manera equivalente cuando estas reciben punteros como argumentos. Esta dificultad se debe a la complejidad inherente del análisis simbólico de punteros, especialmente en escenarios con aliasing, aritmética de punteros o accesos indirectos a memoria [17]. Además, en otros estudios se ha observado que CBMC tiene un bajo rendimiento en la verificación de problemas relacionados con arreglos y vectores de bits [13].

8. Conclusión

En este informe presentamos un análisis general de CBMC. Abordamos su funcionamiento interno, su usabilidad desde la línea de comandos, sus aplicaciones en distintos dominios, su desempeño en comparación con otras herramientas y algunas de sus limitaciones. Además, incluimos un caso de estudio en criptografía para evidenciar su versatilidad.

Su enfoque basado en Bounded Model Checking (BMC) y las transformaciones que aplica para traducir código a su representación lógica para luego aplicar SAT/SMT solvers, la convierte en una herramienta eficaz para la detección de errores comunes. Además, no solo permite verificar propiedades estándares de seguridad, sino también que el usuario pueda definir aserciones personalizadas que considere relevantes para el software específico que se esté analizando, lo cual amplía significativamente su aplicación.

Cabe destacar también su facilidad de uso desde la línea de comandos: las flags son directas y permiten configurar rápidamente distintos modos de análisis. No es necesario descargar herramientas extras para su uso, y viene instalado por defecto en algunas distribuciones de Linux (Ubuntu y Debian), por lo que es fácil de acceder a la misma (en caso de Windows o MacOS, su instalación es muy simple).

Más allá de su uso tradicional en la verificación formal de software, CBMC ha demostrado ser útil en una variedad de dominios, desde la verificación de controladores en sistemas embebidos hasta aplicaciones de criptografía o biología. Esto queda evidenciado en el caso de estudio presentado en este informe.

Sin embargo, es importante tener en cuenta sus limitaciones, como la necesidad de definir una cota fija para el análisis y su compatibilidad reducida con versiones recientes de C++. Estas limitaciones pueden afectar su aplicabilidad en ciertos contextos, especialmente en sistemas complejos o modernos que utilizan características avanzadas del lenguaje.

En conclusión, CBMC no solo destaca por su eficiencia, si no también por su flexibilidad y lo accesible que es, lo que la convierte en una valiosa opción tanto para proyectos de la academia como para algunos de la industria.

Referencias

1. Angeletti, D., Giunchiglia, E., Narizzano, M., Puddu, A., Sabina, S.: Automatic test generation for coverage analysis using CBMC. In: Moreno-Díaz, R., Pichler, F., Arencibia, A.Q. (eds.) Computer Aided Systems Theory - EUROCAST 2009, 12th International Conference, Las Palmas de Gran Canaria, Spain, February 15-20, 2009, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5717, pp. 287–294. Springer (2009). https://doi.org/10.1007/978-3-642-04772-5_38, https://doi.org/10.1007/978-3-642-04772-5_38
2. Becker, M., Metta, R., Venkatesh, R., Chakraborty, S.: Scalable and precise estimation and debugging of the worst-case execution time for analysis-friendly processors: a comeback of model checking. Int. J. Softw. Tools Technol. Transf. **21**(5), 515–543 (2019). https://doi.org/10.1007/S10009-018-0497-2, https://doi.org/10.1007/s10009-018-0497-2

3. Bhattacharyya, A., Gupta, A., Kuppusamy, L., Mani, S., Shukla, A., Srivas, M.K., Thattai, M.: A formal methods approach to predicting new features of the eukaryotic vesicle traffic system. Acta Informatica **58**(1-2), 57–93 (2021). https://doi.org/10.1007/S00236-019-00357-3, https://doi.org/10.1007/s00236-019-00357-3
4. Bucur, D., Kwiatkowska, M.Z.: Software verification for tinyos. In: Abdelzaher, T.F., Voigt, T., Wolisz, A. (eds.) Proceedings of the 9th International Conference on Information Processing in Sensor Networks, IPSN 2010, April 12-16, 2010, Stockholm, Sweden. pp. 400–401. ACM (2010). https://doi.org/10.1145/1791212.1791274, https://doi.org/10.1145/1791212.1791274
5. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems , 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15, https://doi.org/10.1007/978-3-540-24730-2_15
6. Dörre, F., Klebanov, V.: Practical detection of entropy loss in pseudo-random number generators. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 678–689. ACM (2016). https://doi.org/10.1145/2976749.2978369, https://doi.org/10.1145/2976749.2978369
7. Edmund Clarke, D.K.: Ansi-c bounded model checker user manual. https://www.cprover.org/cbmc/doc/cbmc-techreport.pdf/, accessed: May 28, 2025
8. Han, Z., He, F.: Robustness verification for checking crash consistency of non-volatile memory. In: Eeckhout, L., Smaragdakis, G., Liang, K., Sampson, A., Kim, M.A., Rossbach, C.J. (eds.) Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025. pp. 955–969. ACM (2025). https://doi.org/10.1145/3669940.3707269, https://doi.org/10.1145/3669940.3707269
9. K., K., Rebeiro, C., Hazra, A.: An algorithmic approach to formally verify an ECC library. ACM Trans. Design Autom. Electr. Syst. **23**(5), 63:1–63:26 (2018). https://doi.org/10.1145/3224205, https://doi.org/10.1145/3224205
10. Kroening, D.: Cbmc: Bounded model checking for ansi-c. https://www.cprover.org/cbmc/doc/cbmc-slides.pdf, accessed: May 28, 2025
11. Kroening, D.: Cbmc bounded model checking for c. https://www.cprover.org/cbmc/, accessed: May 28, 2025
12. Kroening, D., Schrammel, P., Tautschnig, M.: CBMC: the C bounded model checker. CoRR **abs/2302.02384** (2023). https://doi.org/10.48550/ARXIV.2302.02384, https://doi.org/10.48550/arXiv.2302.02384
13. Lemberger, T.: Towards cooperative software verification with test generation and formal verification. Ph.D. thesis, Ludwig Maximilian University of Munich, Germany (2022), https://edoc.ub.uni-muenchen.de/32852/
14. Menéndez, H.D., Jahangirova, G., Sarro, F., Tonella, P., Clark, D.: Diversifying focused testing for unit testing. ACM Trans. Softw. Eng. Methodol. **30**(4), 44:1–44:24 (2021). https://doi.org/10.1145/3447265, https://doi.org/10.1145/3447265
15. Orvalho, P., Janota, M., Manquinho, V.: Cfaults: Model-based diagnosis for fault localization in C programs with multiple test cases. CoRR **abs/2407.09337**

(2024). https://doi.org/10.48550/ARXIV.2407.09337, https://doi.org/10.48550/arXiv.2407.09337
16. Post, H., Küchlin, W.: Integrated static analysis for linux device driver verification. In: Davies, J., Gibbons, J. (eds.) Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4591, pp. 518–537. Springer (2007). https://doi.org/10.1007/978-3-540-73210-5_27, https://doi.org/10.1007/978-3-540-73210-5_27
17. Post, H., Sinz, C.: Proving functional equivalence of two AES implementations using bounded model checking. In: Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009. pp. 31–40. IEEE Computer Society (2009). https://doi.org/10.1109/ICST.2009.39, https://doi.org/10.1109/ICST.2009.39
18. Rabinovitz, I., Grumberg, O.: Bounded model checking of concurrent programs. In: Etessami, K., Rajamani, S.K. (eds.) Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3576, pp. 82–97. Springer (2005). https://doi.org/10.1007/11513988_9, https://doi.org/10.1007/11513988_9
19. Sampath, P., Rajeev, A.C., Ramesh, S.: Translation validation for stateflow to C. In: The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014. pp. 23:1–23:6. ACM (2014). https://doi.org/10.1145/2593069.2593237, https://doi.org/10.1145/2593069.2593237
20. Shaikh, S.A., Krishnan, P.: A framework for analysing driver interactions with semi-autonomous vehicles. In: Ölveczky, P.C., Artho, C. (eds.) Proceedings First International Workshop on Formal Techniques for Safety-Critical Systems, FTSCS 2012, Kyoto, Japan, November 12, 2012. EPTCS, vol. 105, pp. 85–99 (2012). https://doi.org/10.4204/EPTCS.105.7, https://doi.org/10.4204/EPTCS.105.7
21. Staats, M., Heimdahl, M.P.E.: Partial translation verification for untrusted code-generators. In: Liu, S., Maibaum, T.S.E., Araki, K. (eds.) Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27-31, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5256, pp. 226–237. Springer (2008). https://doi.org/10.1007/978-3-540-88194-0_15, https://doi.org/10.1007/978-3-540-88194-0_15
22. Stallings, W.: Cryptography and network security - principles and practice (3. ed.). Prentice Hall (2003)
23. University of Oxford: The cprover manual. https://www.cprover.org/cprover-manual/, accessed: May 28, 2025